Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Contents

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

PCI Express Port Bus Driver Support for Linux

Tom Long Nguyen, Dely L. Sy, & Steven Carbonari
*Intel® Corporation**
{tom.l.nguyen, dely.l.sy, steven.carbonari}@intel.com

Abstract

PCI Express®[1] is a high performance general purpose I/O Interconnect defined for a wide variety of computing and communication platforms. It defines PCI Express Ports and switches to provide a fabric based point-to-point topology. PCI Express categorizes PCI Express Ports into three types: the Root Ports, the Switch Upstream Ports, and the Switch Downstream Ports. Each PCI Express Port can provide up to four distinct services: native hot-plug, power management, advanced error reporting, and virtual channels[1][3]. To fit within the existing Linux®[2] PCI Driver Model but provide a clean and modular solution, in which each service driver can be built and loaded independently, requires the PCI Express Port Bus Driver architecture. The PCI Express Port Bus Driver initializes all services and distributes them to their corresponding service drivers. This paper is targeted toward kernel developers and architects interested in the details of enabling service drivers for PCI Express Ports. The i386 Linux implementation will be used as a reference model to provide insight into the implementation of the PCI Express Port Bus Driver and specific service drivers like the advanced error reporting root service driver and the native hot-plug root service driver.

1 Introduction

The Linux PCI Driver Model restricts a device to a single driver. Drivers in Linux are loaded based off the PCI Device ID and function. Once a driver is loaded, no other drivers for that device can be loaded[2]. Referring to Figure 1, if the Root Port hot-plug driver is loaded first, it claims the Root Port device. The Linux PCI Driver Model therefore prevents the support of multiple services per PCI Express Port using individual service drivers.

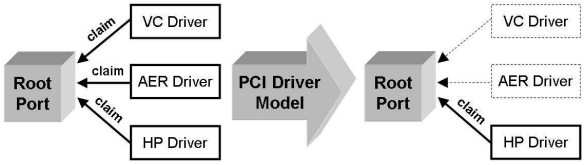

Figure 1: Service Drivers under the Linux PCI Driver Model

A PCI Express Port may have multiple distinct services operating independently. A PCI Express Port is not required to support all services, so some PCI Express Ports within a PCI Express hierarchy may support none, some or all the services. A possible solution is to implement a single driver to handle all services

*Intel is a trademark or registered trademark of Intel Corporation in the United States, other countries, or both. This work represents the view of the authors and does not necessarily represent the view of Intel.

[1]PCI Express is a trademark of the Peripheral Component Interchange Special Interest Group (PCI-SIG)

[2]Linux is a registered trademark of Linus Torvalds

per PCI Express Port. However, this solution would lack the ability to have each service built and loaded independently from each other, preventing extensibility for addition of future services and the ability to have a service driver loaded on more than one PCI Express Port. Separate service drivers are required to support addition of new features and loading of services based on the PCI Express Port capabilities.

To support multiple drivers per device without changing the existing Linux PCI Driver Model requires a new architecture that fits within the existing Linux PCI Driver Model but provides the flexibility required to support multiple service drivers per PCI Express Port. As shown in Figure 2, the PCI Express Port Bus Driver (PBD)[5] fits into the existing Linux PCI Driver Model while providing an interface to allow multiple independent service drivers to be loaded for a single PCI Express Root Port. The PBD acts as a service manager that owns all services implemented by the Ports. Each of these services is then distributed and handled by a unique service driver. The PBD achieves the following key advantages:

- Allows multiple service drivers to run simultaneously and independently from each other and to service more than one PCI Express Port.
- Allows service drivers to be designed and implemented in a modular fashion.
- Centralizes management and distribution of resources of the PCI Express Port devices to requested service drivers.

This paper describes the PCI Express Port Bus Driver architecture. Following the port bus driver architecture are two examples of service drivers. The first example is the advanced error reporting service driver that was designed to work with the port bus architecture. The second example is the hot-plug service driver that was originally designed as an independent driver then converted to a service driver to operate with the Port Bus Driver. Lastly, an overview of the impact to device drivers and future service drivers is outlined.

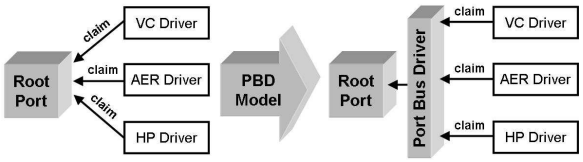

Figure 2: Service Drivers under the PBD

2 PCI Express Port Bus Driver

2.1 PCI Express Port Topology

To understand the Port Bus Driver architecture, it helps to begin with the basics of PCI Express Port topology. Figure 3 illustrates two types of PCI Express Port devices: the Root Port and the Switch Port. The Root Port originates a PCI Express Link from a PCI Express Root Complex. The Switch Port, which has its secondary bus representing switch internal routing logic, is called the Switch Upstream Port. The Switch Port which is bridging from switch internal routing buses to the bus representing the downstream PCI Express Link is called the Switch Downstream Port[1].

Each PCI Express Port device can be implemented to support up to four distinct services: native hot plug (HP), power management event (PME), advanced error reporting (AER), virtual channels (VC). The PCI Express services discussed are optional, so in any given PCI Express hierarchy a port may support none, some, or all of the services.

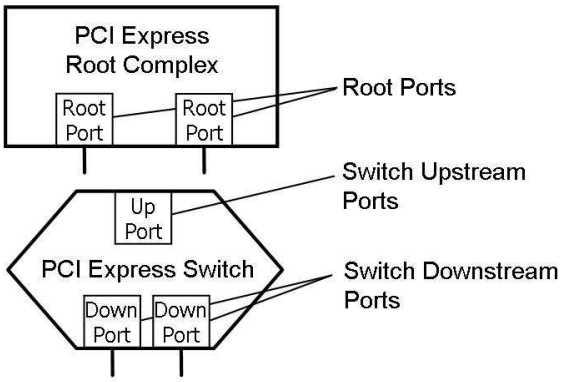

Figure 3: PCI Express Port Topology

2.2 PCI Express Port Bus Driver Architecture

The design of the PCI Express Port Bus Driver achieves a clean and modular solution in which each service driver can be built and loaded independently from each other. The PCI Express Port Bus Driver serves as a service manager that loads and unloads the service drivers accordingly, as illustrated in Figure 4.

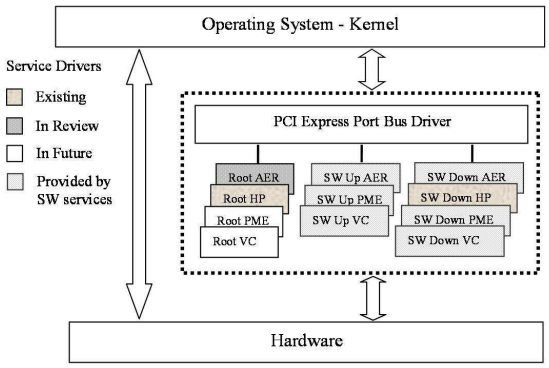

Figure 4: PCI Express Port Bus Driver System View

The PCI Express Port Bus Driver is a PCI-PCI Bridge device driver, which attaches to PCI Express Port devices. For each PCI Express Port device, the PCI Express Port Bus Driver searches for all possible services, such as native HP, PME, AER, and VC, implemented by PCI Express Port device. For each service found, the PCI Express Port Bus Driver creates a corresponding service device, named pcieXY where X indicates the PCI Express Port type and Y indicates the PCI Express service type as described in Table 1, and then registers this service device into a system device hierarchy. Figure 5 shows an example of how the PCI Express Port Bus Driver creates service devices on a system populated with two Root Port devices, one Switch Upstream Port device, and two Switch Downstream Port devices.

Port Type (X)	Service Type (Y)	Service Entity Description (pcieXY)
0	0	PME service on PCI Express Root Port (PMErs)
0	1	AER service on PCI Express Root Port (AERrs)
0	2	HP service on PCI Express Root Port (HPrs)
0	3	VC service on PCI Express Root Port (VCrs)
1	0	PME service on PCI Express Switch Upstream Port (PMEus)
1	1	AER service on PCI Express Switch Upstream Port (AERus)
1	2	Not a supported PCI Express configuration
1	3	VC service on PCI Express Switch Upstream Port (VCus)
2	0	PME service on PCI Express Switch Downstream Port (PMEds)
2	1	AER service on PCI Express Switch Downstream Port (AERds)
2	2	HP service on PCI Express Switch Downstream Port (HPds)
2	3	VC service on PCI Express Switch Downstream Port (VCds)

Table 1: Service Entity Description

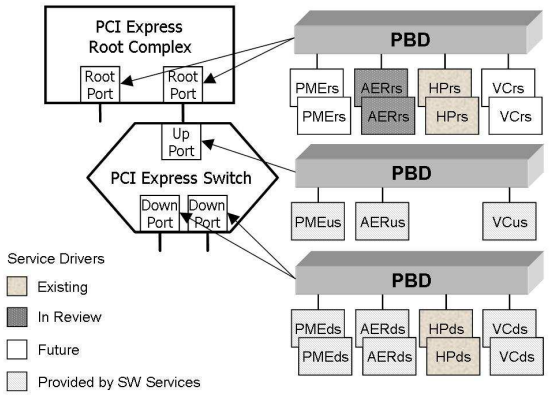

Figure 5: Service Devices in a PCI Express Port Bus Driver Architecture

Once service devices are discovered and added in the system device hierarchy, a service driver is loaded accordingly if it registers its service with the PCI Express Port Bus Driver. The PCI Express Port Bus Driver provides an interface, named `pcie_port_service_register`, to allow a service driver to register its service[4]. The registration enables the user

to configure services during kernel configuration regardless of HW support. It enables debugging and adding of new services in a modular fashion. When a service driver calls `pcie_port_service_register`, the PCI Express Port Bus Driver loads a service driver by invoking the PCI subsystem, which walks through a system device hierarchy for a service match. If the port bus finds a match, it loads a service driver for that service device.

In addition, the PCI Express Port Bus Driver provides `pcie_port_service_unregister`, to undo the effects of calling function `pcie_port_service_register` [4]. Note that a service driver should always call `pcie_port_service_unregister` when a service driver is unloading.

2.3 The Service Driver

To maintain modularity in the PCI Express Port Bus Driver design, individual service drivers are required. In some cases a driver may already exist for a PCI Express Port. In these instances the driver must be ported to the service driver to allow other service drivers to load on the PCI Express Port. To port drivers to service drivers, the following three basic steps are required. Refer to Sections 3.1.1 to 3.1.3 for a specific example.

- Specify service ID. The PCI Express Port Bus Driver defines the data structure of service ID similar to the data structure of `pci_device_id` except with two additional fields: the `port_type` and `service_type` fields as described in Table 1. Note that failure to specify a correct service ID will prevent the port bus from loading a service driver.
- Initialize service driver. The PCI Express Port Bus Driver defines the data structure of service driver similar to the `pci_driver` data structure. The pointer to the `pci_dev` data structure is replaced with a pointer to the `pcie_device` data structure in each callback function.
- Call `pcie_port_service_register` instead `pci_register_driver`.

Once a service driver is loaded, a service driver should always configure and initialize its own capability structure and required IOs to operate normally without any support from the PCI Express Port Bus Driver. However, a service driver is prohibited from doing the following:

- Switch the interrupt mode on a device. The interrupt mode can be INTx legacy, MSI or MSI-X. A service driver should always use the assigned service IRQ to call `request_irq` to have its software interrupt service routine hookup. Note that the assigned service IRQ may be shared among service drivers; therefore, a service driver should always treat this assigned service IRQ as shared interrupt.
- Access resources that are not directly required by the service. For example, the advanced error reporting service driver is prohibited from accessing any configuration registers other than the Advanced Error Reporting Capability structure. A service driver uses the `port` pointer, a member of the `pcie_device` data structure defined by PBD, to access PCI configuration and memory mapped IO space.
- Call `pci_enable_device` and `pci_set_master` functions. This is no longer necessary because these functions now get called by the PCI Express Port Bus Driver.

2.4 Resource Allocation and Distribution

Service drivers must adhere to the guidelines in this document to deal with resource allocation and distribution. Since all service drivers of a PCI Express Port device are allowed to run simultaneously, a decision of which driver (Port Bus Driver vs. service driver) owns which resource is described in Sections 2.4.1 to 2.4.3. These resources include the MSI capability structure, the MSI-X capability structure, and PCI IO resources.

2.4.1 The MSI Capability Structure

The MSI capability structure enables a device software driver to call `pci_enable_msi` to request an MSI based interrupt. Once MSI is enabled on a device, it stays in this mode until a device driver calls `pci_disable_msi` to return to INTx emulation. Since each service driver runs independently from each other, changing the interrupt mode on the PCI Express Port by an individual service driver may result in unpredictable behavior. Each service driver is therefore prohibited from calling these APIs. The PCI Express Port Bus Driver is responsible for determining the interrupt mode and assigning the service IRQ to each service device accordingly. A service driver must use its service vector when calling `request_irq`/`free_irq`.

2.4.2 The MSI-X Capability Structure

Similar to MSI a device driver for an MSI-X capable device can call `pci_enable_msix` to request MSI-X interrupts. The key difference is that the MSI-X capability structure enables a PCI Express Port device to generate multiple messages. Managing multiple MSI-X vectors is handled by the PCI Express Port Bus driver. The PCI Express Port Bus Driver is responsible for determining the interrupt mode transparent to the service drivers. A service driver must use its service vector when calling `request_irq`/`free_irq`.

If a PCI Express Port device supports MSI-X, the PCI Express Port Bus Driver will request the number of MSI-X messages equal to the number of supported services for the device. This allows each service to have it own hardware interrupt resource independently generated from other services.

2.4.3 PCI IO Resources

PCI IO resources include PCI memory/IO ranges and PCI configuration registers are assigned by BIOS during boot. For PCI memory/IO ranges, the service driver is responsible for initializing its PCI memory/IO maps during service startup. There is possibly where the PCI memory/IO ranges are shared. If this occurs, each service driver is responsible for mapping its PCI memory/IO regions without overstepping on resources of others. The PCI Express Port Bus Driver does not arbitrate access to the regions and assumes service drivers to be well behaved.

For PCI configuration registers, each service runs PCI configuration operation on its own capability structure except the PCI Express capability structure, in which the Device Control register and the Root Control register have unique control bits assigned to AER service and PME service. A read-modify-write should always be handled by the AER/PME service driver. Again this paper assumes that all service drivers are responsible for not overstepping on resources of others.

3 PCI Express Advanced Error Reporting Root Service Driver

PCI Express error signaling can occur on the PCI Express link itself or on behalf of transactions initiated on the link. PCI Express defines the Advanced Error Reporting capability, which is implemented with the PCI Express Advanced Error Reporting Extended Capability Structure, to allow a PCI Express component (agent) to send an error reporting message to the Root Port. The Root Port, a host receiver of all error messages associated with its hierarchy, decodes an error message into an error type and an agent ID and then logs these into its PCI Express Advanced Error Reporting Extended Capability Structure. Depending on whether an error reporting message is enabled in the Root Error Command Register, the Root Port device generates an interrupt if an error is detected[1]. The PCI Express advanced error reporting service driver is implemented to service AER interrupts generated by the Root Ports[6].

Once the PCI Express advanced error reporting service driver is loaded, it claims all AER Root service devices in a system device hierarchy, as shown in Figure 6. For each AERrs service device, the advanced error reporting service driver configures its service device to generate an interrupt when an error is detected. For each detected error, the advanced error reporting service driver performs the followings[6]:

- Gather comprehensive error information.
- Guide error recovery associated with the hierarchy in question based on the comprehensive error information gathered.
- Report error to user in a format with more precise what error type and severity.

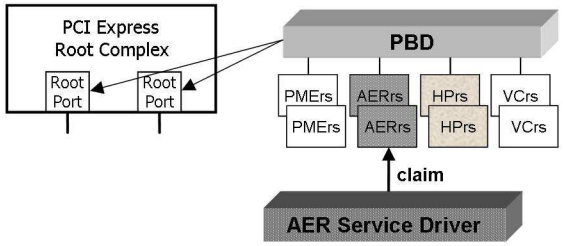

Figure 6: AER Root Service Driver

3.1 Register AER Service

The advanced error reporting service driver is implemented based on the service driver framework as defined in Section 2.3. Sections 3.1.1 to 3.1.3 below illustrate how the advanced error reporting service driver follows three basic steps as required.

3.1.1 Specify AER Service ID

Since the PCI Express advanced error reporting service driver is implemented to serve only the Root Ports, the data structure of AER service ID is defined below[7]:

```
static struct pcie_port_service_id aer_id[]={{
  .vendor = PCI_ANY_ID,
  .device = PCI_ANY_ID,
  .port_type = PCIE_RC_PORT,
  .service_type = PCIE_PORT_SERVICE_AER,
  }, {}
};
```

3.1.2 Initialize AER Service Driver

Once the AER service ID is defined, the advanced error reporting service driver initializes the service callbacks as defined in the `pcie_port_service_driver` data structure. The data structure of service callbacks is defined below[7]:

```
static struct pcie_port_service_driver aerdrv={
  .name = "aer",
  .id_table = &aer_id[0],

  .probe = aer_probe,
  .remove = aer_remove,

  .suspend = aer_suspend,
  .resume = aer_resume,
};
```

3.1.3 Calling `pcie_port_service_register`

The final step in initialization of the advanced error reporting service driver is calling function `pcie_port_service_register` to register AER service with the PBD. During driver initialization, the module routine is called for initialization when the kernel calls the advanced error reporting service driver. Calling `pcie_port_service_register`/`pcie_port_service_unregister` should always be done in `module_init`/`module_exit` as depicted below[7]:

```
static int __init aer_service_init(void)
{
  return pcie_port_service_register(&aerdrv);
}

static void __exit aer_service_exit(void)
{
  pcie_port_service_unregister(&aerdrv);
}

module_init(aer_service_init);
module_exit(aer_service_exit);
```

Figure 7 depicts the state diagram once the advanced error reporting service driver's module routine is called.

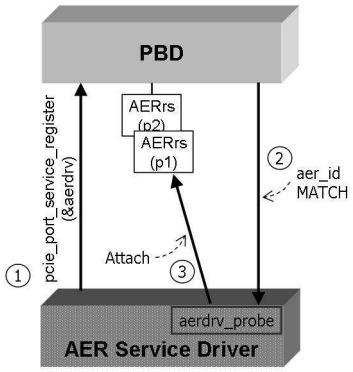

Figure 7: State Diagram of Registering AER Service with PBD

4 PCI Express Native Hot-Plug Service Driver

The PCI Express Hot-Plug standard usage model is derived from the standard usage model defined in the PCI Standard Hot-Plug Controller and Subsystem Specification, Rev. 1.0[8].

4.1 PCI Express Native Hot Plug Features

PCI Express Native Hot-Plug features are:

- Root ports and downstream ports of switches are hot-pluggable ports in PCI Express hierarchy.
- Interrupt driven hot plug events will result in hot-plug interrupts.
- Hot plug registers are part of the PCI Express Capability register set; hot-plug operations are invoked by writing to these registers.
- Based on SHPC usage model, but not the bus centric SHPC register set.

4.2 Porting the PCI Express Hot-Plug Driver to a Service Driver

As mentioned in Section 2.2, the PCI Express Port Bus Driver provides a mechanism for a

service driver to register its service. If the requested service is found in a service device hierarchy, the service driver can successfully load. This section focuses on showing what the changes are required to port the PCI Express native hot-plug driver to a service driver.

4.2.1 Registering the Hot Plug Service Driver

The pciehp driver calls `pcie_port_service_register (struct pcie_port_service_driver *driver)` to register its hot-plug service with the PBD. The pciehp driver is responsible for setting up the data structures before calling `pcie_port_service_register`. Below shows the difference in the data structures used when the driver is used as a standard driver or as a service driver[9].

```
+ static struct pcie_port_service_id
+   port_pci_ids[] = {{
+   .vendor = PCI_ANY_ID,
+   .device = PCI_ANY_ID,
+   .port_type = PCIE_ANY_PORT,
+   .service_type = PCIE_PORT_SERVICE_HP,
+   .driver_data = 0,
+ }, { /* end: all zeroes */ }
+ };

- static struct pci_device_id pcied_pci_tbl[]={
- {
-   .class = ((PCI_CLASS_BRIDGE_PCI << 8) |
-   0x00),
-   .class_mask = ~0,
-   .vendor = PCI_ANY_ID,
-   .device = PCI_ANY_ID,
-   .subvendor = PCI_ANY_ID,
-   .subdevice = PCI_ANY_ID,
- }, { /* end: all zeroes */ }
- };
```

4.2.2 Initialize the Hot-Plug Service Driver

Once the HP service ID is defined, the service driver initializes the service callbacks as defined in the `pcie_port_service_driver` data structure. The following shows the changes that need to be made in porting the PCI Express hot-plug driver to a service driver[9].

```
+ static struct pcie_port_service_driver
+   hpdriver_portdrv = {
+   .name = "hpdriver",
+   .id_table = &port_pci_ids[0],
+   .probe = pciehp_probe,
+   .remove = pciehp_remove,
+   .suspend = pciehp_suspend,
+   .resume = pciehp_resume,
+ };

- static struct pci_driver pcie_driver = {
-   .name = "pciehp",
-   .id_table = pcied_pci_tbl,
-   .probe = pcie_probe,
-   .remove = pcie_remove,
- };
```

4.2.3 Calling `pcie_port_service_register` API

The final step in initialization of the HP service driver is calling `pcie_port_service_register` to register HP service with the PBD. The following shows the changes that need to be made in the standalone driver to port it to a service driver[9].

```
static int __init pcied_init(void)
{
     :
+   retval = pcie_port_service_register(
+   &hpdriver_portdrv);
-   retval = pci_register_driver(
-   &pcie_driver);
    :
}

static void __exit pcied_cleanup(void)
{
    :
+   pcie_port_service_unregister(
+   &hpdriver_portdrv);
-   pci_unregister_driver(&pcie_driver);
    :
}
```

Figure 8 depicts the state diagram once HP service driver's module routine is called.

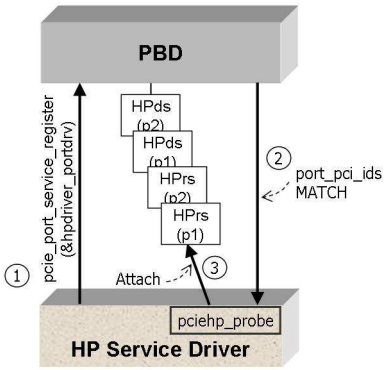

Figure 8: State Diagram of Registering HP Service with PBD

4.2.4 Available Resources

As a service driver, dev->irq is provided by the PCI Express Port Bus Driver and is passed to the pciehp driver. Whether dev->irq is a regular system interrupt, MSI or MSI-X, the PCI Express Port Bus Driver assigns the value to it. The pciehp service driver does not need to call `pci_enable_msi` to request use of MSI/MSI-X if the OS supports that.

5 Impacts to PCI Express Drivers

The Port Bus Driver design does not directly impact existing PCI Express endpoint device drivers. However, a service driver may impact a PCI Express endpoint driver. Additional PCI Express services may require endpoint driver changes to take full advantage of the new functionality. For example, to take full advantage of AER error recovery will require drivers to support the AER callback API. Driver writers for PCI Express components should be well versed with this architecture and evaluate driver impacts as new services (VC or PME) become available.

The Port Bus perspective impacts device drivers for PCI Express Switch components. The PCI Express Port Bus Driver claims all PCI Express Ports in a system device hierarchy, including ports in a PCI Express switch. Switch service drivers must follow the port bus driver framework. Switch vendors can use existing root service drivers as a reference while writing their own service drivers.

When developing a switch service driver the usage model at each level in the PCI Express hierarchy needs to be considered. A service driver for a downstream switch port may be required to provide different functionality than a similar root port service driver. For example, the AER Root service driver cannot be reused `as-is`. The usage model is different. AER Switch service driver should provide error-handling callbacks and AER initialization of the switch, while the AER Root service driver provides the primary mechanism to handle the error recovery process. However, in the case of the hot-plug driver, the same service driver may be used for both the Root Ports and the Switch Downstream Ports because the hot-plug usage model is identical.

6 Conclusion

The design of the PCI Express Port Bus Driver delivers a clean and modular solution to support the multiple features of PCI Express while remaining compatible with the Linux Driver Model. Each feature can have its own software service driver that can be built and loaded as a separate module. In addition when/if future PCI Express features come available or are added to future specification revisions, the PCI Express Port Bus architecture is extensible to support those additions. The PCI Express Port Bus Driver and changes to port the native PCI Express hot-plug driver has been incorporated Linux kernel version 2.6.11. The advanced error reporting service driver is currently under review on the LKML.

7 Acknowledgements

Special thanks to Greg Kroah-Hartman for his contributions to the architecture design of PCI Express Port Bus driver.

References

[1] PCI Express Base Specification Revision 1.1. March 28, 2005. `http://www.pcisig.com/specifications/pciexpress/`.

[2] Linux Device Drivers, 3rd Edition. Publisher: O'Reilly & Associates; 3 edition (February 10, 2005) by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman.

[3] Renato John Recio. Promises and Reality: Server I/O networks, past, present, and future. In Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications. Pages 163-178, Karlsruhe, Germany, August 2003.

[4] PCIEBUS-HOWTO.txt. Available from: `2.6.11/Documentation`.

[5] PCI Express Port Bus Driver code. Available from: `2.6.11/drivers/pci/pcie`.

[6] PCIEAER-HOWTO.txt, under review. If being accepted: `2.6.x/Documentation`.

[7] PCI Express advanced error reporting driver code, under review. If accepted: 2.6.x/drivers/pci/pcie/aer.

[8] PCI Standard Hot-Plug Controller and Subsystem Specification Revision 1.0. June 20, 2001. `http://www.pcisig.com/specifications/conventional/pci_hot_plug/SHPC_10/`.

[9] PCI Express hot-plug driver code. Available from: `2.6.11/drivers/pci/hotplug`.

pktgen the linux packet generator

Robert Olsson
Uppsala Universitet & SLU
robert.olsson@its.uu.se

Abstract

pktgen is a high-performance testing tool included in the Linux kernel. pktgen is currently the best tool to test the TX process of device driver and NIC. pktgen can also be used to generate ordinary packets to test other network devices. Especially of interest is the use of pktgen to test routers or bridges which often also use the Linux network stack. Because pktgen is "in-kernel," it can generate high bandwith and very high packet rates to load routers, bridges, or other network devices.

1 Introduction

This paper describes the novel rework of pktgen in Linux 2.6.11. Much of the rework has been focused on multi-threading and SMP support. The main goal is to have one pktgen thread per CPU which can then drive one or more NICs. An in-kernel pseudo driver offers unique possibilities in performance and capabilities. The trade-off is additional responsibility in terms of robustness and avoiding kernel bloat (vs user mode application).

Pktgen is not an all-in-one testing tool. It offers a very efficient direct access to the host system NIC driver/chip TX-process and bypasses most of the Linux networking stack. Because of this, use of pktgen requires root access. The packet stream generated by pktgen can be used as input to other network devices. Pktgen also exercises other subsystems such as packet memory allocators and I/O buses. The author has done tests sending packets from memory to several GIGE interfaces on different PCI-buses using several CPU's. Aggregate Rates > 10 GBit/s have been seen.

1.1 Other testing tools

There are lots of good testing tools for network and TCP testing. netperf and ttcp are probably among the most widespread. Pktgen is not a substitute for those tools but complements for some types of tests. The test possibilities is described later in this paper. Most importantly, pktgen cannot do any TCP testing.

2 Pktgen performance

Performance varies of course with hardware and type of test. Some examples follow. A single flow of 1.48 Mpps is seen with a XEON 2.67 GHz using a patched e1000 driver (64 byte packets). High numbers are also reported with bcm5703 with tg3 driver. Aggregated performance of >10 Gbit/s (1500 byte packets) comes from using 12 GIGE NIC's and DUAL XEON

2.67 MHz with hyperthreading enabled (motherboard has 4 independent PCI-X buses). Similarly, DUAL 1.6ăGHz Opterons can generate 2.4 Mpps (64 byte packets). Tests involving lots of alloc's results in lower sending performance (see `clone_skb()`).

Many other things also affect performance: PCI bus speed, PCI vs PCI-X, PCI-PCI Bridge, CPU speed, memory latency, DMA latency, number of MMIO reads/writes per packet or per interrupt, etc.

Figure 1 compares performance of Intel's DUAL Port NIC (2 x 82546EB) with Intel's QUAD NIC (4 x 82546EB; Secondary PCI-X Bus runs at 120 Mhz). on a Dual Opteron 242 (Linux 2.6.7 32-bit).

The graph shows a faster I/O bus gives higher performance as this probably lowers DMA latency. The effects of the PCI-X bridge are also evident as the bridge is the difference between the DUAL and QUAD boards.

It's interesting to note that even bus bandwidth is much faster than 1 Gbit/s it degrades the small packet performance as seen from the experiment. 133 MHz would theoretically correspond to 8.5 Gbit/s. The patched version of e1000 driver adds data prefetching and skb refill at `hard_xmit()`.

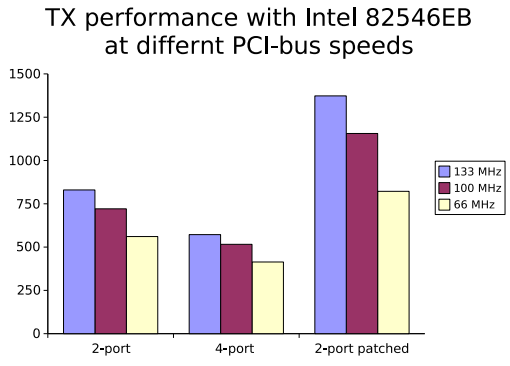

Figure 1: PCI Bus Topology vs TX perf

3 Getting pktgen to run

Enable `CONFIG_NET_PKTGEN` in the .config, compile and build pktgen.o either in-kernel or as module, insmod pktgen if needed. Once running, pktgen creates a kernel thread and binds thread to that CPU. One can the register a device to exactly one of those threads.This to give full control of the device to CPU relationship. Modern platforms allow interrupts to be assigned to a CPU (aka IRQ affinity) and this is necessary to minimize cache-line bouncing.

Generally, we want the same CPU that generates the packets to also take the interrupts given a symmetrical configuration (CPU:NIC is 1:1).

On a dual system we see two pktgen threads: [pktgen/0], [pktgen/1]

pktgen is controlled and monitored via the /proc file system. To help document a test configuration and parameters, shell scripts are recommended to setup and start a test. Again referring to our dual system, at start up the files below are created in ă `/proc/net/pktgen/` kpktgend_0, kpktgend_1, pgctrl

Assigning devices (e.g. eth1, eth2) to kpktgend_X thread, makes new instances of the devices show up in `/proc/net/pktgen/` to be further configured at the device level.

A test can be configured to run forever or terminate after a fixed number of packets. Ctrl-C aborts the run.

pktgen sends UDP packets to port 9 (discard port) by default. IP, MAC addresses, etc. can be configured. Pktgen packets can hence be identified within the kernel network stack for profiling and testing.

4 Pktgen versioninfo

The pktgen version is printed in dmesg when pktgen starts. Version info is also in `/proc/net/pktgen/pgctrl`.

5 Interrupt affinity

When adding a device to a specific pktgen thread, one should also set `/proc/irq/X/smp_affinity` to bind the NIC to the same CPU. This reduces cache line bouncing in several areas: when freeing skb's and in the NIC driver. The `clone_skb` parameter can in some cases mitigate the effect of cache line bouncing as skb's are not fully freed. One must experiment a bit to achieve maximum performance.

The irq numbers assigned to particular NICs can be seen in `/proc/interrupts`. In the example below, eth0 uses irq 26, eth1 uses irq 27 etc.

```
26: 933931      0 IO-APIC-level eth0
27: 936392      0 IO-APIC-level eth1
28:      8 936457 IO-APIC-level eth2
29:      8 939310 IO-APIC-level eth3
```

The example below assigns eth0, eth1 to CPU0, and eth2, eth3 to CPU1:

```
echo 1 > /proc/irq/26/smp_affinity
echo 1 > /proc/irq/27/smp_affinity
echo 2 > /proc/irq/28/smp_affinity
echo 2 > /proc/irq/29/smp_affinity
```

The graph below illustrates the performance effects of affinity assignment of PII system.

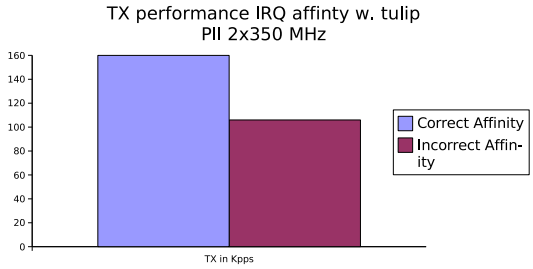

Figure 2: Effects of irq affinity

5.1 clone_skb: limiting memory allocation

pktgen uses a trick to increment the skb's refcnt to avoid full path of kfree and alloc when sending identical skb's. This generally gives very high sending rates. For Denial of Service (DoS) and flow tests this technique can not be used as each skb has to be modified.

The parameter `clone_skb` controls this functionality. Think of `clone_skb` as the number of packet clones followed by a master packet. Setting `clone_skb=0` gives no clones, just master packets, and `clone_skb=1000000` givs 1 master packet followed by one million clones.

`clone_skb` does not test normal use of a NIC. While the kfree and alloc are avoided by using `clone_skb`, one also avoids sending packets from dirty cachelines. The clean cache can contribute as much as 20% in performance as shown in Table 1.

Data in Table 1 was collected on HP rx2600-Itanium2 with BCM5703 (PCI-X) NIC running 2.6.11 kernel. The difference in performance between columns (RC on vs. off) shows how much dirty cache can affect DMA. Numbers are in packets per second. Read Current (RC) is a Mckinley bus transaction that allows the CPU to respond to a cacheline request directly from cache and retain ownership of the dirty cacheline. I.e., the cacheline can stay dirty-private

clone_skb	RC on	RC off	% Drop
on	947315	913768	–3.54%
off	630736	506711	–19.66%

Table 1: clone_skb and cache effects (pps)

and the CPU can write the same cacheline again without having to acquire ownership first.

It's likely cache effects contribute to the difference in performance between rows too (with and without `clone_skb`). But it's just as likely `clone_skb` reduces the CPU's use of memory bus bandwidth and thus contends less with DMA. This data is contributed by Grant Grundler.

5.2 Delay: Decreasing sending rate

pktgen can insert an extra artificial delay between packets. The unit is specified in nanoseconds. For small delays, pktgen busywaits before putting this skb on the TX-ring. This means traffic is still bursty and somewhat hard to control. Experimentation is probably needed.

6 Setup examples

Below a very simple example of pktgen sending on eth0. One only needs to bring up the link.

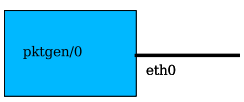

Figure 3: Just send/Link up

pktgen can send if the device is UP but many derives also requires that link is up can be done using a crossover cable connected to another NIC in the same box. If generated packets should be seen (i.e. Received) by the same host, set dstmac to match the NIC on the cross over cable as shown in Figure 4. Using a "fake" dstmac value (e.g. 0) will cause the other NIC to just ignore the packets.

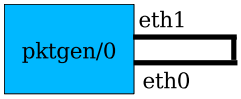

Figure 4: RX/TX in one Host

On SMP systems, it's better if the TX flow (pktgen thread) is on a different CPU from the RX flow (set IRQ affinity). One way to test Full Duplex functionality is to connect two hosts and point the TX flows to each other's NIC.

Next, the box with pktgen is used just a packet source to inject packets into a local or remote system. Note you need to configure dstmac of localhost or gateway appropriate.

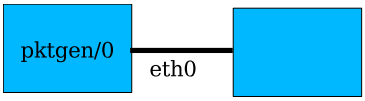

Figure 5: Send to other

Below pktgen in a forwarding setup. The sink host receives and discards packets. Of course, forwarding has to be configured on all boxes. It might be possible to use a dummy device instead of sink box.

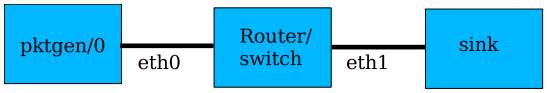

Figure 6: Forwarding setup

Forwarding setup using dual devices. Pktgen can use different threads to achieve high load in terms of small packets or concurrent flows.

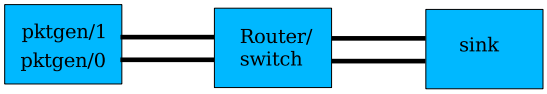

Figure 7: Parallel Forwarding setup

7 Viewing pktgen threads

Thread information as which devices are handled by this thread as actual status for each device is seen. `max_before_softirq` is used to avoid pktgen to avoid pktgen monopolize kernel resources. This will probably be removed as this of less problem with the threaded design. Result: is the "return" code "from the last `/proc` write.

/proc/net/pktgen/kpktgend_0

```
Name: kpktgend_0
max_before_softirq: 10000
Running:
Stopped: eth1
Result: OK: max_before_softirq=10000
```

7.1 Viewing pktgen devices

'Parm' sections holds configured info. 'Current' holds running stats. Result is printed after run or after interruption for example: See Appendix.

8 Configuring

Configuring is done via the /proc interface this is easiest done via scripts. Select a suitable script and customize. This paper includes one full example in Section 8. Additional example scripts are available from:

..1-1	# 1 CPU 1 dev
..1-2	# 1 CPU 2 dev
..2-1	# 2 CPU's 1 dev
..2-2	# 2 CPU's 2 dev
..1-1-rdos	# 1 CPU 1 dev route DoS
..1-1-ip6	# 1 CPU 1 dev ipv6
..1-1-ip6-rdos	# 1 CPU 1 dev ipv6 route DoS
..1-1-flows	# 1 CPU 1 dev multiple flows

Table 2: Script Filename Extensions

`ftp://robur.slu.se/pub/Linux/net-development/pktgen-testing/examples/`

Additional examples have been contributed by Grant Grundler `<grundler@parisc-linux.org>` `ftp://gsyprf10.external.hp.com/pub/pktgen-testing/`

See Appendix A for a quick-reference guide for currently implemented commands. It's divided into three parts: Pgcontrol, Threads, and Device. Each part has corresponding files in the `/proc` file system.

A collection of small tutorial scripts for pktgen are in examples dir. The file name extension is described in Table reffilename-ext.

Run in shell: ./pktgen.conf-X-Y

It does all the setup and then starts/stops TX thread. The scripts will need to be adjusted based on which NICs one wishes to test.

8.1 Configuration examples

Below is concentrated anatomy of the example scripts. This should be easy to follow.

pktgen.conf-1-2 A script fragment assigning eth1, eth2 to CPU on single CPU system.

```
PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth1"
pgset "add_device eth2"
```

pktgen.conf-2-2 A script fragment assigning eth1 to CPU0 respectivly eth2 to CPU1.

```
PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth1"

PGDEV=/proc/net/pktgen/kpktgend_1
pgset "rem_device_all"
pgset "add_device eth2"
```

pktgen.conf-2-1 A script fragment assigning eth1 and eth2 to CPU0 on a dual CPU system.

```
PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth1"
pgset "add_device eth2"

PGDEV=/proc/net/pktgen/kpktgend_1
pgset "rem_device_all"
```

pktgen.conf-1-2 A script fragment assigning eth1, eth2 to CPU on single CPU system.

```
PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth1"
pgset "add_device eth2"
```

pktgen.conf-1-1-rdos A script fragment for route DoS testing. Note clone_skb 0

```
PGDEV=/proc/net/pktgen/eth1
pgset "clone_skb 0"
# Random address with in the
# min-max range
# pgset "flag IPDST_RND"
pgset "dst_min 10.0.0.0"
pgset "dst_max 10.255.255.255"
```

pktgen.conf-1-1-ipv6 Setting device ipv6 addresses.

```
PGDEV=/proc/net/pktgen/eth1
pgset "dst6 fec0::1"
pgset "src6 fec0::2"
```

pktgen.conf-1-1-ipv6-rdos

```
PGDEV=/proc/net/pktgen/eth1
pgset "clone_skb 0"
# pgset "flag IPDST_RND"
pgset "dst6_min fec0::1"
pgset "dst6_max fec0::FFFF:FFFF"
```

pktgen.conf-1-1-flows A script fragment for route flow testing. Note clone_skb 0

```
PGDEV=/proc/net/pktgen/eth1
pgset "clone_skb 0"
# Random address within the
# min-max range
# pgset "flag IPDST_RND"
pgset "dst_min 10.0.0.0"
pgset "dst_max 10.255.255.255"
# 8k Concurrent flows at 4 pkts
pgset "flows 8192"
pgset "flowlen 4"
```

2x4+2 script

```
#Script contributed by Grant Grundler
# <grundler@parisc-linux.org>
# Note! 10 devices

PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth3"
pgset "add_device eth5"
pgset "add_device eth7"
pgset "add_device eth9"
pgset "add_device eth11"
pgset "max_before_softirq 10000"

PGDEV=/proc/net/pktgen/kpktgend_1
pgset "rem_device_all"
pgset "add_device eth2"
```

```
pgset "add_device eth4"
pgset "add_device eth6"
pgset "add_device eth8"
pgset "add_device eth10"
pgset "max_before_softirq 10000"

# Configure the individual devices

for i in 2 3 4 5 6 7 8 9 10 11
do
   PGDEV=/proc/net/pktgen/eth$i
   echo "Configuring $PGDEV"

   pgset "clone_skb 500000"
   pgset "min_pkt_size 60"
   pgset "max_pkt_size 60"
   pgset "dst 192.168.3.10$i"
   pgset "dst_mac 01:02:03:04:05:0$i"
   pgset "count 0"
done
echo "Running... CTRL-C to stop"
PGDEV=/proc/net/pktgen/pgctrl
pgset "start"

tail -2 /proc/net/pktgen/eth*
```

9 Tips for driver/chip testing

When testing a particular driver/chip/platform, start with TX. Use pktgen on the host system to get a sense of which ptkgen parameters are optimal and how well a particular NIC can perform TX. Try with a range of packet sizes from 64 bytes to 1500 bytes or jumbo frames.

Then start looking at the RX on the target platform by using pktgen to inject packets either direct via crossover cable or via pktgen from another host.

Again, vary the packet size etc To isolate driver/chip from other parts of kernel stack pktgen packets can be counted and dropped at various points. See section on detecting pktgen packets.

Depending on the purpose of the test repeat the process with additional devices, one at a time. Multiple devices are trickier since one needs to know I/O bus topology. Typically one tries to balance I/O loads by installing the NICs in the "right" slots or utilizing built-in devices appropriately.

9.1 Multiple Devices

With multiple devices, it is best to use CTRL-C to stop a test run. This prevents any pktgen thread from stopping before others and skewing the test results. Sometimes, one NIC will TX packets faster than another NIC just because of bias in the DMA latency or PCI bus arbiter (to name only two of several possibilities). Using CTRL-C to stop a test run aborts all pktgen threads at once. This results in a clean snapshot of how many packets a given configuration could generate over the same period of time. After the CTRL-C is received, pktgen will print the statistics the same as if the test had been stopped by a counter going to zero.

9.2 Other testing aspects

To isolate driver/chip from other parts of kernel stack, pktgen packets can be counted and dropped at various points. See Section 9.3 on detecting pktgen packets.

If the tested system only has one interface, the dummy interface can be setup as the output device. The advantage is we can test the system at very high load and the results are very reproduceable. Of course, other variables such as different types of offload and checksumming should be tested as well.

Besides knowing the hardware topology, one should know what workloads are expected to be present on the target system when placed in production (i.e. real world use). An FTP server can see quite a different workload than a web

server, mail handler, or router, etc. Roughly 160 Kpps seems to fill a Gigabit link when running an FTP server. While this can vary, it gives an useful estimate of required packet per second (pps) versus bandwidth for this type of production system.

For routers the number of routes in the routing table is also an issue as lookup times and other behaviour may be affected. The author has taken snapshots from current Internet routing table IPV4 and IPV6 (BGP) and formed into scripts for this purpose. The routes are added via the ip utility so the tested system does not need any routing connectivity nor routing daemon. Some scripts are available from:

`ftp://robur.slu.se/pub/Linux/net-development/inet_routes/`

At last use your fantasy when testing, elaborate with new setups try to understand how things are functioning, monitor interested and related variables add printouts etc. Testing understanding and development are closely related.

9.3 Detecting pktgen packets in kernel

Sometimes it's very useful to monitor/drop pktgen packets within the driver/network stack either at ingress or egress. The technique for both is essentially the same. The patchlet in Section 13.1 drops pktgen packets at ingress and uses an unused counter.

Also it should be possible to capture pktgen packets via the tc command and the u32 classifier which might be a better solution in most cases.

10 Thanks to...

Thanks to Grant Grundler, Jamal Hadi Salim, Jens Låås, and Hans Wassen for comments and useful insights. This paper covers several years of work and conversations with all of the above.

Relevant site:
`ftp://robur.slu.se://pub/Linux/net-development/pktgen-testing/`

Good luck with the linux net-development!

11 Appendix A

Table 3: Command Summary

Commands	
Pgcontrol commands	
start	Starts sending on all threads
stop	
Threads commands	
add_device	Add a device to thread i.e eth0
rem_device_all	Removes all devices from this thread
max_before_softirq	do_softirq() after sending a number of packets
Device commands	
debug	
clone_skb	Number of identical copies of the same packet 0 means alloc for each skb. For DoS etc we must alloc new skb's.
clear_counters	normally handled automatically
pkt_size	Link packet size minus CRC (4)
min_pkt_size	Range pkt_size setting If < max_pkt_size, then cycle through the port range.
max_pkt_size	
frags	Number of fragments for a packet
count	Number of packets to send. Use zero for continious sending
delay	Artificial gap inserted between packets in nanoseconds
dst	IP destination address i.e 10.0.0.1
dst_min	Same as dst If < dst_max, then cycle through the port range.
dst_max	Maximum destination IP. i.e 10.0.0..1
src_min	Minimum (or only) source IP. i.e. 10.0.0.254 If < src_max, then cycle through the port range.
src_max	Maximum source IP.
dst6	IPV6 destination address i.e fec0::1
src6	IPV6 source address i.e fec0::2
dstmac	MAC destination adress 00:00:00:00:00:00
srcmac	MAC source adress. If omitted it's automatically taken from source device
src_mac_count	Number of MACs we'll range through. Minimum' MAC is what you set with srcmac.
dst_mac_count	Number of MACs we'll range through. Minimum' MAC is what you set with dstmac.
Flags	
IPSRC_RND IPDST_RND TXSIZE_RND UDPSRC_RND	IP Source is random (between min/max), Etc

Commands continued	
UDPDST_RND MACSRC_RND MACDST_RND	
udp_src_min	UDP source port min, If < udp_src_max, then cycle through the port range.
udp_src_max	UDP source port max.
udp_dst_min	UDP destination port min, If < udp_dst_max, then cycle through the port range.
udp_dst_max	UDP destination port max.
stop	Aborts packet injection. Ctrl-C also aborts generator. **Note**: Use count 0 (forever) and stop the run with Ctrl-C when multiple devices are assigned to one pktgen thread. This avoids some devices finishing before others and skewing the results. We are primarily interested in how many packets all devices can send at the same time, not absolute number of packets each NIC sent.
flows	Number of concurrent flows
flowlen	Length of a flow

12 Appendix B

12.1 Sample pktgen output

/proc/net/pktgen/eth1 output after run

```
Params: count 10000000  min_pkt_size: 60  max_pkt_size: 60
 frags: 0  delay: 0  clone_skb: 1000000  ifname: eth1
 flows: 0 flowlen: 0
 dst_min: 10.10.11.2  dst_max:
 src_min:   src_max:
 src_mac: 00:00:00:00:00:00  dst_mac: 00:07:E9:13:5C:3E
 udp_src_min: 9  udp_src_max: 9  udp_dst_min: 9  udp_dst_max: 9
 src_mac_count: 0  dst_mac_count: 0
 Flags:
Current:
 pkts-sofar: 10000000  errors: 39192
 started: 1076616572728240us  stopped: 1076616585502839us idle: 1037781us
 seq_num: 11  cur_dst_mac_offset: 0  cur_src_mac_offset: 0
 cur_saddr: 0x10a0a0a  cur_daddr: 0x20b0a0a
 cur_udp_dst: 9  cur_udp_src: 9
 flows: 0
Result: OK: 12774599(c11736818+d1037781) usec, 10000000 (64byte)
 782840pps 382Mb/sec (400814080bps)  errors: 39192


Results show 10 millon 64 byte packets were sent on eth1 to 10.10.11.2
with a rate at 783 kpps
```

```
\section{Appendix C}
\subsection{pktgen.conf-1-1 script}

Below is the full pktgen.conf-1-1 script

\begin{footnotesize}
\begin{verbatim}
#!/bin/sh

#modprobe pktgen

function pgset() {
  local result

  echo $1 > $PGDEV

  result=‘cat $PGDEV | fgrep "Result: OK:"‘
  if [ "$result" = "" ]; then
       cat $PGDEV | fgrep Result:
  fi
}

function pg() {
    echo inject > $PGDEV
    cat $PGDEV
}

# Config Start Here -----------------------------------

# thread config
# Each CPU has own thread. Two CPU exammple.
# We add eth1, eth2 respectively.

PGDEV=/proc/net/pktgen/kpktgend_0
  echo "Removing all devices"
 pgset "rem_device_all"
  echo "Adding eth1"
 pgset "add_device eth1"
  echo "Setting max_before_softirq 10000"
 pgset "max_before_softirq 10000"

# device config
# delay is inter packet gap. 0 means maximum speed.

CLONE_SKB="clone_skb 1000000"
# NIC adds 4 bytes CRC
PKT_SIZE="pkt_size 60"

# COUNT 0 means forever
#COUNT="count 0"
COUNT="count 10000000"
delay="delay 0"
```

```
PGDEV=/proc/net/pktgen/eth1
  echo "Configuring $PGDEV"
 pgset "$COUNT"
 pgset "$CLONE_SKB"
 pgset "$PKT_SIZE"
 pgset "$delay"
 pgset "dst 10.10.11.2"
 pgset "dst_mac  00:04:23:08:91:dc"

# Time to run
PGDEV=/proc/net/pktgen/pgctrl

 echo "Running... ctrl^C to stop"
 pgset "start"
 echo "Done"

# Result can be vieved in /proc/net/pktgen/eth1
```

13 Appendix D

13.1 Patchlet to ip_input.c

Below is the patchlet to count and drop pktgen packets.

```
--- linux/net/ipv4/ip_input.c.orig      Mon Feb 10 19:37:57 2003
+++ linux/net/ipv4/ip_input.c   Fri Feb 21 21:42:45 2003
@@ -372,6 +372,23 @@
             IP_INC_STATS_BH(IpInDiscards);
             goto out;
     }

+           {
+                 __u8 *data = (__u8 *) skb->data+20;
+
+                 /* src and dst port 9 --> pktgen */
+
+                 if(data[0] == 0 &&
+                    data[1] == 9 &&
+                    data[2] == 0 &&
+                    data[3] == 9) {
+                        netdev_rx_stat[smp_processor_id()].fastroute_hit+
+;
+                         goto drop;
+                 }
+           }
+
```

```
if (!pskb_may_pull(skb, sizeof(struct iphdr)))
        goto inhdr_error;
```

TWIN: A Window System for 'Sub-PDA' Devices

Keith Packard
HP Cambridge Research Laboratory
keithp@keithp.com

Abstract

With embedded systems gaining high resolution displays and powerful CPUs, the desire for sophisticated graphical user interfaces can be realized in even the smallest of systems. While the CPU power available for a given power budget has increased dramatically, these tiny systems remain severely memory constrained. This unique environment presents interesting challenges in graphical system design and implementation. To explore this particular space, a new window system, TWIN, has been developed. Using ideas from modern window systems in larger environments, TWIN offers overlapping translucent windows, anti-aliased graphics and scalable fonts in a total memory budget of 100KB.

Motivation

Researchers at the HP Cambridge Research Laboratory are building a collection of sub-PDA sized general purpose networked computers as platforms for dissociated, distributed computing research. These devices include small LCD or OLED screens, a few buttons and occasionally some kind of pointing device.

One of the hardware platforms under development consists of a TMS320 series DSP (200MHz, fixed point, 384KB on-chip RAM), 8MB of flash memory, an Agilent ADNS-2030 Optical mouse sensor, a Zigbee (802.15.4) wireless networking interface and an Epson L2F50176T00 LCD screen (1.1", 120 x 160 color). At 200MHz, this processor is capable of significant computation, but 384KB holds little data.

In contrast, early graphical user interfaces for desktop platforms was more constrained by available CPU performance than by memory. Early workstations had at least a million pixels and a megabyte of physical memory but only about 1 MIPS of processing power. Software in this environment was much more a matter of what could be made fast enough than what would fit in memory.

While the X window system[7] has been ported to reasonably small environments[2], a minimal combination of window system server, protocol library and application toolkit consumes on the order of 4 to 5MB of memory, some ten times more than is available in the target platform.

Given the new challenge of providing a graphical user interface in these tiny devices, it seemed reasonable to revisit the whole graphical architecture and construct a new system from the ground up. The TWIN window system (for Tiny WINdow system) is the result of this research.

Assumptions

The hardware described above can be generalized to provide a framework within which the TWIN architecture fits. By focusing on specific hardware capabilities and limitations, the window system will more completely utilize those limited resources. Of course, over-constraining the requirements can limit the potential target environments. Given the very general nature of existing window systems, it seems interesting to explore what happens when less variation is permitted.

The first assumption made was that little-to-no graphics acceleration is available within the frame buffer, and that the frame buffer is attached to the CPU through a relatively slow link. This combination means that most drawing should be done with the CPU in local memory, and not directly to the frame buffer. This has an additional benefit in encouraging synchronized screen updates where intermediate rendering results are never made visible to the user. If the CPU has sufficient on-chip storage, this design can also reduce power consumption by reducing off-chip data references.

The second limitation imposed was to require a color screen with fixed color mapping. While this may appear purely beneficial to the user, the software advantages are numerous as well. Imprecise rendering operations can now generate small nearly invisible errors instead of visibly incorrect results through the use of anti-aliased drawing. With smooth gradations of color available, there is no requirement that the system support dithering or other color-approximating schemes.

Finally, TWIN assumes that the target machine provides respectable CPU performance. This reduces the need to cache intermediate rendering results, like glyph images for text. Having a homogeneously performant target market means that TWIN need support only one general performance class of drawing operations. For example, TWIN supports only anti-aliased drawing; non-antialiased drawing would be faster, but the CPUs supported by twin are required to be fast enough to make this irrelevant.

The combined effect of these environmental limitations means that TWIN can provide significant functionality with little wasted code. Window systems designed for a range of target platforms must often generalize functionality and expose applications to variability which will not, in practice, ever been experienced by them. For example, X provides six different color models for monochrome, colormapped and static color displays. In practice, only TrueColor (separate monotonic red, green, blue elements in each pixel) will ever be used by the majority of X users. Eliminating choice has benefits beyond the mere reduction of window system code, it reflects throughout the application stack.

Windowing

Windowing can be thought of as the process of simulating multiple, separate, two-dimensional surfaces sharing the same display. These virtual surfaces, or 'windows,' are then combined into a single presentation. Traditional window systems do this by presenting a '2 ½' dimensional user interface which assigns different constant Z values to each object so that the windows appear to be stacked on top of one another.

TWIN provides this traditional metaphor through an architecture similar to the X window system Composite extension in that all applications draw to off-screen image buffers which are then combined and placed in the physical frame buffer. This has many advantages:

• Rendering performance is decoupled from frame buffer performance. As the embedded frame buffer controllers include a private frame buffer, the bandwidth available to the CPU for that memory is quite restricted. Decoupling these two operations means that rendering can operate at full main memory speed instead of the reduced video controller memory speed
• Rendering operations needn't clip to overlapping windows. Eliminating the need to perform clipping reduces the complexity and size of the window system by eliminating the code needed to construct and maintain the clip list data structures.
• Applications need not deal with damage events. In a traditional clipping-based window system, applications must be able to reconstruct their presentation data quickly to provide data for newly visible portions of windows.
• Multiple window image formats can be supported, including those with translucency information. By constructing the physical frame buffer data from the combination of various window contents, it is possible to perform arbitrary image manipulation operations on those window contents, including translucency effects.

In the model supported in the X window system by the Composite extension, an external application is responsible for directing the system in constructing the final screen image from the off-screen window contents. TWIN has a simpler model where window contents are composited together through a fixed mechanism. This, of course, eliminates significant complexity but at the cost of also eliminating significant generality. TWIN does not, and is not likely to, support immersive 3D environments.

TWIN tracks rectangular regions of modified pixels within each window. When updating the screen, a single scanline of intermediate storage is used to compute new screen contents. The list of displayed windows is traversed and any section* of the window overlapping the scanline is painted into the intermediate scanline. When complete, the scanline is sent to the frame buffer. This single scanline provides the benefits of a double buffered display without the need for a duplicate frame buffer.

Graphics

The availability of small color screens using either LCD or OLED technologies combined with sufficient CPUpower have encouraged the inclusion of a rendering model designed to take maximal advantage of the limited pixel resolution available. Anti-aliasing and sub-pixel addressing is used to produce higher fidelity renderings within the limited screen resolution. Per-pixel translucency is included to 'see through' objects as well as permit arbitrary object shapes to minimize unused space on the screen.

The complete drawing stack provides a simacrulum of the PDF 1.4 drawing environment, complete with affine transforms, color image blending and PostScript path construction and drawing tools. Leveraging this classic and well known environment ensures both that developers will feel comfortable with the tools and that the system is 'complete' in some informal sense.

Pixel Manipulation

TWIN uses the rendering operational model from 8 ½[5], the window system developed for the Plan 9 operating system by Cox and Pike, the same as used in the X render extension[4]. This three-operand rendering operator forms the base upon which all drawing is built:

dst = (src IN mask) OVER|SOURCE dst

The IN, OVER and SOURCE operators are as defined by Porter and Duff.[6] By manipulating the operands, this single operator performs all of the rendering facilities in the TWIN system. Geometric operations are performed by constructing a suitable mask operand based on the shape of the geometry.

Pixel data are limited in TWIN to three formats, 8 bit alpha, 32 bit ARGB and 16 bit RGB. Limiting formats in this way along with the limited number of operators in the rendering equation provided an opportunity to instantiate each combination in custom compositing code. With three formats for each operand and two operators, there are 54 different rendering functions in 13KB of code.

Geometric Objects

For geometric operations, TWIN uses the model from PostScript as implemented in the cairo graphics system.[8] 'Paths' are constructed from a sequence of lines and Bézier splines. An arbitrary path can be convolved with a convex path to construct a new path representing the original path as stroked by the convex path. The convolution operation approximates the outline of the Minkowski sum of the two paths.

A path can then be drawn by scan converting it to a mask for use in the rendering operation described above. Because the rendering operation can handle translucency, this scan conversion operation does anti-aliasing by sampling the path in a 4×4 grid over each pixel to compute approximate coverage data. This sampling grid can be easily adjusted to trade quality for performance.

The application interface includes an affine transformation from an arbitrary 16.16 fixed point coordinate space to 12.4 fixed point pixel space. The 16.16 fixed point values provide reasonable dynamic range for hardware which does not include floating point acceleration. The 12.4 fixed point pixel coordinates provide sufficient resolution to accurately reproduce object geometry on the screen. Note that the screen is therefore implicitly limited to 4096 pixels square.

Glyph Representation

Providing text at multiple sizes allows the user interface to take maximal advantage of the limited screen size. This can either be done by storing pre-computed glyphs at multiple sizes or preparing glyphs at run-time from scalable data. Commercial scalable font formats all represent glyphs in outline form. The resulting glyph is constructed by filling a complex shape constructed from lines and splines. The outline data for one face for the ASCII character set could be compressed to less than 7KB – significantly smaller than the storage needed for a bitmap face at a single size.

However, a straightforward rasterization of an outline does not provide an ideal presentation on the screen. Outline fonts often include hinting information to adjust glyph shapes at small pixel sizes to improve sharpness and readability. This hinting information requires significantly more code and data than the outlines themselves, making it impractical for the target device class.

An alternative representation for glyphs is as stroke data. With only the path of the pen recorded, the amount of data necessary to represent each glyph is reduced. More significantly, with the stroke width information iso-

lated from the stroke path, it is possible to automatically adjust the stroke positions to improve the presentation on the screen. A secondary adjustment of the pen shape completes the hinting process. The results compare favorably with fully hinted outline text.

An additional feature of the stroke representation is that producing oblique and bold variants of the face are straightforward; slanting the text without changing the pen shape provides a convincing oblique while increasing the pen width produces a usable bold.

The glyphs themselves have a venerable history. The shapes come from work done by Dr A.V. Hershey for the US National Bureau of Standards. Those glyphs were designed for period pen plotters and were constructed from straight line segments on a relatively low resolution grid. The complete set of glyphs contains many different letterforms from simple gothic shapes to letters constructed from multiple parallel strokes that provide an illusion of varying stroke widths. Many additional decorative glyphs were also designed.

From this set of shapes, a simple gothic set of letters, numbers and punctuation was chosen. Additional glyphs were designed to provide a complete ASCII set. The curves within the Hershey glyphs, designed as sequences of short line segments, were replaced by cubic splines. This served both to improve the appearance of the glyphs under a variety of transforms as well as to reduce the storage required for the glyphs as a single cubic spline can replace many line segments. Figure 1 shows a glyph as originally designed with 33 line segments and the same glyph described as seven Bézier splines. Storage for this glyph was reduced from 99 to 52 bytes.

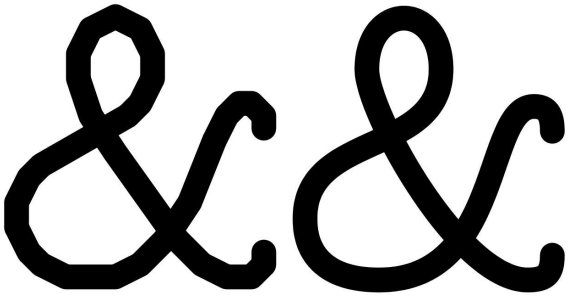

Figure 1: Converting Lines To Splines

Glyph Hinting

Given the desire to present text at a variety of sizes, the glyph shapes need to undergo a scaling transformation and then be rasterized to create an image. Unless this scaling is restricted to integer values, the edges of the resulting strokes will not necessarily align with the pixel grid. The resulting glyphs will appear fuzzy and will be hard to read.

To improve the appearance of the glyphs on the screen, a straightforward mechanism was developed to reposition the glyph control points to improve the rasterized result. The glyph data was augmented to include a list of X and a list of Y coordinates. Each ‘snap’ list contains values along the respective axis where some point within the glyph is designed to lie on a pixel boundary. These were constructed automatically by identifying all vertical and horizontal segments of each glyph, including splines whose ends are tangent to the vertical or horizontal.

The glyph coordinates are then scaled to the desired size. The two snap lists (X and Y) are used to push glyph coordinates to the nearest pixel grid line. Coordinates between points on a snap list are moved so that the relative distance from the nearest snapped coordinates remain the same. The pen width is snapped to the nearest integer size. If the snapped pen width is odd, the entire glyph is pushed ½ a pixel

in both directions to align the pen edges with the pixel edges. Figure 2 shows a glyph being hinted in this fashion.

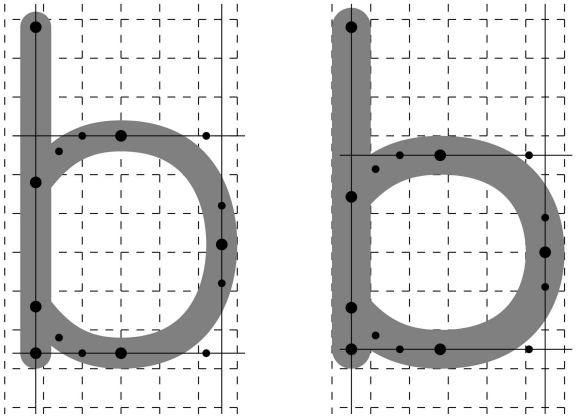

Figure 2: Hinting A Glyph

The effect is to stretch or shrink the glyph to align vertical and horizontal strokes to the pixel grid. Glyphs designed with evenly spaced vertical or horizontal stems (like 'm') may end up unevenly spaced; a more sophisticated hinting systems could take this into account by preserving the relative spacing among multiple strokes.

User Interface Objects

With the window system supporting a single screen containing many windows, the toolkit extends this model by creating a single top-level widget. This top-level widget contains a single box for layout purposes. Each box can contain a number of widgets or other boxes.

Layout within each box is done either horizontally or vertically with an algorithm which comes from the Layout Widget[3] that the author developed for Xt[1] library. Each widget has a natural size and stretch in both directions. The natural size and stretch of a box is computed from the objects it contains. This forms the sole geometry management mechanism within the toolkit and is reasonably competent at both constructing a usable initial layout and adapting to externally imposed size changes.

Process & Thread Model

TWIN was initially developed to run on a custom embedded operating system. This operating system design initially included simple cooperative threading support, and TWIN was designed to run different parts of the window system in different threads:

- Input would run in one thread, events were dispatched without queuing directly to the receiving object.
- Each window would have a thread to redisplay the window contents. These threads would block on a semaphore awaiting a change in application state before reconstructing the window contents. Per window locks could block updates until the application state was consistent.
- The window system had a separate thread to compose the separate window contents into the final screen display. The global redisplay thread would block on a semaphore which the per-window redisplay threads would signal when any window content changed. A global system lock could block updates while any application state was inconsistent.

This architecture was difficult to manage as it required per-task locking between input and output. The lack of actual multi-tasking of the application processing eliminated much of the value of threads.

Once this was working, support for threading was removed from the custom operating system.

With no thread support at all, TWIN was redesigned with a global event loop monitoring input, timers and work queues. The combination of these three mechanisms replaced the collection of threads described above fairly easily, and the complexities of locking between input and output within a single logical task were removed.

Of course, once this was all working, the custom operating system was replaced with ucLinux.

While the single thread model works fine in ucLinux, it would be nice to split separate out tasks into processes. Right now, all of the tasks are linked into a monolithic executable. This modularization work is underway.

Input Model

A window system is responsible for collecting raw input data from the user in the form of button, pointer and key manipulation and distributing them to the appropriate applications.

TWIN takes a simplistic approach to this process, providing a single immutable model. Pointer events are delivered to the window containing the pointing device. Transparent areas of each window are excluded from this containment, so arbitrary shapes can be used to select for input.

TWIN assumes that any pointing device will have at least one associated signal – a mouse button, a screen touch or perhaps something else. When pressed, the pointing device is 'grabbed' by the window containing the pointer at that point. Motion information is delivered only to that window until the button is released.

Device events not associated with a pointer, such as keyboards, are routed to a fixed 'active' window. The active window is set under application control, such as when a mouse button press occurs within an inactive window. The active window need need not be the top-most window.

Under both the original multi-threaded model and the current single-threaded model, there is no event queueing within the window system; events are dispatched directly upon being received from a device. This is certainly easy to manage and allows motion events to be easily discarded when the system is too busy to process them. However, with the switch to multiple independent processes running on ucLinux, it may become necessary to queue events between the input collection agent and the application processing them.

Within the toolkit, events are dispatched through each level of the hierarchy. Within each box, keyboard events are statically routed to the active box or widget while mouse events are routed to the containing box or widget. By explicitly dispatching down each level, the containing widgets and boxes can enforce whatever policy they like for event delivery, including mouse or keyboard grabs, focus traversal and event replay.

While this mechanism is fully implemented, much investigation remains to be done to explore what kinds of operations are useful and whether portions of what is now application-defined behavior should be migrated into common code.

Window Management

TWIN embeds window management right into the toolkit. Support for resize, move and minimization is not under the control of an external application. Instead, the toolkit automatically constructs suitable decorations for each

window as regular toolkit objects and the normal event dispatch mechanism directs window management activities.

While external management is a valuable architectural feature in a heterogeneous desktop environment, the additional space, time and complexity rules this out in today's Sub-PDA world.

Status and Future Work

As computing systems continue to press into ever smaller environments, the ability to bring sophisticated user interface technologies along greatly increases both the value of such products as well as the scope of the potential market.

The TWIN window compositing mechanism, graphics model and event delivery system have been implemented using a mock-up of the hardware running on Linux using the X window system. Figure 3 shows most of the current capabilities in the system.

While the structure of the TWIN window system is complete, the toolkit is far from complete, having only a few rudimentary widgets. And, as mentioned above, the port to ucLinux is not yet taking advantage of the multiple process support in that environment. These changes will likely be accompanied by others as TWIN is finally running on the target hardware.

In the x86 emulation environment, the window system along with a small cadre of demonstration applications now fits in about 50KB of text space with memory above that limited largely to the storage of the off-screen window contents. Performance on a 1.2GHz laptop processor is more than adequate; it will be rather interesting to see how these algorithms scale down to the target CPU.

The current source code is available from via CVS, follow the link from http://keithp.com. The code is licensed with an MIT-style license, permitting liberal commercial use.

References

[1] Paul J. Asente and Ralph R. Swick. *X Window System Toolkit*. Digital Press, 1990.

[2] William R. Hamburgen, Deborah A. Wallach, Marc A. Viredaz, Lawrence S. Brakmo, Carl A. Waldspurger, Joel F. Bartlett, Timothy Mann, and Keith I. Farkas. Itsy: Stretching the Bounds of Mobile Computing. *IEEE Computer*, 34(4):28–35, April 2001.

[3] Keith Packard. The LayoutWidget: A TeX Style Constraint Widget Class. *The X Resource*, 5, Winter 1993.

[4] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.

[5] Rob Pike. *draw - screen graphics*. Bell Laboratories, 2000. Plan 9 Manual Page Entry.

[6] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[7] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

[8] Carl Worth and Keith Packard. Xr: Cross-device Rendering for Vector Graphics. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, ON, July 2003. OLS.

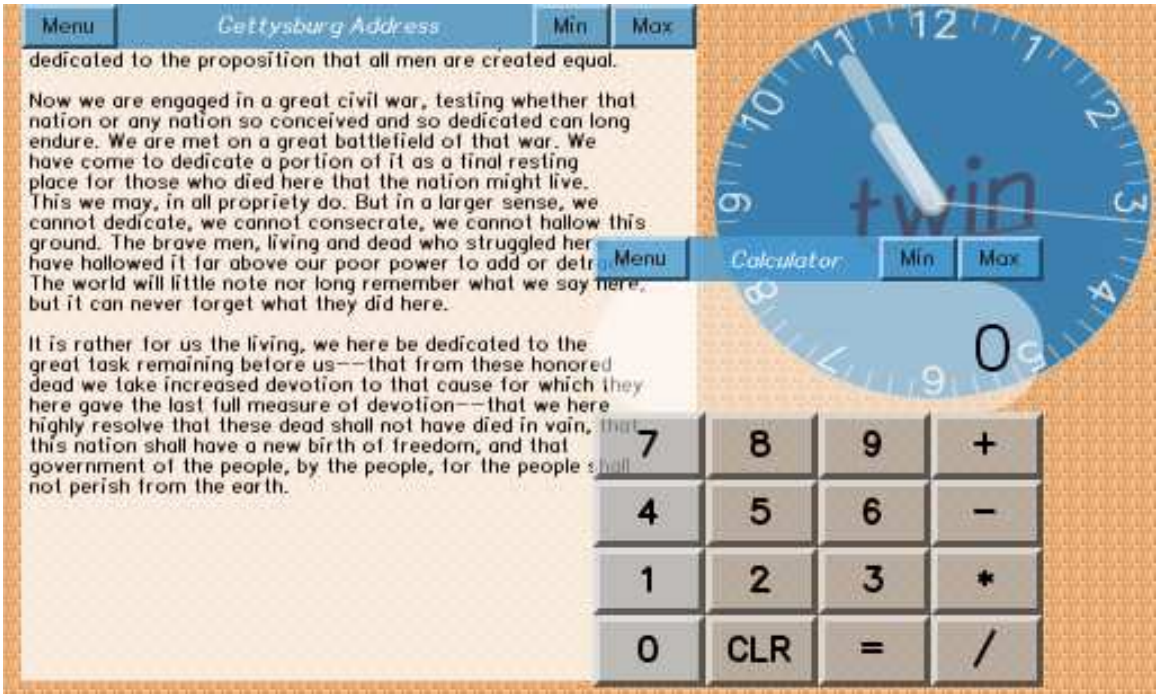

Figure 3: Sample Screen Image

RapidIO for Linux

Matt Porter
MontaVista Software, Inc.
mporter@mvista.com
mporter@kernel.crashing.org

Abstract

RapidIO is a switched fabric interconnect standard intended for embedded systems. Providing a message based interface, it is currently capable of speeds up to 10Gb/s full duplex and is available in many form factors including ATCA for telecom applications. In this paper, the author introduces a RapidIO subsystem for the Linux kernel. The implementation provides support for discovery and enumeration of devices, management of resources, and a consistent access mechanism for drivers and other kernel facilities. As an example of the use of the subsystem feature set, the author presents a Linux network driver implementation which communicates via RapidIO message packets.

1 Introduction to RapidIO

1.1 Busses and Switched Fabrics

To date, most well known system interconnect technologies have been shared memory bus designs. ISA, PCI, and VMEbus are all examples of shared memory bus systems. A shared memory bus interconnect will have a specific bus width which is measured by the number of data lines routed to each participant on the bus. An arbitration mechanism is required to determine which participant owns the bus for purposes of asserting an address-data cycle onto the bus. Other participants will decode the address-data cycle and latch data locally if the address cycle is intended for them. In the shared memory bus architecture, there is one global address space shared amongst all participants.

Switched fabric interconnect technology has been around for some time with proprietary implementations like StarFabric. In recent years though, standardized switched fabrics like HyperTransport, Infiniband, PCI Express, and RapidIO have become more familiar names. A switched fabric interconnect is usually modeled much like a switched network architecture. However, it provides features that a chip to chip or intra-chassis conventional shared memory bus standard would provide. Each node has at least one link that can be connected point to point or into a switch element. An implementation-specific routing method determines packet routing in the network. Typically, a switched fabric interconnect incorporates some method of sending messages and events through the network. In some cases, the switched fabric will implement memory mapped I/O over the network.

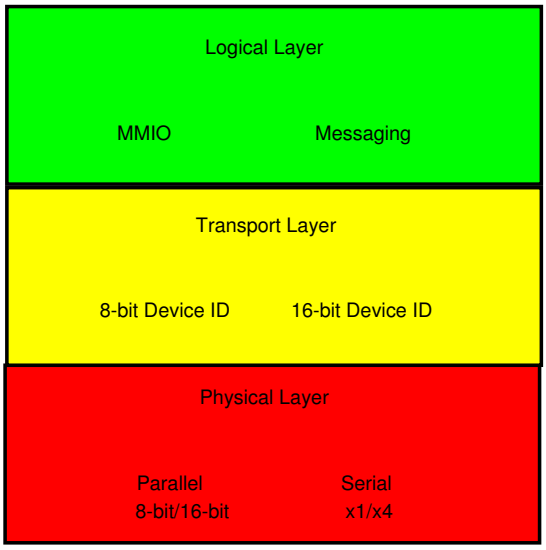

Figure 1: RapidIO Layers

1.2 RapidIO Overview

The RapidIO interconnect technology was originally created by Motorola for use in embedded computing systems. Motorola (now Freescale Semiconductor) later created the RapidIO Trade Association (RTA) to guide future development of the specification. A number of embedded silicon vendors are active members of the RTA and are now shipping or announcing RapidIO devices.

The RapidIO specification is divided into three distinct layers. These layers are illustrated in Figure 1.

1. Logical Layer
   Provides methods for memory mapped I/O (MMIO) and message-based access to devices. MMIO allows for accesses within a local memory space to generate read/write transactions within the address space of a remote device. Each device has a unique RapidIO address space that can range from 34-bits to 66-bits in size. RapidIO provides a messaging model with mailbox and doorbell facilities. A mailbox is a hardware port which can send and receive messages up to 4KB in size. A doorbell is a specialized message port which can be used for event notifications similar to message signaled interrupts.

2. Transport Layer
   Implements the device ID routing methodology. In RapidIO, packets are routed by a unique device ID. Two different sizes of device IDs are defined , either a small (8-bit) or large (16-bit) device ID. The small device ID allows a maximum of 256 devices whereas the large device ID allows a maximum of 65536 devices.

3. Physical Layer
   Offers either parallel or serial implementations for the physical interconnect. The parallel version is available in 8-bit or 16-bit configurations with full duplex speeds up to 8 Gb/s and 16 Gb/s, respectively. The serial implementation offers lane configurations of x1 or x4. In the x1 configuration, the single lane offers up to 3.125 Gb/s full duplex data throughput. In the x4 configuration, each lane offers up to 2.5Gb/s full duplex data throughput resulting in 10 Gb/s total bandwidth.

1.3 RapidIO versus PCI Express

RapidIO is often compared to PCI Express because of the popularity of PCI Express in the commodity PC workstation/server market. On the surface, both seem very similar, offering features that improve upon the use of conventional PCI as a system interconnect. RapidIO, however, is designed with some features targeted at specific embedded system needs that will likely facilitate its inclusion in many applications. There are now embedded processors available which include both PCI Express and RapidIO support on the chip.

PCI Express and RapidIO have similar data rate capabilities. PCI Express offers lane configurations of x1 through x32 where each lane offers 2 Gb/s full duplex data throughput. In a typical PC implementation there is a single x16 slot for graphics that handles 32 Gb/s full duplex and multiple x1 slots which can handle 4 Gb/s full duplex.

Both interconnects also have similar discovery models. A separate set of transactions is used to access configuration space registers. Configuration space accesses are used to determine existence of nodes in the system and additional information about those nodes.

A major difference between PCI Express and RapidIO is in system topology capability. PCI Express is backward compatible with PCI and therefore depends on the host and multiple slave device model. RapidIO is designed for multiple hosts in the system performing redundant discovery. In addition, it can be configured in any network topology allowing direct node to node communication.

Device addressing is very different as well. In PCI Express, the globally shared address space with hierarchical windows of address space is retained from PCI. This is important for backward compatibility of software and allows routing of packets via base address assignments. RapidIO's device ID based routing simplifies changes to the network due to device failure or hot plug events.

PCI Express does not offer a standardized messaging facility. Most modern distributed applications are based on message passing architectures.

2 RapidIO Hardware

The current generation RapidIO parts use an 8-bit wide parallel physical layer. These parts can support up to 8Gb/s full duplex data throughput. The first RapidIO processor elements (endpoints with a processor) are Freescale's MPC8540 and MPC8560 Systems-on-a-Chip (SoC). The first RapidIO switch is the Tundra Tsi500.

The first commercially available system with these parts is the STx GP3 HIPPS2 development platform. This system includes one or more STx GP3 boards containing the MPC8560 processor and a HIPPS2 RapidIO backplane with two Tsi500 switches. The Linux RapidIO subsystem is being developed using this platform with two STx GP3 boards plugged into the HIPPS2 backplane.

3 Linux RapidIO Subsystem

3.1 Subsystem Overview

Due to the discovery mechanism similarities between PCI and RapidIO, the RapidIO subsystem has a structure which is similar to that of the PCI subsystem. The subsystem hooks into the standard Linux Device Model (LDM) in a similar fashion to other busses in the kernel. RapidIO specific device and bus types are defined and registered with the LDM. The core subsystem is designed such that there is a clear separation between the generic subsystem interfaces and architecture specific interfaces which support RapidIO. Finally, a set of subsystem device driver interfaces is defined to abstract access to facilities by device drivers.

3.2 Subsystem Core

The core of the Linux RapidIO subsystem revolves around four major components.

1. Master Port. A master port is an interface which allows RapidIO transactions to be transmitted and received in a system. A master port provides a bridge from a processor running Linux into the switched fabric network.

2. Device. A RapidIO device is any endpoint or switch on the network.

3. Switch. A RapidIO switch is a special class of device which routes packets between point to point connections to reach their final destination.

4. Network. A RapidIO network comprises a set of endpoints and switches that are interconnected.

Each of these components is mapped into a subsystem structure. The RapidIO subsystem uses these structures as the root handle for manipulating the hardware components abstracted by the structures.

`struct rio_mport` (Figure 2) contains information regarding a specific master port. Master port specific resources such as inbound mailboxes and doorbells are contained in this structure. If a master port is defined as a enumerating host, then the structure will contain a unique host device ID. The host device ID is used for multi-host locking purposes during enumeration.

`struct rio_switch` (Figure 3) contains information about a RapidIO switch device. The structure is populated during enumeration and discovery of the system with information such as the number of hops to the switch and the routing table present in the switch. In addition, pointers to switch specific routing table operations reside here.

`struct rio_dev` (Figure 4) contains information about an endpoint or switch that is part of the RapidIO system. Fields are present to cache many common configuration space registers.

`struct rio_net` (Figure 5) contains information about a specific RapidIO network known to the system. It defines a list of all devices that are part of the network. Another list tracks all of the local processor master ports that can access this network. The `hport` field points to the default master port which is used to communicate with devices within the network.

3.3 Subsystem Initialization

In order to initialize the RapidIO subsystem, an architecture must register at least one master port to send and receive transactions within the RapidIO network. A `subsys_initcall()` is registered which is responsible for any arch-specific RapidIO initialization. This includes hardware initialization and registration of active master ports in the system. The final step of the initcall is to execute `rio_init_mports()` which performs enumeration and discovery on all registered master ports.

3.4 Enumeration and Discovery

The enumeration and discovery process is implemented to comply with the multiple host enumeration algorithm detailed in the *RapidIO Interconnect Specification: Annex I* [1]. Enumeration is performed by a master port which is designated as a host port. A host port is defined as a master port which has a host device ID greater than or equal to zero. A host device ID is assigned to a master port in a platform specific manner or can be passed on the command line.

```
struct rio_mport {
    struct list_head dbells;         /* list of doorbell events */
    struct list_head node;           /* node in global list of ports */
    struct list_head nnode;          /* node in net list of ports */
    struct resource iores;
    struct resource riores[RIO_MAX_MPORT_RESOURCES];
    struct rio_msg inb_msg[RIO_MAX_MBOX];
    struct rio_msg outb_msg[RIO_MAX_MBOX];
    int host_deviceid;               /* Host device ID */
    struct rio_ops *ops;             /* maintenance transaction functions */
    unsigned char id;                /* port ID, unique among all ports */
    unsigned char index;             /* port index, unique among all port
                                                interfaces of the same type */
    unsigned char    name[40];
};
```

Figure 2: struct rio_mport

```
struct rio_switch {
    struct list_head node;
    u16 switchid;
    u16 hopcount;
    u16 destid;
    u16 route_table[RIO_MAX_ROUTE_ENTRIES];
    int (*add_entry)(struct rio_mport *mport, u16 destid, u8 hopcount,
            u16 table, u16 route_destid, u8 route_port);
    int (*get_entry)(struct rio_mport *mport, u16 destid, u8 hopcount,
            u16 table, u16 route_destid, u8 *route_port);
};
```

Figure 3: struct rio_switch

```
struct rio_dev {
    struct list_head global_list;    /* node in list of all RIO devices */
    struct list_head net_list;       /* node in per net list */
    struct rio_net *net;             /* RIO net this device resides in */
    u16 did;
    u16 vid;
    u32 device_rev;
    u16 asm_did;
    u16 asm_vid;
    u16 asm_rev;
    u16 efptr;
    u32 pef;
    u32 swpinfo;                     /* Only used for switches */
    u32 src_ops;
    u32 dst_ops;
    struct rio_switch *rswitch;      /* RIO switch info */
    struct rio_driver *driver;       /* RIO driver claiming this device */
    struct device dev;               /* LDM device structure */
    struct resource riores[RIO_MAX_DEV_RESOURCES];
    u16 destid;
};
```

Figure 4: struct rio_dev

```
struct rio_net {
    struct list_head node;           /* node in list of networks */
    struct list_head devices;        /* list of devices in this net */
    struct list_head mports;         /* list of ports accessing net */
    struct rio_mport *hport;         /* primary port for accessing net */
    unsigned char id;                /* RIO network ID */
};
```

Figure 5: struct rio_net

During enumeration, maintenance transactions are used to access the configuration space of devices. A maintenance transaction has two components to address a device, a device ID and a hopcount. The device ID is normally used for endpoint devices to determine if they should accept a packet. It is a requirement for all devices to ignore the device ID and accept any transaction during enumeration. Switches are a different case, however, as they do not implement a device ID. Transactions which reach a switch device must have their hopcount set appropriately. If a maintenance transaction with a hopcount of 0 reaches a switch, then the switch will process the packet against its own configuration space. If a maintenance transaction has a hopcount greater than 0, then the switch decrements the hopcount in the packet and forwards it along according to the route set for the corresponding device ID in the packet.

The enumeration process walks the network depth first. Like PCI enumeration, this is easily implemented by recursion. When a device is found, the Host Device ID Lock Register is written to ensure that the enumerator has exclusive enumeration ownership of the device. The device's capabilities are then queried to determine if it is a switch or endpoint device.

If the device is an endpoint, it is allocated a new unique device ID and this value is written to the endpoint. A new `rio_dev` is allocated and initialized.

If the device is a switch, its vendor and device ID are queried against a table of known RapidIO switches. A switch table entry has a set of switch routing operations which are specific to the located switch. The routing operations are used to read and write route entries in the switch. New `rio_dev` and `rio_switch` structures are then allocated and initialized.

Enumeration past a switch device is accomplished by iterating over each active switch port on the switch. For each active link, a route to a fake device ID (0xFF for 8-bit systems and 0xFFFF for 16-bit systems) is written to the route table. The algorithm recurses by calling itself with hopcount + 1 and the fake device ID in order to access the device on the active port. While traversing the network, the current allocated device ID is tracked. When the depth first traversal completes, the recursion unwinds and permanent routes are written into the switch routing tables. The device IDs that were found beyond a switch port are assigned route entries pointing to the port which they were found behind.

When the host has completed enumeration of the entire network it calls `rio_clear_locks()` to clean up. For each device in the system, it writes a magic "enumeration complete" value to the Component Tag Register. This register is essentially a scratch pad register reserved for enumeration housekeeping. After this process, all Host Device ID Lock Registers are cleared. Remote nodes that are to initiate passive discovery of the network wait for the magic value to appear in the Component Tag Register and then begin discovery.

The discovery process is similar to the enumeration process that has already been described. However, the discovery process is performed passively. This means that all devices in the network are traversed without modifying device IDs or routing tables. This is necessary in the case where there are multiple enumeration capable endpoints in the system. Typically, only one or two processors with endpoints will be designated as enumerating hosts. Out of the competing enumeration hosts, only one host can win. The losing hosts and other non-enumerating processors are forced to wait until enumeration is complete. At that point, they may traverse the network to find all devices without disturbing the network configuration. When discovery completes, the Linux

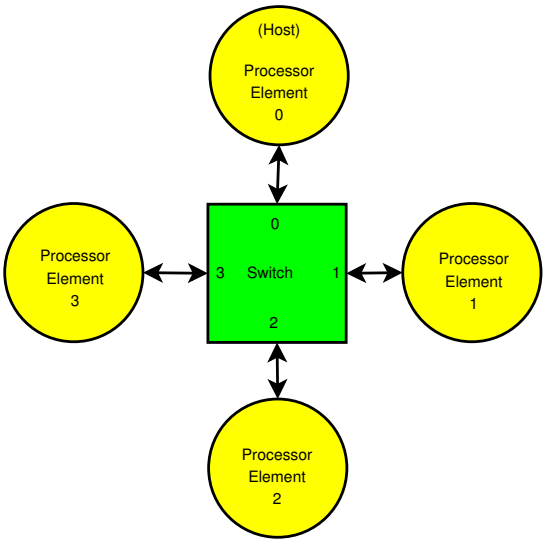

Figure 6: Example RapidIO System

RapidIO subsystem will have a complete view of all RapidIO devices in the network.

In the passive discovery process, the network is walked depth first as with enumeration. However, the existing route table entries are utilized to generate transactions that pass through a switch. When an endpoint device is discovered, a `rio_dev` is allocated but the device ID is retrieved from the value written in the Base Device ID Register. When a switch device is found, discovery iterates over each active switch port as with enumeration. However, in order to generate transactions for devices beyond that switch port, the routing table is scanned for an entry which is routed out that switch port. Using the device ID associated with the switch port, discovery issues a transaction with the associated device ID and a hopcount equal to the number of hops into the network. The process continues in a similar manner as described with enumeration until all devices have been discovered.

3.5 Enumeration and Discovery Example

Figure 6 illustrates a typical RapidIO system. There are four processor elements (PEs) numbered zero through three. Each PE provides RapidIO endpoint functionality and is connected to each of four ports on the switch in the center. PE 0 is the only designated enumerating host in the system and is assigned a host device ID of 0. PEs 1-3 do not perform enumeration, but rather wait for the signal indicating that enumeration has been completed by PE 0.

PE 0 begins enumeration by attempting to obtain the host device ID lock on the adjoining device. The transaction to configuration space is issued with a hopcount of 0 and a device ID of 0xFF. Since the hopcount of the transaction is 0, the switch will process the request and allow PE 0 to obtain the lock. Once the lock is obtained, PE 0 queries the device to learn that it is a switch and allocates `rio_dev` and `rio_switch` structures.

PE 0 queries the switch to determine that there are 4 ports with active links present. PE 0 then begins a loop to iterate over the 4 active ports, skipping the input port which it is using to access the switch device. For each active switch port, PE 0 performs the following:

1. Writes a route entry that assigns device ID 0xFF to the current active switch port.

2. Issues configuration space transactions with a hopcount of 1 to access the devices that are one hop from PE 0:

   - Obtains the host device ID lock for each device.
   - Queries the device to determine that it is an endpoint and allocates a `rio_dev` structure.

- Assigns the next available device ID to the endpoint. PEs 1-3 are assigned device IDs 0x01-0x03, respectively.

3. Assigns route entries corresponding to the switch ports where the PEs were discovered. Route entries for device IDs 0x01-0x03 are assigned to switch ports 1-3, respectively.

After this process completes, PE 0 writes the magic "enumeration complete" value into the Component Tag Register on each device. This is followed by PE 0 releasing the host device ID lock on each device in the system. Once PEs 1-3 detect that enumeration is complete, they are free to begin their discovery process.

3.6 Driver Interface

RapidIO device drivers are provided a specific set of functions to use in their implementation. In order to guarantee proper functioning of the subsystem, drivers may not access hardware resources directly.

Configuration space access is managed similar to configuration space access in the PCI subsystem.

- `rio_config_read_8()`
  `rio_config_read_16()`
  `rio_config_read_32()`
  `rio_config_write_8()`
  `rio_config_write_16()`
  `rio_config_write_32()`

  Read or write a specific size at an offset of a device.

- `rio_local_config_read_8()`
  `rio_local_config_read_16()`
  `rio_local_config_read_32()`
  `rio_local_config_write_8()`
  `rio_local_config_write_16()`
  `rio_local_config_write_32()`

  Read or write a specific size at an offset of the local master port's configuration space.

Several calls handle the ownership and initialization of mailbox and doorbell resources on a master port or remote device.

- `rio_request_outb_mbox()`
  `rio_request_inb_mbox()`

  Claim ownership of an outbound or inbound mailbox, initialize the mailbox for processing of messages, and register a notification callback. The outbound mailbox callback provides a interrupt context event when a message has been sent. The inbound mailbox callback provides an event when a message has been received.

- `rio_release_outb_mbox()`
  `rio_release_inb_mbox()`

  Give up ownership of an outbound or inbound mailbox and unregister notification callback.

- `rio_request_outb_dbell()`

  Claim ownership of a range of doorbells on a remote device. Ownership is only valid for the local processor.

- `rio_request_inb_dbell()`

  Claim ownership of a range of doorbells on the inbound doorbell queue, initialize the doorbell queue, and register a callback. The doorbell callback provides an event when a doorbell within the registered range is received.

- `rio_release_outb_dbell()`

  Give up ownership of a range of doorbells on a remote device.

- `rio_release_inb_dbell()`

  Give up ownership of a range of doorbells on the inbound doorbell queue.

Several calls provide access to doorbell and message queues.

- `rio_send_doorbell()`

  Send a doorbell message to a specific device.

- `rio_add_outb_message()`

  Add a message to an outbound mailbox queue.

- `rio_add_inb_buffer()`

  Add an empty buffer to an inbound mailbox queue.

- `rio_get_inb_message()`

  Get the next available message from an inbound mailbox queue.

3.7 Architecture Interface

Every architecture must provide implementations for a set of RapidIO functions. These functions manage hardware-specific features of configuration space access, mailbox access, and doorbell access.

- `rio_ops.lcwrite()`
  `rio_ops.lcread()`
  `rio_ops.cwrite()`
  `rio_ops.cread()`

  Hardware specific implementations for generation of read and write transactions to configuration space. These master port specific routines are assigned to a `struct rio_ops` which is in turn bound to a `struct rio_mport`. These low-level operations are used by the driver interface configuration space access routines.

- `rio_ops.dsend()`

  Hardware specific implementation for generation of a doorbell write transaction. This master port specific routine is assigned to a `struct rio_mport` and used by the `rio_send_doorbell()` call.

- `rio_hw_open_outb_mbox()`
  `rio_hw_open_inb_mbox()`

  Hardware specific initialization for outbound and inbound mailbox queues.

- `rio_hw_close_outb_mbox()`
  `rio_hw_close_inb_mbox()`

  Hardware specific cleanup for outbound and inbound mailbox queues.

- `rio_hw_add_outb_message()`

  Hardware specific implementation to add a message buffer to the outbound mailbox queue.

- `rio_hw_add_inb_buffer()`

  Hardware specific implementation to add an empty buffer to the inbound mailbox queue.

- `rio_hw_get_inb_message()`

  Hardware specific implementation to get the next available inbound message.

An architecture must also implement interrupt handlers for mailbox and doorbell queue events. Typically, inbound doorbell and mailbox hardware will generate a hardware interrupt to indicate that a message has arrived. Outbound doorbell hardware will typically generate a hardware interrupt when a message has

been successfully sent. The architecture interrupt handler must process the event in an appropriate manner for the message type and acknowledge the hardware interrupt.

For inbound doorbell messages, the handler must extract the doorbell message info and check for a callback that has been registered for the doorbell message it has received. If a callback has been registered (using `rio_request_inb_dbell()`) for a doorbell range that includes the received doorbell message, the callback is executed. The callback indicates the source, destination, and 16-bit info field (the doorbell message) that was received.

A mailbox interrupt handler must execute the registered callback for the mailbox that generated the hardware interrupt. It may be required to do some hardware-specific ring buffer management and must acknowledge the hardware interrupt. The callback is registered using `rio_request_inb_mbox()` or `rio_request_outb_mbox()`

3.8 Device Model

The RapidIO subsystem ties into the Linux Device Model in a similar way to most other device subsystems. A RapidIO bus is registered with the device subsystem and each RapidIO device is registered as a child of that bus. RapidIO specific `match` and `dev_attrs` implementations are provided.

`rio_match_bus()` implementation is a simple device to driver matching implementation. It compares vendor and device IDs of a candidate RapidIO device to determine if a driver will claim ownership of the device.

The `rio_dev_attrs[]` implementation exports all of the common register fields in the `rio_dev` structure to sysfs. In addition to the standard `dev_attrs` sysfs support, a *config* node is exported similar to the same node in the PCI subsystem. It provides userspace access to the 2MB configuration space on each RapidIO device.

RapidIO specific implementations of `probe()`, `remove()`, and driver register/unregister are also provided.

4 RapidIO Messaging Network Driver (*rionet*)

4.1 *rionet* Overview

With the subsystem in place, a driver is still needed to make use of the new functionality. Since the first RapidIO parts available are processors with RapidIO interfaces, a network driver to provide communication over the RapidIO switched fabric makes good sense. The RapidIO messaging model makes this easy since managing outbound and inbound messages is much like a managing a modern descriptor-based network controller.

4.2 *rionet* Features

rionet has the following features:

- Ethernet driver model for simplicity
- Dynamic discovery of network peers using doorbell messages
- Unique MAC address generation based on RapidIO device ID
- Maximum MTU of 4082
- Uses standard RapidIO subsystem message model to work on any RapidIO endpoints with mailboxes and doorbells

4.3 *rionet* Implementation

The *rionet* driver is initialized with a `rio_register_driver()` call. The `id_table` is configured to match all RapidIO devices so that the `rionet_probe()` call will qualify *rionet* devices. The probe routine verifies that the device has mailbox and doorbell capabilities. If the device is mailbox and doorbell capable, then it is added to a list of potential *rionet* peers. If at least one potential peer is found, the local RapidIO device is queried for its device ID. The MAC address is generated by concatenating 3 bytes of a well known Ethernet test network address with a 1 byte zero pad and finally the 2 byte device ID of the local device.

When *rionet* is opened, it requests a range of doorbell messages and registers a doorbell callback to process doorbell events. Two messages, `RIONET_JOIN` and `RIONET_LEAVE`, are defined to manage the active peer discovery process. For each device in the potential peer list, the `RIONET_JOIN` and `RIONET_LEAVE` outbound doorbell resources are claimed. After verifying that the potential peer device has initialized inbound doorbell service, a `RIONET_JOIN` doorbell is sent to it.

The doorbell event handler processes a `RIONET_JOIN` doorbell by doing the following:

1. Adds the originating device ID to the active peer list.
2. Sends a `RIONET_JOIN` doorbell as a reply to the originator.

If a `RIONET_LEAVE` doorbell is received, the originating device ID is removed from the active peer list.

rionet is designed such that it defaults to the maximum allowable MTU size. With a maximum RapidIO message payload of 4096 bytes, the default MTU size is 4082 after allowing for the 14 byte Ethernet header overhead. Due to the inclusion of the RapidIO device ID in the generated MAC address, Ethernet packets in this driver contain all the information required to send the packets over RapidIO.

The `hard_start_xmit()` implementation in *rionet* is similar to any standard Ethernet driver except that it must verify that a destination node is active before queuing a packet. The active peer list that was created during the *rionet* discovery process is used for this verification. The least significant 2 bytes of the destination MAC address are used to index into the active peer list to verify that the node is active. If the node is active, then the packet is queued for transmission using `rio_add_outb_message()`. Housekeeping for freeing of completed skbs is handled using the outbound mailbox transmission complete event. This is similar to how a standard Ethernet driver uses a direct hardware interrupt event for TX complete events.

Ethernet packet reception is also very similar to standard Ethernet drivers. In this case, it is driven from the inbound mailbox event handler. This callback is executed when the hardware mailbox receives an inbound message in its queue. `rio_get_inb_message()` is used to retrieve the next inbound Ethernet packet from the inbound mailbox queue. As skbs are consumed, a ring refill function adds additional empty skbs to the inbound mailbox queue using `rio_add_inb_buffer()`.

The result is an Ethernet compatible driver which can be used to leverage the huge set of TCP/IP userspace applications for development, testing, and deployment. The Ethernet implementation allows routing between *rionet* and wired Ethernet networks, opening up

many interesting application possibilities. It is possible to provide the root file system to nodes via NFS over RapidIO. Coupling this with firmware support for booting over RapidIO, it is possible to boot an entire network of RapidIO processor devices over the RapidIO network.

5 Going Forward

Although the Linux RapidIO subsystem encapsulates much of the hardware functionality of RapidIO, a few areas have been left incomplete. The following features are in development or planned for development.

- In the future, the Linux RapidIO subsystem will add an interface for managing MMIO regions which are mapped to per-device address spaces. As a part of this effort, mmapable sysfs nodes for each region will be exported for use from userspace.

- Although parallel RapidIO provided the first available RapidIO hardware, 16-bit device ID addressable serial RapidIO is the direction where all future hardware is heading. The subsystem is being extended to handle 16-bit device IDs and the serial RapidIO physical layer.

- In order to make use of the standardized error reporting facilities in RapidIO, an interface will be required to register and process Port Write Events. These are unsolicited transactions which are reported to a specified host in RapidIO Typically, they will be used for error reporting.

6 Conclusion

Today, the Linux RapidIO subsystem provides a complete layer for initialization of a RapidIO network and a driver interface for message passing based drivers. The message passing network driver, *rionet*, provides a simple mechanism for application developers to take advantage of RapidIO messaging. As new RapidIO devices are released, *rionet* will serve as a reference driver for authors of new RapidIO device drivers.

References

[1] RapidIO Trade Association. RapidIO Interconnect Specification. `http://www.rapidio.org`.

Locating System Problems Using Dynamic Instrumentation

Vara Prasad *IBM*
prasadav@us.ibm.com

William Cohen *Red Hat, Inc.*
wcohen@redhat.com

Frank Ch. Eigler *Red Hat, Inc.*
fche@redhat.com

Martin Hunt *Red Hat, Inc.*
hunt@redhat.com

Jim Keniston *IBM*
jkenisto@us.ibm.com

Brad Chen *Intel Corporation*
brad.chen@intel.com

Abstract

Diagnosing complex performance or kernel debugging problems often requires kernel modifications with multiple rebuilds and reboots. This is tedious, time-consuming work that most developers would prefer to minimize.

Systemtap uses the kprobes infrastructure to dynamically instrument the kernel and user applications. Systemtap instrumentation incurs low overhead when enabled, and zero overhead when disabled. SystemTap provides facilities to define instrumentation points in a high-level language, and to aggregate and analyze the instrumentation data. Details of the SystemTap architecture and implementation are presented, along with an example of its application.

1 Introduction

This paper introduces SystemTap, a new performance and kernel troubleshooting infrastructure for Linux. SystemTap provides a scripting environment that can eliminate the modify-build-test loop often required for understanding details of Linux kernel behavior. SystemTap is designed to be sufficiently robust and efficient to support applications in production environments. Our broad goals are to reduce the time and complexity for analyzing problems that involve kernel activity, to greatly expand the community of engineers to which such analyses are available, and to reduce the need to modify and rebuild the kernel as a troubleshooting technique.

Today, identifying functional problems in Linux systems often involves modifying kernel source with diagnostic print statements. The process can be time-consuming and require detailed knowledge of multiple subsystems. SystemTap uses dynamic instrumentation to make this same level of data available without the need to modify kernel source or rebuild the kernel. It delivers this data via a powerful scripting facility. Interesting problem-analysis tools can be implemented as simple scripts.

SystemTap is also designed for analyzing system-wide performance problems. While existing Linux performance tools like `iostat`, `vmstat`, `top`, and `oprofile` are valuable

for understanding certain types of performance problems, there are many kinds of problems that they don't readily expose, including:

- Interactions between applications and the operating system
- Interactions between processes
- Interactions between kernel subsystems
- Problems that are obscured by ordinary behavior and require examination of an activity trace

Often these problems are difficult to reproduce in a test environment, making it desirable to have a tool that is sufficiently flexible, robust and efficient to be used in production environments. These scenarios further motivate our work on SystemTap.

SystemTap builds on, and extends, the capabilities of the `kprobes` [6, 7] kernel debugging infrastructure. SystemTap has been influenced by a number of earlier systems, including kerninst [9], Dprobes [6], the Linux Trace Toolkit (LTT) [10], the Linux Kernel State Tracer (LKST) [1], and Solaris DTrace [5, 8].

This paper starts with a brief discussion of the existing dynamic instrumentation provided by Kprobes in the Linux 2.6 kernel, and explains the disadvantages of this approach. Next we describe a few key aspects of the SystemTap design, including the programming environment, the tapset abstraction, and safety in SystemTap. We continue with an example that illustrates the power of SystemTap for troubleshooting performance problems that are difficult to address with existing Linux tools. We close the paper with conclusions and future work.

2 Kprobes

Kprobes, a new feature in the Linux 2.6 kernel, allows for dynamic, in-memory kernel instrumentation. To use kprobes, the developer creates a loadable kernel module with calls into the kprobes interface. These calls specify a kernel instruction address, the *probe point*, and an analysis routine or *probe handler*. Kprobes arranges for control flow to be intercepted by patching the probe point in memory, with control passed to the probe handler. Kprobes has been carefully designed to allow safe insertion and removal of probes and to allow instrumentation of almost any kernel routine. It lets developers add debugging code into a running kernel. Because the instrumentation is dynamic, there is no performance penalty when probes are not used.

The basic control flow interception facility of kprobes has been enhanced with a number of additional facilities. *Jprobes* makes it easy to trace function calls and examine function call parameters. *Kretprobes* is used to intercept function returns and examine return values. Although it is a powerful system for dynamic instrumentation, a number of limitations prevent kprobes from broader use:

- Kprobes does very little safety checking of its probe parameters, making it easy to crash a system through accidental misuse.
- Safe use of kprobes often requires detailed knowledge of the code path to be instrumented. This limits the group of developers who will use kprobes.
- Due to references to kernel addresses and specific kernel symbols, the portability of the instrumentation code using the kprobes interface is poor. This lack of portability also limits re-usability of kprobes-based instrumentation.

- Kprobes does not provide a convenient mechanism to access a function's local variables, except for a jprobe's access to the arguments passed into the function.

- Although using kprobes doesn't require a kernel build-install-reboot, it does require knowledge to build a kernel module and lacks the support library routines for common tasks. This is a significant barrier for potential users. A script-based system that provides the support for common operations and hides the details of building and loading a kernel module will serve a much larger community.

These limitations are part of our motivation for creating SystemTap.

3 SystemTap

SystemTap [2] is being designed and developed to simplify the development of system instrumentation. The SystemTap scripting language allows developers to write custom instrumentation and analysis tools to address the performance problems they are examining. It also improves the reuse of existing instrumentation. Thus, people can build on the expertise of other developers who have already created instrumentation for specific kernel subsystems.

Portability is a concern of SystemTap. The intent is to provide SystemTap on all architectures to which kprobes has been ported.

Safety of the SystemTap instrumentation is another major concern. The tools minimize the chance that the SystemTap instrumentation will cause system crashes or corruption.

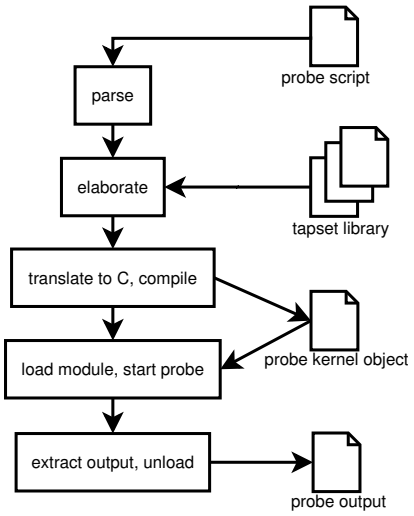

Figure 1: SystemTap processing steps

3.1 SystemTap processing steps

The steps SystemTap uses to convert an instrumentation script into executable instrumentation and extract collected data are shown in Figure 1. SystemTap takes a compilation approach to generate instrumentation code, unlike the interpreter approach other similar systems have taken [6, 5, 8]. A compiler converts the instrumentation script and tapset library into C code for a kernel module. After compilation and linking with the SystemTap runtime, the kernel module is loaded to start the data collection. Data is extracted from module into user-space via reliable and high performance transport. Data collection ends when the module is unloaded from the kernel. The elaboration, translation, and execution steps are described in greater detail in the following subsections.

3.2 Probe language

The SystemTap input consists of a script, written in a simple language described in Section 4. The language describes an association of handler subroutines with probe points. A *probe*

point may be a particular place in kernel/user code, or a particular event (timers, counters) that may occur at any time. A *handler* is a subroutine that is run whenever the associated probe point is hit.

The SystemTap language is inspired by the UNIX scripting language `awk` [4] and is similar in capabilities to `DTrace's` "D" [5]. It uses a simplified C-like syntax, lacking types, declarations, and most indirection, but adding associative arrays and simplified string processing. The language includes some extensions to interoperate with the target software being instrumented, in order to refer to its data and program state.

3.3 Elaboration

Elaboration is a processing phase that analyzes the input script and resolves references to the kernel or user symbols and *tapsets*. Tapsets are libraries of script or C code used to extend the capability of a basic script, and are described in Section 5. Elaboration resolves external references in the script file to symbolic information and imported script subroutines in preparation for translation to C. In this way, it is analogous to linking an object file with needed libraries.

References to kernel data such as function parameters, local and global variables, functions, and source locations all need to be resolved to actual run-time addresses. This is done by processing the DWARF debugging information emitted by the compiler during the kernel build, as is done in a debugger. All debug data processing occurs prior to execution of the resulting kernel module.

Debugging data contains enough information to locate inlined copies of functions (very common in the Linux kernel), local variables, types, and declarations beyond what are ordinarily exported to kernel modules. It enables placement of probe points in the interior of functions. However, the lack of debug data in some user programs (for example, stripped binaries) will limit SystemTap's ability to place probes in such code.

3.4 Translation

Once a script has been elaborated, it is translated into C.

Each script subroutine is expanded to a block of C that includes necessary locking and safety checks. Looping constructs are augmented with checks to prevent infinite loops. Each variable shared by multiple probes is mapped to an appropriate static declaration, and accesses are protected by locks. To minimize the use of kernel stack space, local variables are placed in a synthetic call frame.

Probe handlers are registered with the kernel using one of the `kprobes` [6, 7] family of registration APIs. For location-type probe points in the kernel, probe points are inserted in kernel memory. For user-level locations, the probe point is inserted in the executable code loaded into user memory while the probe handler is executed in the kernel.

The translated script includes references to a common runtime that provides routines for generic associative arrays, constrained memory management, startup, shutdown, I/O, and other functions.

When translation is complete, the generated C code is compiled and linked with the runtime into a stand-alone kernel module. The final module may be cryptographically signed for safe archiving or remote use.

3.5 Execution

After linking, the SystemTap driver program simply loads the kernel module using `insmod`. The module will initialize itself, insert the probes, then wait for probe points to be hit. When a probe is hit, the associated handler routine is invoked, suspending the thread of execution. When all handlers for that probe point have been executed, the thread of execution resumes. Because thread of execution is suspended, handlers must not block. Probe handlers should hold locks only while manipulating shared SystemTap variables, or as necessary to access previously unlocked target-side data.

The SystemTap script concludes when the user sends an interrupt to the driver program, or when the script calls `exit`. At the end of the run, the module is unloaded and its probes are removed.

3.6 Data Collection and Presentation

Data collected from SystemTap in the kernel must be transmitted to user space. This transport must provide high throughput and low latency, and impose minimal performance impact on the monitored system. Two mechanisms are currently being tested: relayfs and netlink.

Relayfs provides an efficient way to move large blocks of data from the kernel to user space. The data is sent via per-cpu buffers. Relayfs can be compiled into the kernel or built as a loadable module.

Netlink allows a simple stream of data to be sent using the socket APIs. Performance testing suggests that netlink provides less bandwidth than relayfs for transferring large amounts of trace data.

By default, SystemTap output will be processed in batches and written to `stdout` at script exit. The output will also be automatically saved to a file. SystemTap can optionally produce a real-time stream as required by the application.

In user-space, SystemTap can report data as simple text, or in structured computer-parsable forms for consumption by applications such as graphics generators.

4 SystemTap Programming Language

A SystemTap script file is a sequence of top-level constructs, of which there are three types: probe definitions, auxiliary function definitions, and global variable declarations. These may occur in any order, and forward references are permitted.

A probe definition identifies one or more probe points and a body of code to execute when any of them is hit. Multiple probe handlers may execute concurrently on a multiprocessor. Multiple probe definitions may end up referring to the same event or program location: all of them are run in an unspecified sequence when the probe point is hit. For tapset builders, there is also a probe aliasing mechanism discussed in Section 5.1

An auxiliary function is a subroutine for probe handlers and other functions. In order to conserve stack space, Systemtap limits the number of outstanding nested or recursive calls. The translator provides a number of built-in functions, which are implicitly declared.

A global variable declaration lists variables that are shared by all probe handlers and auxiliary functions. (If a variable is not declared global, it is assumed to be local to the function or probe that references it.)

A script may make references to an identifier defined elsewhere in the library of script tapsets. Such a cross-reference causes the entire tapset file providing the definition to be merged into the elaborated script, as if it was simply concatenated. See Section 5 for more information about tapsets.

Fatal errors that occur during script execution cause a cleanup of activity associated with the SystemTap script, and an early abort. Running out of memory, dividing by zero, exceeding an operation count limit and calling too many nested functions are a few types of errors that will terminate a script.

4.1 Probe points

A probe definition specifies one or more probe points in a comma-separated list, and an associated action in the form of a statement block. A trigger of any of the probe points will run the block. Each probe point specification has a "dotted-functor" syntax such as `kernel.function("foo").return`. The core SystemTap translator recognizes a family of these patterns, and tapsets may define new ones. The basic idea of these patterns is to provide a variety of user-friendly ways to refer to program spots of interest, which the translator can map to a kprobe on a particular PC value or an event.

The first group of probe point patterns relates to program points in the kernel and kernel modules. The first element, `kernel` or `module("foo")`, identifies the probe's target software as kernel or a kernel module named `foo.ko`. This first element is used to find the symbolic debug information to resolve the rest of the pattern.

For a probe point defined on a statically known symbol or other program structure, the translator can use debug information to expose local variables within the scopes of the active functions to the script.

4.1.1 Functions

To identify a function, the `function("fn")` element does so by name. If the function is inlineable, all points of inlining are included in the set. The function name may be suffixed by `@filename` or even `@filename:lineno` to identify a source-level scope within which the identifiers should be searched. The function name may include wildcard characters `*` and `?` to refer to all suitable matching names. These may expand to a huge list of matches, and therefore must be used with discretion. The optional element `return` may be added to refer to the moment of each function's return rather than the default `entry`. Below are some sample specifications for function probe points:

```
kernel.function("sys_read")
   .return
```
A return probe on the named function.

```
module("ext3").function("*@fs/
   ext3/inode.c")
```
Every function in the named source file, which is part of ext3fs.

4.1.2 Events

Probe points may be defined on abstract events, which are not associated with particular locations in the target program. Therefore, the translator cannot expose much symbolic information about the context of the probe hit to the script. Examples of probes that would fall in this category include probes that perform sampling based on timers or performance monitoring hardware, and probes that watch for changes in a variable's value.

SystemTap defines special events associated with initialization and shutdown of the instrumentation. The special element `begin` triggers a probe handler early during SystemTap initialization, before normal probes are enabled. Similarly, `end` triggers a probe during late shutdown, after all normal probes have been disabled.

4.2 Language Elements

Function and probe handler bodies are written using standard statement/expression syntax that borrows heavily from awk and C. The SystemTap language allows the C, C++, and awk style comments. White space and comments are treated as in C.

SystemTap identifiers have the same syntax as C identifiers, except that `$` is also a legal character. Identifiers are used to name variables and functions. Identifiers that begin with `$` are interpreted as references to variables in the target software, rather than to SystemTap script variables.

The language includes a small number of data types, but no type declarations: a variable's type is inferred from its use. To support this, the translator enforces consistent typing of function arguments and return values, array indexes and values. Similarly, there are no implicit type conversions between strings and numbers.

- Numbers are 64-bit signed integers. Literals can be expressed in decimal, octal, or hexadecimal, using C notation. Type suffixes (e.g., `L` or `U`) are not used.
- Strings. Literals are written as in C. Overall lengths are limited by the runtime system.
- Associative arrays are as in awk. A given array may be indexed by any consistent combination of strings and numbers, and may contain strings, numbers, or statistical objects.
- Statistics. These are special objects that compute aggregations (statistical averages, minima, histograms, etc.) over numbers.

The language has traditional `if-then-else` statements and expressions of C and awk. The language also allows structured control statements such as `for` and `while` loops. Unstructured control flow operations such as labels and `goto` statements are not supported. The translator inserts runtime checks to bound the number of procedure calls and backward branches.

To support associative arrays, the SystemTap language has iterator and `delete` statements. The iterator statement allows the programmer to specify an operation to perform on all the elements in the associative array. The delete operation can remove one or all the elements in the associative array. The associative arrays allow selection of an item by one or more keys. The `in` operation allows the code to determine whether an entry exists in the associative array.

The typical set of arithmetic, bit, assignment, and unary operations in C are available in the SystemTap language, but they operate on 64-bit quantities. The assignment and comparison operations are overloaded for strings.

The SystemTap statistic type allows script writers to keep track of the typical statistics such as minimum, maximum, and average. The `<<<` operator updates a variable storing statistics information as shown in the example below:

```
global avg(s)
probe kernel.syscall("read") {
```

```
  process->s <<< $size
}
probe end {
  trace (s)
}
```

SystemTap does not support type casts, address-of operations, or following of arbitrary pointers through structures. However, macro operations will allow access to elements of a particular structure.

4.3 Auxiliary functions

An auxiliary function in SystemTap has essentially the same syntax and semantics as in awk. Specifically, an auxiliary function definition consists of the keyword `function`, a formal argument list and a brace-enclosed statement block. SystemTap deduces the types of the function and its arguments from the expressions that refer to the function. An auxiliary function must always return a value even if it is ignored.

5 Tapsets

When diagnosing systemic problems, one is faced with tracing various subsystems of the operating system and applications. To facilitate such diagnosis, SystemTap includes a library of instrumentation modules for various subsystems known as *tapsets*. The list of available tapsets is published for use in end-user scripts. There are two ways to create tapsets: via the SystemTap scripting language and via the C language.

5.1 Script tapsets

The simplest kind of tapset is one that uses the SystemTap script language to define new probes, auxiliary functions, and global variables, for invocation by an end-user script or another tapset. One can use this mechanism to define commonly useful auxiliary functions like `stp_print()` for special purpose formatting of output data. This facility can also be used to create global variables that can be referenced in the end user scripts as built-in functions. In Figure 2 a `tgid_history` global variable is created that gives a history of the last few scheduled tasks.

In addition, a script tapset can define a *probe alias*. Aliasing is a way of synthesizing a higher level probe from a lower level one. The example tapset shown in Figure 3 defines aliases for the `read` system call, so that a SystemTap user does not have to know the name of the corresponding kernel function.

Aliasing consists of renaming a probe point, and may include some script statements. These statements are all executed *before* the others that are within the user's probe definition (which referenced the alias), as if they were simply transcribed there. This way, they can prepare some useful local variables, or even conditionally reject a probe hit using the `next` statement.

Aliases can also be used to define a new "event" and supply some local variables for use by its handlers as in Figure 4.

An end-user script that uses the probe alias in Figure 4 may look like Figure 5.

5.2 C language tapsets

To allow kernel developers to work in a familar programming language, SystemTap supports a C interface for creating tapsets. A C tapset is a set of data-collection functions for a given subsystem. Data collection functions

```
global tgid_history # the last few tgids scheduled

global _histsize

probe begin {
  _histsize = 10
}

probe kernel.function("context_switch") {
  # rotate array
  for (i=_histsize-1; i>0; i--)
    tgid_history [i] = tgid_history [i-1];
  tgid_history [0] = $prev->tgid;
}
```

Figure 2: SystemTap script using global variable.

```
probe kernel.syscall.read = kernel.function("sys_read")
{ }
```

Figure 3: SystemTap script using probe alias.

```
probe kernel.resource.oom.nonroot =
  kernel.statement("do_page_fault").label("out_of_memory") {
   if ($tsk->uid == 0) next;

   victim_tgid = $tsk->tgid;
   victim_pid = $tsk->pid;
   victim_uid = $tsk->uid;
   victim_fault_addr = $address
}
```

Figure 4: SystemTap script for new out of memory event.

```
probe kernel.resource.oom.nonroot {
  trace ("OOM for pid " . string (victim_pid))
}
```

Figure 5: SystemTap script using out of memory event.

in the tapset are called tapset functions. Tapset functions export data using one or more variables. The C API requires a tapset writer to register each probe point, corresponding data-collection function, and the data exported by the function. When an end-user script refers to the data exported by the corresponding tapset function in the action block, SystemTap calls the associated tapset function in the probe handler. The result is that local variables in the user script are initialized with values from the tapset function.

5.3 System call tapset

SystemTap provides tapsets for various subsystems of the kernel; the system call tapset is an example of one such tapset. As system calls are the primary interface for applications to interact with the kernel, understanding them is a powerful diagnostic tool. The system call tapset provides a probe handler for each system call entry and exit. A system call entry probe gives the values of the arguments to the system call, and the exit probe gives the return value of the system call.

6 Safety

SystemTap is designed for safe use in production systems. One implication is that it should be extremely difficult, if not impossible, to disable or crash a system through use or misuse of SystemTap. Problems like infinite loops, division by zero, and illegal memory references should lead to a graceful failure of a SystemTap script without otherwise disrupting the monitored system. At the same time, we'd like to compile extensions to native machine code, to benefit from the stability of the existing tool chain, minimize new kernel code, and approach native performance.

Our basic approach to safety is to design a safe scripting language, with some safety properties supported by runtime checks. Table 1 provides some details of our basic approach. SystemTap compiles the script file into native code and links it with the SystemTap runtime library to create a loadable kernel module. Version and symbol name checks are applied by `insmod`. The elaborator generates instrumentation code that gracefully terminates loops and recursion, if they run beyond a configurable threshold. We avoid privileged and illegal kernel instructions by excluding constructs in the script language for inlined assembler, and by using compiler options used for building kernel modules.

SystemTap incorporates several additional design features that enhance safety. Explicit dynamic memory allocation by scripts is not allowed, and dynamic memory allocation by the runtime is avoided. SystemTap can frequently use explicitly synthesized frames in static memory for local variables, avoiding usage of kernel stack. Language and runtime systems ensure that SystemTap-generated code for probe handlers is strictly terminating and non-blocking.

SystemTap safety requires controlling access to kernel memory. Kernel code cannot be invoked directly from a SystemTap script. SystemTap language features make it impossible to express kernel data writes or to store a pointer to kernel data. Additionally, a modified trap handler is used to safely handle invalid memory references. SystemTap supports a "guru" mode where certain of these constraints can be removed (e.g., in a tapset), allowing a tradeoff between safety and kernel debugging requirements.

6.1 Safety Enhancements

A number of options are planned that extend the safety and flexibility of SystemTap to match

	language design	translator	insmod checks	runtime checks	memory portal	static validator
infinite loops		x		o		o
recursion		x		o		o
division by zero		x		o		o
resource constraints		x		x		
locking constraints		x		x		
array bounds errors	x	x		x		o
invalid pointers		o		o		o
heap memory bugs	x					o
illegal instructions	x				o	
privileged instructions	x				o	
memory r/w restrictions	x			x	o	o
memory execute restrictions	x			x	o	o
version alignment		o	x			
end-to-end safety					x	x
safety policy specification facility					x	

Table 1: SystemTap safety mechanisms. An “x” indicates that an aspect of the implementation (columns) is used to implement a particular safety feature (rows). An “o” indicates optional functionality.

and exceed that of other systems. A memory and code "portal" directs references to kernel memory outside the loadable module through a special-purpose interpeter or "portal." This provides a single point of control for related safety issues, and facilitates a desireable separation of safety policy from mechanism. Trivial policies would support "guru mode" (no restrictions) and default mode (read restrictions to I/O memory, restricted write and code access). Other simple policies expand access incrementally, for example, allowing external calls to an explicit list of kernel subroutines. Eventually, the policy could be extended to support security goals such as secure non-root execution and restricting memory access based on user credentials.

An optional static analyzer examines a disassembled kernel module and confirms that it satisfies certain safety properties. Simple checks include disallowing privileged instructions, locking primitives and instructions that are illegal in kernel mode. In the future, more elaborate checks may be included to confirm that loop counters, memory portals and other safety features are used.

6.2 Comparision to Other Systems

Solaris DTrace includes a number of unusual features intended to enhance the safety and security of the system. These features include a very restricted scripting language and the scripts being interpreted rather than compiled.

DTrace's D language does not support procedure declarations or a general purpose looping construct. This avoids a number of safety issues in scripts including infinite loops and infinite recursion.

Because D scripts are interpreted rather than executed directly, it is impossible for them to include illegal or privileged instructions or to invoke code outside of the DTrace execution environment. The interpreter can also catch invalid pointer dereferences, division by zero, and other run-time errors.

SystemTap will support kernel debugging features in guru mode that DTrace does not, including the ability to write arbitrary locations in kernel memory and the ability to invoke arbitrary kernel subroutines.

Because the language infrastructure used by SystemTap is common to all C programs, it tends to be better tested and more robust than the special-purpose interpreter used by DTrace.

The embedding of an interpreter in the Solaris kernel represents significant additional kernel functionality. This introduces an increased risk of kernel bugs that could lead to security or reliability issues.

Dprobes and Dtrace have many safety features in common. Both use an interpreted language. Like SystemTap, both use a modified kernel trap-handler to capture illegal memory references. Like kprobes, dprobes is intended for use primarily by kernel developers. Consequently, it exposes the kprobes layer in such a way that it is not crashproof. SystemTap seeks to address these safety issues.

6.3 Security

It is important that SystemTap can be used without significantly impacting the overall security of the system. Given that SystemTap is only available to privileged users, our initial security concerns are that the system be crashproof by design, and that its implementation is of sufficient quality and simplicity to protect users from unintentional lapses. A specific concern is the security of the communication layer;

that the kernel-to-user transport is secured from non-privileged users.

Future versions of SystemTap may provide features that support secure use of SystemTap by non-privileged users. Specific features that might be required include:

- Protection of kernel memory based on user credentials.
- Protection of kernel-to-user transport based on user credentials.
- Recognition of a restricted subset of the SystemTap language that is permissible for non-privileged users.

A security scheme based on a virtual machine monitor such as Xen [3] might provide a simpler and general solution to secure SystemTap use by non-privileged users.

7 Example SystemTap Script

The SystemTap scripting language lends itself to writing compact instrumentation. The following example demonstrates a simple script to collect information. On SMP machines, the interprocessor interrupt is an expensive operation. One can find how many interprocessor interrupts are performed on an SMP machine by examining the `LOC:` entry of `/proc/interrupts`. However, this entry does not give a complete picture of what is causing the interprocessor interrupts.

A developer would like to know the process (PID), the process name, and the backtrace to get a better context of what is triggering the interprocessor interrupts. Figure 6 shows the SystemTap script used to accumulate that information into an associative array. Each time `smp_call_function` is called, the appropriate associative array entry is incremented. The `$pid` provides the process id number, the `$pname` provides the name of the process, and `stack()` the back trace in the kernel. This data is recorded in an associative array `traces`. When the data collection is over and the instrumentation is removed, the "end probe" prints out information.

Figure 7 shows the beginning of the data generated from a dual processor x86-64 machine when a DVD has just been loaded on the machine. From the samples listed below, we see that process 4010, hald, has caused a number of interprocessor interrupts. With the stack backtrace as part of the hash key, we can see that the first entry has to do with the disk change in the CDROM drive, and the second entry is caused by `sys_close`.

8 Conclusions and Future Work

We have described current dynamic instrumentation facilities in the Linux kernel and the need for improvements. These motivate the SystemTap architecture and salient features of its scripting language. We described the tapset library and its importance in SystemTap. Safety is a very important consideration of SystemTap design and we described how safety considerations impacted our SystemTap design. We presented an example of how SystemTap is used to gather interesting data to diagnose a problem. The Systemtap project is still in development. In our continuing work, we plan to implement tapset libraries for various kernel subsystems, and expand SystemTap to trace user-level activity.

```
global traces

probe kernel.function("smp_call_function") {
  traces[$pid, $pname, stack()] += 1;
}

probe end {
  print(traces);
}
```

Figure 6: SystemTap script to collect interprocessor interrupt information.

```
root# stp scf.stp
Press Control-C to stop.
All kprobes removed
traces[4010, hald, trace for 4010 (hald)
0xffffffff8011a551 : smp_call_function+0x1/0x70
0xffffffff80182c0c : invalidate_bdev+0x1c/0x40
0xffffffff8019bc48 : __invalidate_device+0x58/0x70
0xffffffff80188f89 : check_disk_change+0x39/0xa0
0xffffffff80133c90 : default_wake_function+0x0/0x10
0xffffffff802abeef : cdrom_open+0xa0f/0xa60
0xffffffff80133c90 : default_wake_function+0x0/0x10
0xffffffff80132650 : finish_task_switch+0x40/0x90
0xffffffff80346bb9 : thread_return+0x54/0x8b
0xffffffff801419cd : __mod_timer+0x13d/0x150
] = 18
traces[4010, hald, trace for 4010 (hald)
0xffffffff8011a551 : smp_call_function+0x1/0x70
0xffffffff80182c0c : invalidate_bdev+0x1c/0x40
0xffffffff8018856e : kill_bdev+0xe/0x30
0xffffffff801890d6 : blkdev_put+0x76/0x1c0
0xffffffff80181eb2 : __fput+0x72/0x160
0xffffffff801806de : filp_close+0x7e/0xa0
0xffffffff80180793 : sys_close+0x93/0xc0
0xffffffff8010e51a : system_call+0x7e/0x83
] = 27
```

...

Figure 7: Run of SMP call instrumentation.

9 Acknowledgements

We would like to express our thanks to Ananth N. Mavinakayanahalli, Hien Q. Nguyen, Prasanna S. Panchamukhi, and Thomas Zanussi for their valuable contributions to the project. The authors are indebted to Ulrich Drepper and Roland McGrath for their help and advice in the project. Thanks are in order to Suparna Bhattacharya and Richard Moore for sharing their knowledge of Kprobes and Dprobes. We would also like to thank Rohit Seth, Rusty Lynch, and Anil Keshavamurthy for Linux kernel and Itanium expertise. Special thanks to Doug Armstrong and Victoria Gromova for their input on features for parallel program analysis. Thanks to K. Sridharan and Charles Spirakis for studying support of common profiling tasks.

10 Trademarks and Disclaimer

References

[1] Linux kernel state tracer, May 2005. `http://lkst.sourceforge.net/`.

[2] Systemtap, May 2005. `http://sourceware.org/systemtap/`.

[3] The xen virtual machine monitor, May 2005. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/`.

[4] Alfred V. Aho, Brian K. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[5] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Levinthal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Technical Conference*, pages 15–28, June 2004.

[6] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In *FREENIX*, 2001.

[7] Prasanna S. Panchamukhi. Kernel debugging with kprobes: Insert printk's into linux kernel on the fly, Aug 2004. `http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnx%w07Kprobe`.

[8] Sun Microsystems, Santa Clara, California. *Solaris Dynamic Tracing Guide*, 2004.

[9] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[10] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *In Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.

Xen 3.0 and the Art of Virtualization

Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield
University of Cambridge
{first.last}@cl.cam.ac.uk

Dan Magenheimer
Hewlett-Packard Laboratories
{first.last}@hp.com

Jun Nakajima, Asit Mallick
Intel Open Source Technology Center
{first.last}@intel.com

Abstract

The Xen Virtual Machine Monitor will soon be undergoing its third major release, and is maturing into a stable, secure, and full-featured virtualization solution for Linux and other operating systems. Xen has attracted considerable development interest over the past year, and consequently the 3.0 release includes many exciting new features. This paper provides an overview of the major new features, including VM relocation, device driver isolation, support for unmodified operating systems, and new hardware support for both x86/64 and IA-64 processors.

1 VM Relocation

While many server applications may be very long-lived, the hardware that it runs on will invariably need service from time to time. A major benefit of virtualization is the ability to relocate a *running* operating system instance from one physical host to another. Relocation allows a physical host to be unloaded so that hardware may be serviced, it allows coarse-grained load-balancing in a cluster environment, and it allows servers to move closer to the users that they serve. By *pre-copying* VM state to the destination host while it is still running, relocation down-time can be made very small—experiments relocating a running Quake server have achieved repeatable relocation times with outages of less than 100ms.

In the following subsections we describe some of the implementation details of our pre-copying approach. We describe how we use dynamic network rate-limiting to effectively balance network contention against OS downtime. We then proceed to describe how we ameliorate the effects of rapid page dirtying, and show results for the relocation of a running Quake 3 server.

1.1 Managing Relocation

Relocation is performed by daemons running in the management VMs of the source and destination hosts. These are responsible for creating a new VM on the destination machine, and coordinating transfer of live system state over the network.

When transferring the memory image of the still-running OS, the control software performs

rounds of copying in which it performs a complete scan of the VM's memory pages. Although in the first round all pages are transferred to the destination machine, in subsequent rounds this copying is restricted to pages that were dirtied during the previous round, as indicated by a *dirty bitmap* that is copied from Xen at the start of each round.

During normal operation the page tables managed by each guest OS are the ones that are walked by the processor's MMU to fill the TLB. This is possible because guest OSes are exposed to real physical addresses and so the page tables they create do not need to be mapped to physical addresses by Xen.

To log pages that are dirtied, Xen inserts *shadow page tables* underneath the running OS. The shadow tables are populated on demand by translating sections of the guest page tables. Translation is very simple for dirty logging: all page-table entries (PTEs) are initially read-only mappings in the shadow tables, regardless of what is permitted by the guest tables. If the guest tries to modify a page of memory, the resulting page fault is trapped by Xen. If write access is permitted by the relevant guest PTE then this permission is extended to the shadow PTE. At the same time, we set the appropriate bit in the VM's dirty bitmap.

When the bitmap is copied to the control software at the start of each pre-copying round, Xen's bitmap is cleared and the shadow page tables are destroyed and recreated as the relocatee OS continues to run. This causes all write permissions to be lost: all pages that are subsequently updated are then added to the now-clear dirty bitmap.

When it is determined that the pre-copy phase is no longer beneficial, the OS is sent a control message requesting that it suspend itself in a state suitable for relocation. This causes the OS to prepare for resumption on the destination machine; Xen informs the control software once the OS has done this. The dirty bitmap is scanned one last time for remaining inconsistent memory pages, and these are transferred to the destination together with the VM's checkpointed CPU-register state.

Once this final information is received at the destination, the VM state on the source machine can safely be discarded. Control software on the destination machine scans the memory map and rewrites the guest's page tables to reflect the addresses of the memory pages that it has been allocated. Execution is then resumed by starting the new VM at the point that the old VM checkpointed itself. The OS then restarts its virtual device drivers and updates its notion of wallclock time.

1.2 Dynamic Rate-Limiting

It is not always appropriate to select a single network bandwidth limit for relocation traffic. Although a low limit avoids impacting the performance of running services, analysis showed that we must eventually pay in the form of an extended downtime because the hottest pages in the writable working set are not amenable to pre-copy relocation. The downtime can be reduced by increasing the bandwidth limit, albeit at the cost of additional network contention.

Our solution to this impasse is to dynamically adapt the bandwidth limit during each pre-copying round. The administrator selects a minimum and a maximum bandwidth limit. The first pre-copy round transfers pages at the minimum bandwidth. Each subsequent round counts the number of pages dirtied in the previous round, and divides this by the duration of the previous round to calculate the *dirtying rate*. The bandwidth limit for the next round is then determined by adding a constant increment to the previous round's dirtying rate—we

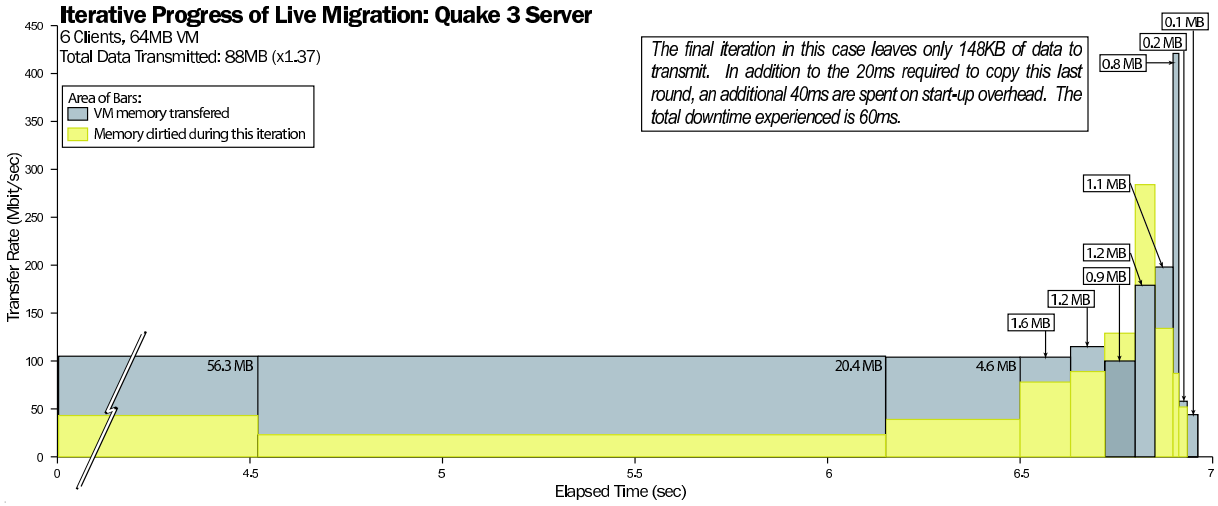

Figure 1: Results of relocating a running Quake 3 server VM.

have empirically determined that 50Mbit/sec is a suitable value. We terminate pre-copying when the calculated rate is greater than the administrator's chosen maximum, or when less than 256KB remains to be transferred. During the final stop-and-copy phase we minimize service downtime by transferring memory at the maximum allowable rate.

Using this adaptive scheme results in the bandwidth usage remaining low during the transfer of the majority of the pages, increasing only at the end of the relocation to transfer the hottest pages in the WWS. This effectively balances short downtime with low average network contention and CPU usage.

1.3 Rapid Page Dirtying

Analysis shows that every OS workload has some set of pages that are updated extremely frequently, and which are therefore not good candidates for pre-copy relocation even when using all available network bandwidth. We observed that rapidly-modified pages are very likely to be dirtied again by the time we attempt to transfer them in any particular pre-copying round. We therefore periodically 'peek' at the current round's dirty bitmap and transfer only those pages dirtied in the previous round that have not been dirtied again at the time we scan them.

We further observed that page dirtying is often physically *clustered*—if a page is dirtied then it is disproportionally likely that a close neighbour will be dirtied soon after. This increases the likelihood that, if our peeking does not detect one page in a cluster, it will detect none. To avoid this unfortunate behaviour we scan the VM's physical memory space in a pseudo-random order.

1.4 Low-Latency Server: Quake 3

A representative application for hosting environments is a multiplayer on-line game server. To determine the effectiveness of our approach in this case we configured a virtual machine with 64MB of memory running a Quake 3 server. Six players joined the game and started to play within a shared arena, at which point we initiated a relocation to another machine. A detailed analysis of this relocation is shown in Figure 1.

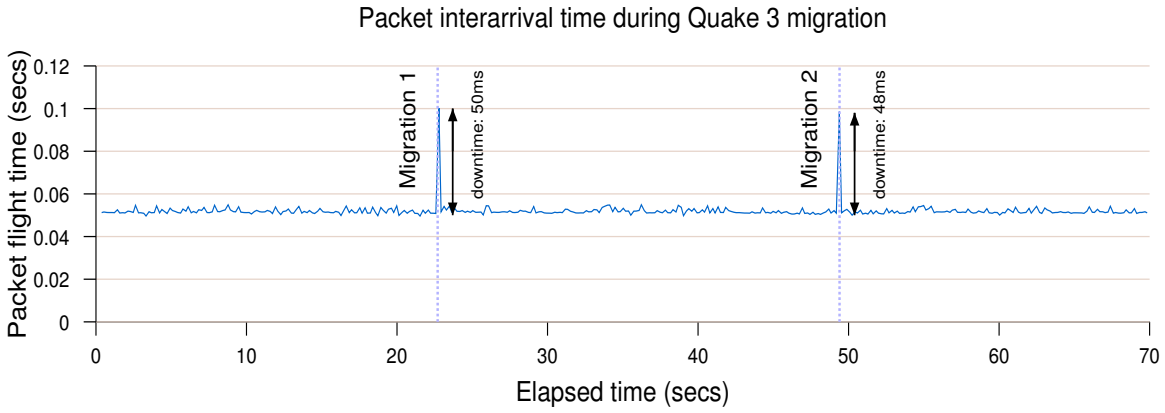

Figure 2: Effect on packet response time of relocating a running Quake 3 server VM.

We were able to perform the live relocation with a total downtime of 60*ms*. To determine the effect of relocation on the live players, we performed an additional experiment in which we relocated the running Quake 3 server twice and measured the inter-arrival time of packets received by clients. The results are shown in Figure 2. As can be seen, from the client point of view relocation manifests itself as a transient increase in response time of 50*ms*. In neither case was this perceptible to the players.

2 Device Virtualization

Xen's strong isolation guarantees have proved very useful in solving two major problems with device drivers: driver availability and reliability. Xen is capable of allowing individual virtual machines to have direct access to specific pieces of hardware. We have taken the approach of using a single virtual machine to run the physical driver for a device (such as a disk or network interface) and then export a virtualized version of the device to all of the other guest OSes that are running on the host. This approach means that a device need only be supported on a single platform (Linux, for instance), and may be available to all the OSes that Xen runs. Each guest implements an idealized disk and network device, which are capable of connecting to the hardware specific driver in an isolated *device domain*. This approach has the added benefit of making drivers, which are a major source of bugs in operating systems, more reliable. By running a driver in its own VM, driver crashes are limited to the driver itself—other applications may continue to run. Device domains can even be rebooted to recover failed drivers, and result in downtimes on the order of hundreds of miliseconds in cases where the entire machine would previously have crashed completely.

This approach will no doubt sound familiar to anyone who has worked with microkernels in the past—Xen's isolation achieves a similar fragmentation of OS subsystems. One major difference between Xen and historical work on microkernels is that we have forgone the architecturally pure fixation on IPC mechanisms in favour of a generalized, shared-memory ring-based communication primitive that is able to achieve very high throughputs by batching requests.

To achieve driver isolation, we restrict access privileges to device I/O registers (whether memory-mapped or accessed via explicit I/O

ports) and interrupt lines. Furthermore, where it is possible within the constraints of existing hardware, we protect against device misbehavior by isolating device-to-host interactions. Finally, we virtualize the PC's hardware *configuration space*, restricting each driver's view of the system so that it cannot see resources that it cannot access.

2.1 I/O Registers

Xen ensures memory isolation amongst domains by checking the validity of address-space updates. Access to a memory-mapped hardware device is permitted by extending these checks to allow access to non-RAM page frames that contain memory-mapped registers belonging to the device. Page-level protection is sufficient to provide isolation because register blocks belonging to different devices are conventionally aligned on no less than a page boundary.

In addition to memory-mapped I/O, many processor families provide an explicit I/O-access primitive. For example, the x86 architecture provides a 16-bit I/O port space to which access may be restricted on a per-port basis, as specified by an access bitmap that is interpreted by the processor on each port-access attempt. Xen uses this hardware protection by rewriting the port-access bitmap when context-switching between domains.

2.2 Interrupts

Whenever a device's interrupt line is asserted it triggers execution of a stub routine within Xen rather than causing immediate entry into the domain that is managing that device. In this way Xen retains tight control of the system by *scheduling* execution of the domain's interrupt service routine (ISR). Taking the interrupt in Xen also allows a timely acknowledgement response to the interrupt controller (which is always managed by Xen) and allows the necessary address-space switch if a different domain is currently executing. When the correct domain is scheduled it is delivered an asynchronous *event notification* which causes execution of the appropriate ISR.

Xen notifies each domain of asynchronous events, including hardware interrupts, via a general-purpose mechanism called *event channels*. Each domain can be allocated up to 1024 event channels, each of which comprises a pair of bit flags in a memory page shared between the domain and Xen. The first flag is used by Xen to signal that an event is *pending*. When an event becomes pending Xen schedules an asynchronous upcall into the domain; if the domain is blocked then it is moved to the run queue. Unnecessary upcalls are avoided by triggering a notification only when an event first becomes pending: further settings of the flag are then ignored until after it is cleared by the domain.

The second event-channel flag is used by the domain to *mask* the event. No notification is triggered when a masked event becomes pending: no asynchronous upcall occurs and a blocked domain is not woken. By setting the mask before clearing the pending flag, a domain can prevent unnecessary upcalls for partially-handled event sources.

To avoid unbounded reentrancy, a level-triggered interrupt line must be masked at the interrupt controller until all relevant devices have been serviced. After handling an event relating to a level-triggered interrupt, the domain must call *down* into Xen to unmask the interrupt line. However, if an interrupt line is not shared by multiple devices then Xen can usually safely reconfigure it as edge-triggering, obviating the need for unmask downcalls.

When an interrupt line is shared by multiple hardware devices, Xen must delay unmasking the interrupt until a downcall is received from every domain that is managing one of the devices. Xen cannot guarantee perfect isolation of a domain that is allocated a shared interrupt: if the domain never unmasks the interrupt then other domains can be prevented from receiving device notifications. However, shared interrupts are rare in server-class systems which typically contain IRQ-steering and interrupt-controller components with enough pins for every device. The problem of sharing is set to disappear completely with the introduction of message-based interrupts as part of PCI Express [1].

2.3 Device-to-Host Interactions

As well as preventing a device driver from circumventing its isolated environment, we must also protect against possible misbehavior of the hardware itself, whether due to inherent design flaws or misconfiguration by the driver software. The two general types of device-to-host interaction that we must consider are assertion of interrupt lines, and accesses to host memory space.

Protecting against arbitrary interrupt assertion is not a significant issue because, except for shared interrupt lines, each hardware device has its own separately-wired connection to the interrupt controller. Thus it is physically impossible for a device to assert any interrupt line other than the one that is assigned to it. Furthermore, Xen retains full control over configuration of the interrupt controller and so can guard against problems such as 'IRQ storms' that could be caused by repeated cycling of a device's interrupt line.

The main 'protection gap' for devices, then, is that they may attempt to access arbitrary ranges of host memory. For example, although a device driver is prevented from using the CPU to write to a particular page of system memory (perhaps because the page does not belong to the driver), it may instead program its hardware device to perform a DMA to the page. Unfortunately there is no good method for protecting against this problem with current hardware as it is infeasible for Xen to validate the programming of DMA-related device registers. Not only would this require intimate knowledge of every device's DMA engine, it also would not protect against bugs in the hardware itself: buggy hardware would still be able to access arbitrary system memory.

A full implementation of this aspect of our design requires integration of an IOMMU into the PC chipset—a feature that is expected to be included in commodity chipsets in the very near future. Similar to the processor's MMU, this translates the addresses requested by a device into valid host addresses. Inappropriate host addresses are not accessible to the device because no mapping is configured in the IOMMU. In our design, Xen would be responsible for configuring the IOMMU in response to requests from domains. The required validation checks are identical to those required for the processor's MMU; for example, to ensure that the requesting domain owns the page frame, and that it is safe to permit arbitrary modification of its contents.

2.4 Hardware Configuration

The PCI standard defines a generic *configuration space* through which PC hardware devices are detected and configured. Xen restricts each domain's access to this space so that it can read and write registers belonging only to a device that it owns. This serves a dual purpose: not only does it prevent cross-configuration of other domains' devices, but it also restricts the

domain's view so that a hardware probe detects only devices that it is permitted to access.

The method of access to the configuration space is system-dependent, and the most common methods are potentially unsafe (either protected-mode BIOS calls, or a small I/O-port 'window' that is shared amongst all device spaces). Domains are therefore not permitted direct access to the configuration space, but are forced to use a virtualized interface provided by Xen. This has the advantage that Xen can perform arbitrary validation and translation of access requests. For example, Xen disallows any attempt to change the base address of an I/O-register block, as the new location may conflict with other devices.

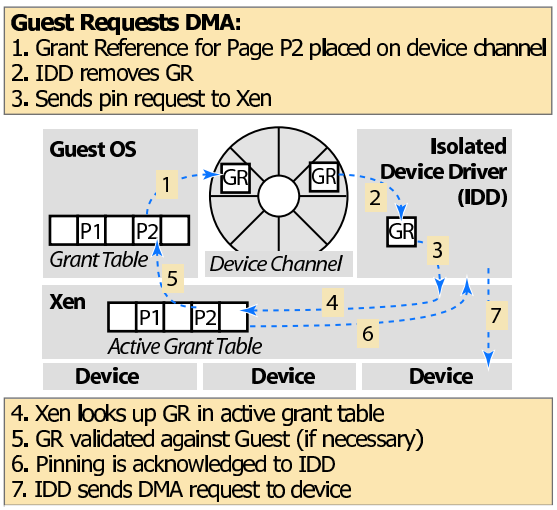

Figure 3: Using device channel to request a data transfer.

2.5 Device Channels

Guest OSs access devices via *device channel* links with isolated driver domains (IDDs). The channel is a point-to-point communication link through which each party can asynchronously send messages to the other. Channels are established by using a privileged *device manager* to introduce an IDD to a guest OS, and vice versa. To facilitate this, the device manager automatically establishes an initial control channel with each domain that it creates. Figure 3 shows a guest OS requesting a data transfer through a device channel. The individual steps involved are discussed later in this section.

Xen itself has no concrete notion of a control or device channel. Messages are communicated via shared memory pages that are allocated by the guest OS but are simultaneously mapped into the address space of the IDD or device manager. For this purpose, Xen permits restricted *sharing* of memory pages between domains.

The sharing mechanism provided by Xen differs from traditional application-level shared memory in two key respects: shared mappings are *asymmetric* and *transitory*. Each page of memory is owned by at most one domain at any time and, with the assistance of Xen and the device manager, that owner may force reclamation of mappings from within other misbehaving domains.

To add a foreign mapping to its address space, a domain must present a valid *grant reference* to Xen in lieu of the page number. A grant reference comprises the identity of the domain that is granting mapping permission, and an index into that domain's private *grant table*. This table contains tuples of the form $(grant, D, P, R, U)$ which permit domain D to map page P into its address space; asserting the boolean flag R restricts D to read-only mappings. The flag U is written by Xen to indicate whether D currently maps P (i.e., whether the grant tuple is *in use*).

When Xen is presented with a grant reference (A, G) by a domain B, it first searches for index G in domain A's *active grant table* (AGT), a table only accessible by Xen. If no match is found, Xen reads the appropriate tuple from domain A's grant table and checks that T=*grant*

and *D=B*, and that *R=false* if *B* is requesting a writable mapping. Only if the validation checks are successful will Xen copy the tuple into the AGT and mark the grant tuple as in use.

Xen tracks grant references by associating a usage count with each AGT entry. When a foreign mapping is created with reference to an existing AGT entry, Xen increments its count. The grant reference cannot be reallocated or reused by the granting domain until the foreign domain destroys all mappings that were created with reference to it.

Although it is clear that this mechanism allows strict checking of foreign mappings when they are created, it is less obvious how these mappings might be revoked. For example, if a faulty IDD stops responding to service requests then guest OSs could end up owning unusable memory pages. We handle the possibility of driver failure by taking a deadline-based approach: if a guest observes that a grant table entry is still marked as in use when it determines that it ought to have been relinquished (e.g., because it requested that the device channel should be destroyed), then it signals a potential domain failure to the device manager.

The device manager checks whether the specified grant reference exists in the notifying domain's AGT and, if so, sets a deadline by which the suspect domain must relinquish the stale mappings. If a registered deadline passes but stale mappings still exist then Xen notifies the device manager. At this point the device manager may choose to destroy and restart the driver, thereby forcibly reclaiming the foreign mappings.

2.6 Descriptor Rings

I/O descriptor rings are used for asynchronous transfers between a guest OS and an IDD. Ring updates are based around two pairs of producer-consumer indexes: the guest OS places service requests onto the ring, advancing a request-producer index, while the IDD removes these requests for handling, advancing an associated request-consumer index. Responses are queued onto the same ring as requests, albeit with the IDD as producer and the guest OS as consumer. A unique identifier on each request/response allows reordering if the IDD so desires.

The guest OS and IDD use a shared *inter-domain* event channel to send asynchronous notifications of queued descriptors. An inter-domain event channel is similar to the interrupt-attached channels described in Section 2.2. The main differences are that notifications are triggered by the domain attached to the opposite end of the channel (rather than Xen), and that the channel is *bidirectional*: each end may independently notify or mask the other.

We decouple the production of requests or responses on a descriptor ring from the notification of the other party. For example, in the case of requests, a guest may enqueue multiple entries before notifying the IDD; in the case of responses, a guest can defer delivery of a notification event by specifying a threshold number of responses. This allows each domain to independently balance its latency and throughput requirements.

2.7 Data Transfer

Although storing I/O data directly within ring descriptors is a suitable approach for low-bandwidth devices, it does not scale to high-performance devices with DMA capabilities. When communicating with this class of device, data buffers are instead allocated out-of-band by the guest OS and indirectly referenced within I/O descriptors.

When programming a DMA transfer directly to or from a hardware device, the IDD must first *pin* the data buffer. We enforce driver isolation by requiring the guest OS to pass a grant reference in lieu of the buffer address: the IDD specifies this grant reference when pinning the buffer. Xen applies the same validation rules to pin requests as it does for address-space mappings. These include ensuring that the memory page belongs to the correct domain, and that it isn't attempting to circumvent memory-management checks (for example, by requesting a device transfer directly into its page tables).

Returning to the example in Figure 3, the guest's data-transfer request includes a grant reference *GR* for a buffer page P_2. The request is dequeued by the IDD which sends a pin request, incorporating GR, to Xen. Xen reads the appropriate tuple from the guest's grant table, checks that P_2 belongs to the guest, and copies the tuple into the AGT. The IDD receives the address of P_2 in the pin response, and then programs the device's DMA engine.

On systems with protection support in the chipset (Section 2.3), pinning would trigger allocation of an entry in the IOMMU. This is the only modification required to enforce safe DMA. Moreover, this modification affects only Xen: the IDDs are unaware of the presence of an IOMMU (in either case pin requests return a bus address through which the device can directly access the guest buffer).

2.8 Device Sharing

Since Xen can simultaneously host many guest OSs it is essential to consider issues arising from device sharing. The control mechanisms for managing device channels naturally support multiple channels to the same IDD. We describe below how our block-device and network IDDs support multiplexing of service requests from different clients.

Within our block-device driver we service *batches* of requests from competing guests in a simple round-robin fashion; these are then passed to a standard elevator scheduler before reaching the disc controller. This balances good throughput with reasonably fair access. We take a similar approach for network transmission, where we implement a credit-based scheduler allowing each device channel to be allocated a bandwidth share of the form *x* bytes every *y* microseconds. When choosing a packet to queue for transmission, we round-robin schedule amongst all the channels that have sufficient credit.

A shared high-performance network-receive path requires careful design because, without demultiplexing packets in hardware [2], it is not possible to DMA directly into a guest-supplied buffer. Instead of copying the packet into a guest buffer after performing demultiplexing, we instead *exchange ownership* of the page containing the packet with an unused page provided by the guest OS. This avoids copying but requires the IDD to queue page-sized buffers at the network interface. When a packet is received, the IDD immediately checks its demultiplexing rules to determine the destination channel—if the guest has no pages queued to receive the packet, it is dropped.

3 Supporting Unmodified OSes

Xen's original goal was to provide fast virtualization, which was achieved by 'paravirtualizing' guest OSes. The downside of paravirtualization is that it requires modification of the guest OS source code—an approach which is untenable for closed-source operating systems.

The alternative, full virtualization of the hardware platform, has traditionally been very difficult on the x86 processor architecture. However, new processor extensions promised by AMD and Intel provide hardware assistance which makes full virtualization much easier to provide.

Preliminary support for Intel Virtualization Technology for x86 processors (VT-x) is already checked into the Xen repository. This provides a 'virtual processor' abstraction to the guest OS which, for example, can transparently notify Xen of any attempt to execute instructions that would modify privileged processor state. While these hardware extensions make transparent virtualization easier, Xen still bears responsibility for device management and enforcing isolation of shared resources such as CPU time and memory.

3.1 VT-x architecture overview

VT-x augments the x86 architecture with two new forms of CPU operation: VMX root operation and VMX non-root operation. Xen runs in VMX root operation, while guests run in VMX non-root operation. Both forms of operation support all four privilege levels (rings 0 through 3), allowing a guest OS to appear to run at its usual 'most privileged' level. VMX root operation is similar to x86 without VT-x. Software running in VMX non-root operation is deprivileged in certain ways, regardless of privilege level.

VT-x defines two new transitions: a *VM entry* that transitions from Xen root operation to guest non-root operation, and a *VM exit* which does the opposite transition. Both VM entries and VM exits load CR3 (the base address of the page-table hierarchy) allowing Xen and the guest to run in different address spaces. VT-x also defines a virtual-machine control structure (VMCS) that manages VM entries and exits and defines processor behavior during non-root execution.

Processor behavior changes substantially in VMX non-root operation. Most importantly, many instructions and events cause VM exits. Some instructions cannot be executed in VMX non-root operation because they cause VM exits unconditionally; these include CPUID, MOV from CR3, RDMSR, and WRMSR. Other instructions, interrupts, and exceptions can be configured to cause VM exits conditionally, using VM-execution control fields in the VMCS.

VM entry loads processor state from the guest-state area of the VMCS. Xen can optionally configure VM entry to inject an interrupt or exception. The CPU effects this injection using the guest IDT, just as if the injected event had occurred immediately after VM entry. This feature removes the need for Xen to emulate delivery of these events. VM exits save processor state into the guest-state area and load processor state from the host-state area. All VM exits use a common entry point into Xen. To simplify the design of Xen, every VM exit saves into the VMCS detailed information specifying the reason for the exit; many exits also record an exit qualification, which provides further details.

3.2 VT-x Support in Xen

The three major components for adding support of VT-x and running unmodified OS in Xen are:

1. Extensions to the Xen hypervisor
2. Device models that emulate the PC platform
3. Administrator control panel

The hypervisor extensions involve adding support for the specific hardware features and instruction opcodes added by VT-x, and extensions to the user-space control tools for building and controlling fully-virtualized guests. Device models provide emulation of the PC platform devices for a VMX domain. The software models emulate all the hardware-level programming interfaces that a normal device driver uses to perform I/O operation, and submit requests to a physical device on the VMX domain's behalf.

QEMU and Bochs are two open source PC platform emulators that provided most of the functionality we needed for I/O emulation for VMX domains. Our basic design has been to run the device models in domain 0 user space and run one process for each VMX domain. We needed to remove all CPU emulation code from these emulators and modify the code that emulated physical memory (RAM). Normally, they allocate a large array to emulate the physical memory. We modified the code to map all the physical memory allocated to the VMX domain.

An example of I/O request handling from VMX guest is as follows:

1. VM exit due to an I/O access.
2. Xen decodes the instruction.
3. Xen constructs an I/O request describing the event.
4. Xen sends the request to the device-model process in domain 0.
5. When reading from a device register, the VMX domain is blocked until a response is received from the device model.

4 New Architectures

Xen was originally designed and implemented to support the x86 architecture. As interest in Xen has increased, several organisations have expressed interest in using Xen as a common hypervisor for other hardware platforms. The last year has seen fervent development of architectural support for both x86/64 and IA-64.

In addition to x86/64 and IA-64, ports of Xen are underway to the IBM Power platform and to both of the upcoming fully virtualized versions of x86, Intel's VT (described in the previous section) and AMD's Pacifica/SVM. Our experience with IA-64 supports our belief that Xen will successfully accommodate these new architectures and any others that come along in the future.

4.1 x86/64

When extending Xen to support the x86/64 architecure, we kept in mind that the platform is largely identical to x86/32, differing only in some of the details of the processor architecture. For example, processor registers are extended to 64 bits and the page-table format is extended to support the larger address space. Fortunately, the hardware platfrom is largely identical: for example, sharing the same I/O bus and chipset implementations.

This led us to implement x86/64 support as a sub-architecture of the existing x86 target. Large swathes of code are shared between sub-architectures, with the main differences being in assembly-code stubs and page-table management.

From a guest perspective, the most interesting change presented by x86/64 is the modified protection model. x86/64 provides very limited segment-level protection which makes

it impossible to protect the hypervisor, running in ring 0, from a guest kernel running in ring 1. This architectural change necessitates running both the guest kernel and applications in ring 3, and raises the problem of protecting one from the other.

The solution is to run the guest kernel in a different address space (i.e., on different page tables) from its applications. When forking a new process, the guest kernel creates two new page tables: one that is used in application context, and the other in kernel context. The kernel page table contain all the same mappings as the application page table but also include a mapping of the kernel address space. All transitions between application and guest-kernel contexts must pass via Xen, which automatically switches between the two page tables.

4.2 IA-64

As of this writing, Xen/ia64 is between its alpha release and beta release. All basic hypervisor capability is present: Domain 0 runs solidly as a 'demoted' guest OS, utilizing all devices while booting to a full graphical console and executing all Linux/ia64 applications unchanged. Multiple guest domains can be launched, but virtual I/O functionality is not finished so any unprivileged domain boots to the point where init fails to find a root disk, then panics and reboots in an infinite cycle. SMP support is not yet present, either in the hypervisor itself or in the guest.

Full functionality in Xen/ia64 is expected later this year, but the port is sufficiently complete to illustrate some similarities and differences that establish credibility that Xen will prove widely portable:

1. Hardware-walked page tables must be carefully managed in Xen/x86 and, indeed, handling page tables is one of the most complex parts of Xen, requiring a fair amount of code in the hypervisor and non-trivial changes in the paravirtualization of guests. On ia64, hardware page-table walking is still necessary for performance, but can be much more easily diverted to hypervisor-managed page tables. The difference is completely hidden from common code and implemented in the arch-specific layer.

2. Like the x86 architecture, ia64 is not fully virtualizable—certain instructions have different results when executed at different privilege levels. Both Xen architectures 'demote' the guest OS and provide an interface to handle these privilege-sensitive operations.

3. While the x86 has a small state vector, the ia64 architecture has well over 500 registers and two stacks that must be carefully managed for each thread. Linux solves this elegantly with multiple state staging areas and lazy save/restore to optimize kernel entry and exit and thread switching. Recognizing the similarity between Linux threads and Xen domains allows most of the Linux code to be directly reusable.

4. The page size on x86 is 4kB. Modern versions of x86 chips support a larger page size, but its use is limited. IA-64 supports nine different page sizes and a guest OS may use all of them simultaneously. Thus, Xen/ia64 must manage this additional complexity. Again, this is safely hidden in Xen through asm macros and arch-specific modules.

5 Conclusion

In this paper, we have presented a brief overview of the major new features in Xen 3.0 including VM relocation, device driver isolation, support for unmodified operating systems, and new hardware support for both x86/64 and IA-64 processors. Xen is quickly maturing into an enterprise-class VMM and is currently being used in production environments around the globe.

References

[1] PCI Express base specification 1.0a. PCI-SIG, 2002.

[2] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *Proceedings of IEEE INFOCOM-01*, pages 67–76, April 2001.

Examining Linux 2.6 Page-Cache Performance

Sonny Rao, Dominique Heger, Steven Pratt
IBM Corporation
{sonnyrao, dheger, slpratt}@us.ibm.com

Abstract

Given the current trends towards ubiquitous 64-bit server/desktop computing with large amounts of cheap system memory, the performance and structure of the Linux® page-cache will undoubtedly become more important in the future. An empirical and analytical examination of performance will be valuable in guiding future development.

The current 2.6 radix-tree based design represents a huge leap forward from the old global hash-table design, but there may be some issues with the current radix-tree structure itself.

The main goal is to understand performance of the current implementation, examine performance with respect to other potential data-structures, and look at ways to improve concurrency.

1 The Radix-Tree based Page Cache in Linux 2.6

The Linux 2.6 page cache is basically a collection of pages that normally belong to files. The pages are kept in memory for performance reasons. As on other UNIX® operating systems, the page cache may take up the majority of the available memory. Whenever a thread reads from or writes to a file, takes a page fault, or is paged out, the page cache becomes involved. Hence, the performance of the page cache has a rather dramatic impact on the performance of the system. As a particular page is referenced, the page cache has to be able to locate the page, or has to determine that the page is not in the cache, in as efficient and effective way as possible with a focus on minimal memory overhead.

1.1 Evolution of the Page Cache

In older versions the Linux kernel utilized a global hash-table based approach to maintain the pages in the cache. The hash based approach had some performance issues:

1. A hash key is normally not unique; hence the system has to resolve collisions. A hash chain had to be built to hold entries (each entry used up 8 bytes per referenced page).
2. A single global lock governed the page cache; causing scalability issues on SMP based systems.

The radix-tree based page cache solution addresses the issues discussed above.

Technically, the Linux 2.6 system consists of many smaller page cache subsystems, or more

specifically, one for each open file in the system.

Segregating page caches has a few advantages:

First, each page cache can have its own lock, avoiding the global page cache lock that was necessary in older versions.

Second, search operations work on a smaller address space, and complete more quickly.

Third, as there is one page cache per open file, the only index required to look up a specific page is the offset within the file.

In the radix-tree, the 32-bit or 64-bit file offset is divided into subsets whose size is based on the value of `MAP_SHIFT` as defined in lib/radix.c. The current implementation uses a `MAP_SHIFT` of six for 6-bit indices. The highest-order sub-field (or set) is used to branch into a 64-entry table in the root of the radix-tree. An entry in that sub-table serves as a pointer to the next node in the tree. The next lower sub-field (from the index) is used to index that particular node, yielding a third abstraction. Eventually, the algorithm will reach the bottom of the tree and obtain the actual page pointer or finds an empty entry, signifiying that the page is not present. Table 2 shows maximum file offset and number of pages versus tree height for the shift value of six.

There is some precedent for using a value other than six for the `MAP_SHIFT`. Originally, seven was used for the `MAP_SHIFT` when the structure was first introduced [7]. Larger values mean smaller trees in terms of height and the possiblity of shorter search times. This possibility comes at the expense of bigger nodes in the slab cache, which means that there is more potential for wasted entries.

Shift	through-put	delta	profile	delta
6	4.61	N/A	13.21	N/A
8	4.745	(+3)%	12.09	(–8.7)%
10	4.705	(+2)%	12.40	(–6.14)%
12	4.695	(+1.8)%	12.31	(–6.72)%
4	4.683	(+1.6)%	17.22	(+30.4)%

Table 1. Sequential read throughput and percent of profile ticks for `radix_tree_lookup` results for different values of `MAP_SHIFT`. The units for throughput are GB/sec, and the profile column represents time spent in `radix_tree_lookup`. These values represent the average of four runs.

height	maximum pages	maximum file offset
0	0	0
1	64	256 KB
2	4096	16 MB
3	262144	1 GB
4	16777216	64 GB
5	1073741823	4 TB
6	4294967296	16 TB

Table 2. Max number of pages by radix-tree height with a 32-bit key and `MAP_SHIFT` of 6, file offset assumes 4k pages

One optimization criterion was to ensure that the radix-tree would only be as deep as necessary. In the case where the system operates on small files (smaller than 65 pages), only one level of abstraction (one node) will be used. In other words, only the least significant sub-field of the offset is being utilized. This property of the current implementation allows the normally detrimental effects of a large key on a radix-tree to be minimized. The only potential downside is in the case of a sparse file where nodes located at relatively large offsets will force a higher tree depth than might otherwise be necessary.

1.2 Newer Features in 2.6

One of the newer features incorporated in Linux revolves around 'tagging' dirty pages in the radix-tree. In other words, a dirty page is only flagged in the radix-tree, and not moved to a separate list as in the pre 2.6.6 design. Along the same lines, pages that are being written back to disk are flagged as well. A new set of radix-tree functions was implemented to locate these pages as necessary. Searching an entire tree structure for these pages is not as efficient as just traversing trough a dedicated list, but based on the feedback from the Linux community, the performance delta is not considered a big issue. There is some concern in the Linux community that with very large files the 2.6 lock-per-file based approach will be as bad as the global lock based 2.4 implementation. The tagging of these pages in the new design required a lot of changes to the page cache and the VM subsystems, respectively. One implication of the changes is that the dirty pages are now always written in file offset order out to disk. As the Linux community reports, this may cause some performance issues involving parallel write() operations on large SMP systems. The tagging of all these pages in the radix-tree contributes to the complexity of switching from a radix-tree based approach to another data structure (if needed). Based on the current implementation, improving the radix-tree seems more feasible than a complete re-design and should therefore be explored first. The `MAP_SHIFT` parameters in the radix code reveal some potential for performance work.

There is a scalability issue when dealing with only a small amount of very large files and a workload that consists of many concurrent read operations on the files. The single lock governing the radix-tree will basically eliminate any potential scalability on SMP systems while exposed to such a workload. Scalability of course is achieved when the workload consists of n worker threads reading from n separate files, hence the locking is distributed over the set of files being accessed. Table 3 shows the severity of the locking problems of the current spinlock design vs the rwlock design and shows that even the rwlock implementation spends a good deal of time overall CPU time in locking functions.

Table 3 shows throughput on an IBM p650 8-CPU POWER4+ server with 16GB of RAM and two 7GB files fully cached with differing numbers of threads attempting to sequentially read the files. Throughput is in GB/sec and the profile columns show the percentage of time from the profile spent in locking functions.

Threads	Spinlock	Profile	Rwlock	Profile
1	1.11	0.10%	1.04	0.80%
4	2.26	12.4%	2.47	4.33%
8	2.01	54.1%	2.82	9.75%
12	2.20	51.6%	2.98	9.86%
14	2.31	49.3%	3.03	9.74%
16	2.34	48.9%	3.13	9.52%

Table 3. Read throughput and time spent in lock functions for spinlock and rwlock kernels.

There has been some ongoing debate over whether a rwlock solution would be more acceptable, however as of this writing it has been held out of mainline due to specific concerns over the performance of the rwlock solution on Pentium-4 machines [9, 10]. Although the cost of locking is substantial on all architectures, this architecture seems to exhibit particularly high latency on the unlock operation. This also seems to indicate that the radix-nodes tend to be cached and that search times are small [8].

2 Alternative Data-Structures

Given the unique nature of the radix implementation in the Linux kernel, comparative analysis

of the radix-tree with alternative data-structures should provide insights into its strengths and weaknesses. In general, for the application of page-cache lookup, speed should be paramount since in the case of a cache hit, the entire read or write operation should occur at memory speed. Inserts, on the other hand, will typically be followed by disk I/O, and that I/O should become the limiting factor for the operation rather than the cache update. Deletes are initiated from a truncate operation or by the page-scanner when the system is under memory pressure. This case of memory pressure is performance critical since the VM wants to release the pages selected as soon as possible, and updates to the caching structures represent pure overhead. Operations such as "tagging" pages as dirty are also interesting because they involve both a lookup and a modification to the state of the data structure. However this operation is specific to the Linux 2.6 radix-tree implementation and is not available on all data-structures. In some cases it may be possible to graft these additional pieces of state information onto other standard data-structures, but it is not practical in all cases.

Given these qualities, it seemed appropriate to test the Linux kernel implementation of radix-trees against a number of other data-structures each with slightly different design trade-offs.

2.1 Extendible-Hashing

One idea suggested was that of extendible-hashing, which is a technique developed for optimizing lookup operations in database systems [6]. Among other interesting properties, extendible hashing guarantees that data can be accessed in just two "page-faults" in database terminology, which translates to two pointer dereferences for our purposes. As the name suggests, it is capable of extending itself as the amount of data stored increases, and it can do this without costly re-hashing of the entire data-set. Conversely, the extendible hash-table can be implemented to contract itself as elements are removed. Naturally, these characteristics are not free and represent a trade-off for the fixed number of memory dereferences in the lookup path.

The extendible hash-table typically is implemented using two structures: buckets, which contain the pointers to the data, and a directory, which contains the pointers to the buckets. The directory is just a large array with a power-of-two size. The logarithm of the current size is called the directory depth.

Elements are inserted by computing a hash key and taking the n most-significant bits of that key, where n is equal to the directory depth. Using this value to index into the directory yields a pointer to the bucket where the new element will reside. Different strategies exist for placing an element into a bucket. Depending on the size of the bucket, the object's hash value can be used again to place the item, or if the bucket is fairly small, a simple linear insert can be effective.

Each pointer in the directory is not necessarily unique, and there can be mulitple pointers to a certain bucket. For this reason, the buckets keep a local-depth value, which can be used to compute the number of pointers to it in the directory. When a bucket becomes full, it must be split into two separate buckets in an operation called a bucket-split. After the bucket-split, each new bucket will get half of the old pointers in the directory, and the local depth of the buckets will increase by one. If the bucket has a local depth equal to the directory depth, then the directory must be first doubled in size before the bucket can be split. In this case, there is only one pointer in the directory to this particular bucket before the directory doubling operation, and afterwards there are two pointers and the bucket-split can proceed. When a

bucket-split occurs, the elements in the original bucket are redistributed into the new buckets in such a way that their hash-keys will lead to the correct bucket from the directory. In this way, the extendible hash-table avoids having to ever globally re-hash and instead limits redistribution to bucket-splits while retaining the original hash function.

One additional characteristic of the extendible-hashing is its ability to handle random sequences of keys equally as well as sequential sequences. Though many typical applications will primarily use sequential I/O patterns, some applications might find this characteristic beneficial.

2.2 Threaded Red-Black Tree

Threaded red-black trees are a twist on the notion of a traditional red-black tree, which try to optimize for sequential access sequences by using normally NULL leaf pointers as "threads" which keep track of nodes with neighboring keys [12]. So, if one already has a reference to a particular leaf node, access to the previous node (in terms of key order) only requires accessing that node's left "thread."

The regular red-black tree properties still apply [1,2], but since almost all child pointers are used in some way, additional state information must be kept in the nodes to differentiate children from threads. Luckily, red-black trees, such as the one in the Linux kernel, already use an extra word per node to keep track of color. This extra word can be overloaded to keep track of thread information as well with no additional space cost.

Since one cannot simply test for NULL during lookups, one must also alter any open-coded lookup sequences to be thread-aware, which is to say such code must examine the state information in the node. Ideally, this should not be a significant cost because the flags should typically have reasonable spatial locality with the other pointers in the node and would be kept in the same cache-line.

As with regular red-black trees, performance of inserts and deletes is traded off to keep the tree balanced and keep average lookup times down. In the case of the threaded version this is even more true as thread information must be kept consistent through rebalancing operations.

The implementation tested was similar to the Linux kernel's present red-black tree implementation which assumes the node contents are embedded into another object and passes off responsibility for memory allocation and implementing lookups onto the tree's user.

2.3 Treap

A treap is the basic data structure (BST) underlying randomized search trees [3]. The name itself refers to the synthesis of a tree and a heap structure. More specifically, a treap represents a set of items where each item has associated with it a key and a priority. In general a priority is randomly assigned to a given key by the implementation. A treap represents a special case of a binary search tree, in which the node set is arranged in order (with respect to the keys) as well as in heap fashion with regards to the priority. The procedure for lookup in a treap is the same as for a normal binary search-tree and the node priorities are simply ignored. In a treap, the access time is proportional to the depth of an element in the tree. An insert of a new item basically consists of a two step process. The first step consists of utilizing the item's key to attach to the treap at the appropriate leaf position, and second to use the priority of the new element to rotate the new entry up in the structure until the item locates the parent node that has a larger priority. Interestingly, it can be shown in the general case that

an insert will only cause two rotations, which means updates are much less costly then in the case of a strictly balanced tree such as an AVL tree or red-black tree.

The implementation tested used a simple polynomial hash function on the key to generate the priority. This approach was used instead of the kernel's random number generator to keep the implementation as self-contained as possible.

Again, the implementation tested follows the Linux kernel's convention of assuming the user must allocate the nodes and open-code the lookup sequences.

2.4 Linux Radix-Tree

The Linux implementation of the radix-tree is highly optimized and customized for use in the kernel and differs signifcantly from what is commonly referred to as a radix-tree [1,4,5] It avoids paying the memory cost of explictly keeping keys, child-pointers, and separate data-pointers on each object but instead uses implicit ordering along with node height to determine the meaning of these pointers. For example, if the tree has a global height of three, then the pointers on the first two levels only point to child nodes and the lowest level uses its pointers for data objects. Data pointers only exist at the lowest level.

By aggressively conserving memory and reducing the tree's overall size, the radix-tree has an extremely small cache footprint which is vital to its success at larger tree sizes.

The main disadvantage of using implicit ordering in the implementation is that a highly sparse file will force the use of more tree-levels across the entire tree for all offsets. The current implementation uses a `MAP_SHIFT` of six which means sixty-four pointers per node, and in the worst case all but one of those pointers is wasted from the root all the way down to the leaf. The height is directly related to the offset of the last object inserted into the tree.

The kernel implementation also supports tagging which means each node not only consists of an array of pointers but a set of bit-fields for each pointer which can be used somewhat arbitrarily by the subsystem utilizing the tree. In the case of the page-cache, these tags are used to keep track of whether a page is dirty or undergoing writeback.

The meaning of these tags is clear at the leaf nodes, but at higher levels, tags are used to refer to the state of any objects in or below the child node at the corresponding offset.

For example, given a three level radix-tree, and the page at offset one is dirty, then the dirty-tag for bit one on the leaf node is set and the tags for bit zero are set in the two nodes above. This way, gang-lookups searching for tagged nodes can be optimized to skip over subtrees without any tagged descendants.

2.5 Analysis

In all three operations tested, there was no significant difference between the data structures until roughly 128K elements where the differences begin and are highlighted by the remaining data points.

The extendible-hashing results were initially very surprising as it seems to perform much worse than the tree structures at high object counts. After analyzing performance counter information, it was determined that the extremely poor spatial and temporal locality of the the hash directory and buckets were causing TLB and similar translation cache misses and thus large amounts of time were spent doing

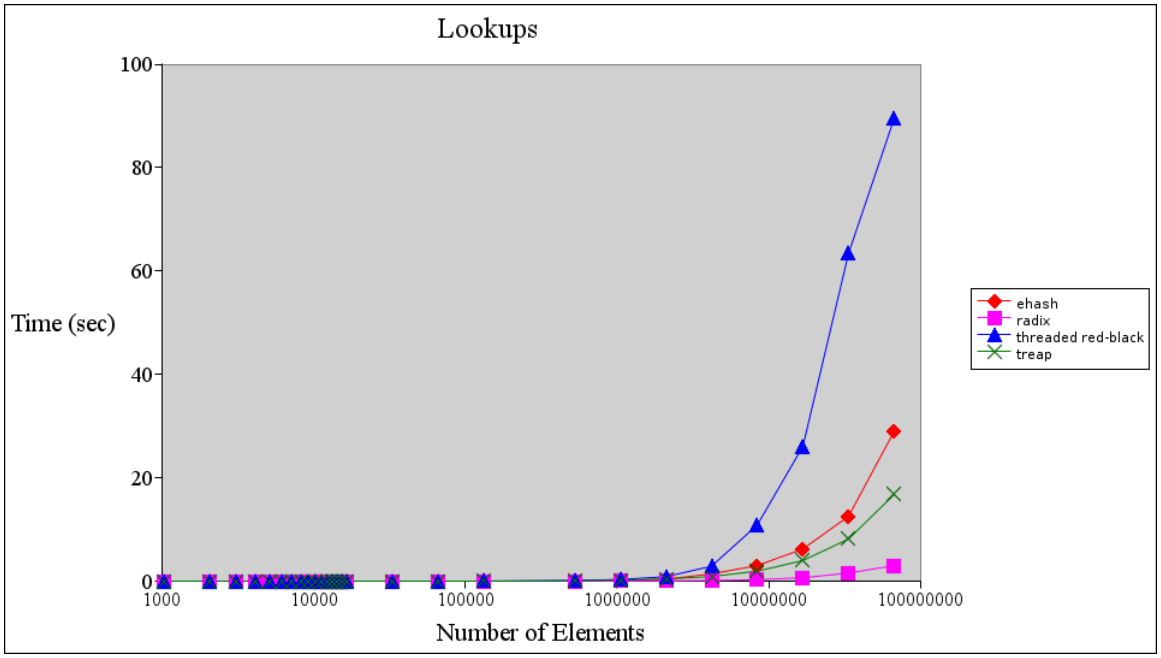

Figure 1: Sequential Lookup Performance

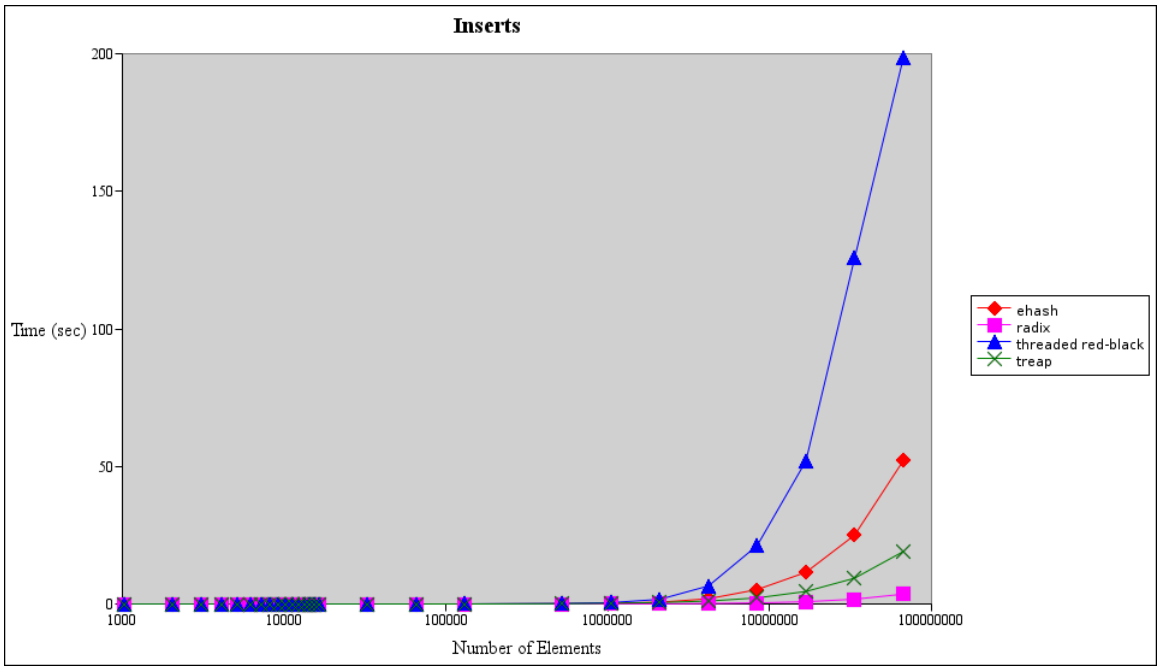

Figure 2: Sequential Insert Performance

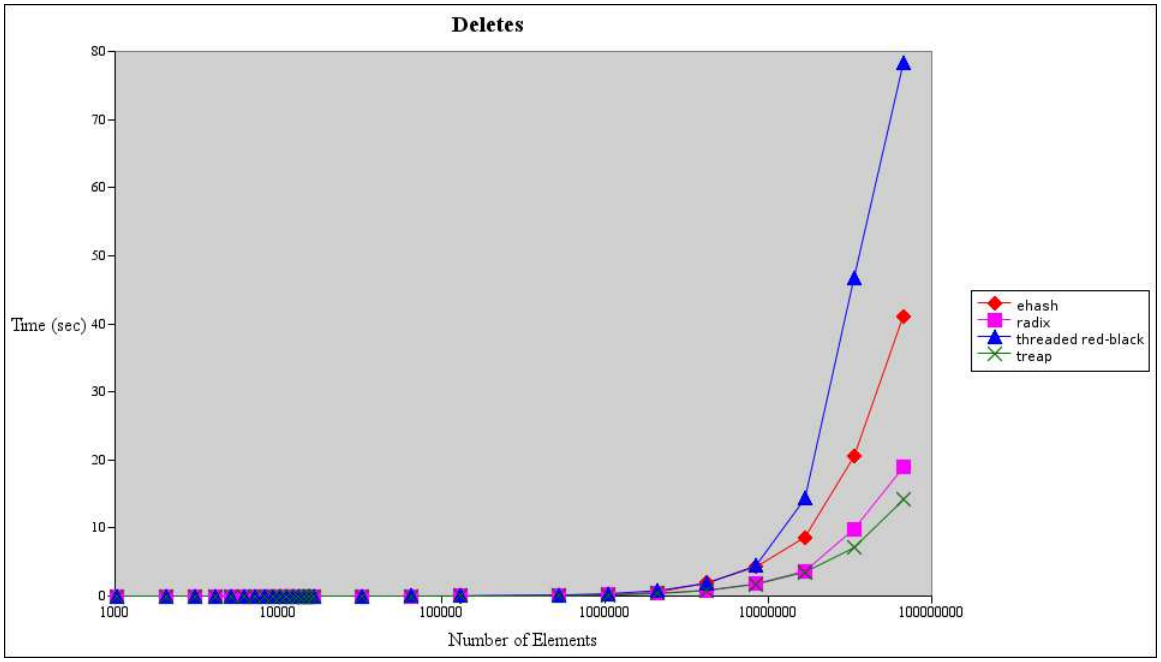

Figure 3: Sequential Delete Performance

page-table walk operations. Also, the poor spatial locality caused a great deal of data-cache misses which compounded the problems. On the other hand, for the tree structures, the sequential nature of the test yielded significant benefits to their cache interactions.

The two binary-tree structures offer mixed performance with generally worse performance on lookups and inserts with only the treap narrowly beating the radix-tree in deletes. The threaded red-black tree also seems to do worse than expected in lookups which will require some further analysis.

The radix-tree scales extremely well into the very large numbers of pages because the tree itself fits into processor caches much better than the alternative designs. In the case of the delete operations the radix-tree still does well, but is beaten in some cases by the treap. Most likely, this is because of the extensive updates which must occur to the tagging information up the tree which typically would not have good spatial or temporal locality with respect to the initial lookup.

This result has also been observed under a 'real' data-base workload where the `radix_tree_delete` call shows up higher in kernel profiles than the `radix_tree_lookup` operations, which was initially rather confusing, as it was expected that most of the time in the radix-tree code would be spent doing lookup operations. Table 4 shows this effect, where `radix_tree_delete` shows up as the third highest kernel function and `radix_tree_lookup` is number ten. Overall, this particular database query is heavily I/O bound, as `dedicated_idle` represents time spent waiting on I/O to complete, and the rest of the functions indicate memory pressure (`shrink_list`, `shrink_cache`, `refill_inactive_zone`, and `radix_tree_delete`) and other filesystem activity (`find_get_block`).

DB Workload: Top 10 Kernel Functions

dedicated_idle
__copy_tofrom_user
radix_tree_delete
_spin_lock_irq
__find_get_block
shrink_list
refill_inactive_zone
__might_sleep
shrink_cache
radix_tree_lookup

Table 4: Kernel functions reported by OProfile from a standard commercial database benchmark which simulates a business decision support workload. The tests were run on IBM OpenPower 720 4-CPU machine running on Ext3 with 92% of time spent in the kernel for this query. Other querys in the workload showed similar results where in all cases `radix_tree_delete` was ordered higher than `radix_tree_lookup`.

2.6 Continuing Work

In the interests of time, all of these results were collected in userspace. As time permits, the tests can be re-done using kernel-space implementations to keep user-space biases to a minium and to avoid any bias due to the memory allocator.

These tests also represent best-case cache-behavior, because actual data pages were not being moved through the memory sub-system. Again, these structures should be re-examined in the future with a mixed workload with sub-optimal caching behaviors.

3 Going Forward with Improvements to the Page-Cache

As far as improving the radix-tree, there does not appear to be any reason to outright replace the current implementation, however performance could probably be improved for the class of workloads desiring concurrent access to the radix-tree structure by improving the locking behaviors for the radix-tree. As an example, a database system using large files for storing tables and using the page-cache could run into this issue.

3.1 A Lockless Design

Ultimately, it would be beneficial to implement a fully lockless design (for readers) using a Read-Copy-Update (RCU) approach [11]. This would allow the tree to better scale with many concurrent readers, and should not cause any difference in performance for a writers. This could cause a number of issues and race-conditions where readers seeing “stale” data could cause problems, and these issues must be

fully explored and understood before an implementation can be attempted.

Of the data-structures mentioned above, the radix-tree and the extendible hash-table would be the best structures suited for a lockless design, while the binary-tree structures are somewhat more difficult to modify for RCU.

In the case of the extendible hash-table, there are two cases to consider: bucket-splits and directory-expansion. In the case of bucket-splits, two new buckets are typically allocated to replace the original, so the original could be left in place for other readers, while the writing thread copied the data from the original bucket to the new ones and then updated the pointers in the directory. The race between readers looking at the directory and seeing the original bucket and seeing one of the new buckets is not problematic, since in either case, the appropriate data will be in whichever bucket is seen. The release of the memory for the old bucket would simply have to wait until all processors had reached a quiescent state. In the case of the directory expansion (or contraction) a similar technique would apply, where the writing thread works to update the new directory while leaving the old one in place. Then it can update the pointer to the directory after it finishes and use a deferred release for the old directory.

For the radix-tree, the main update case is radix-tree extension, where a new offset is inserted which requires an increase in the height of the tree. Luckily, the radix-tree is fairly simple and does not require complex restructuring in this case, but instead merely adds new levels ontop of the exisiting tree. So, in this case the writer thread creates these new nodes and sets them up while letting concurrent readers see the pre-exisiting tree, then when all of the new radix-nodes are set up, the height of the tree can be incremented and a new root installed. There is one problem with doing this today, the radix-tree root object currently consists of three fields including the height and a pointer to the root. For the RCU design to work, it must be able to atomically update a single field for the readers to look at, however both the height and the root pointer require updates. The solution to this is to add another level of indirection and simply keep that information in a separate dynamic object.

3.2 An Evolutionary Improvement

An alternative approach using gang-lookups, which is more evolutionary with respect to the current locking design, was suggested by Suparna Bhattacharya[1].

The current locking design works one page at a time where the radix-tree lock is acquired and released for each page locked. This is one reason why the rwlock approach may not be faster, since it uses an atomic operation both on acquisition and release whereas a spin-lock only uses one atomic operation on a successful lock acquisition. Her suggestion was to instead use a gang-lookup and lock each page requested one after the other before releasing the tree-lock. This approach would drastically reduce the number of costly atomic operations. This would come at the cost of increased lock hold times for the tree, but this could be mitigated somewhat by going back to the rwlock approach. Further, in this case the rwlock becomes a more effective solution since the number of unlock operations is drastically reduced.

method	spin-lock	rwlock
page by page	2n	3n
gang-lookup	n + 1	n + 2

[1] This idea was suggested in a private email to the authors, where she is working on converting the write-path to do something similar

Table 5. Table showing number of atomic operations required to lock n pages for the different locking strategies.

4 Summary

Overall, the performance of the current Linux 2.6 radix-tree is quite good as compared to the other data-structures chosen. Probably the area which is most ripe for improvement is the locking strategy for the radix-tree. A few different alternatives have been suggested, and hopefully by using these or other approaches, page-cache performance can be improved so that it even better than it is today.

5 Legal

6 References

[1] Cormen, T., *Algorithms* , Second Edition, MIT Press, 2001.

[2] Wirth, N., *Algorithms + Data Structures = Programs*, Prentice-Hall.

[3] Seidel, R., Aragon, C., *Randomized Search Trees*, Algorithmica 16, 1996.

[4] Andersson, A., Nielsson, S., *A New Efficient Radix Sort*, FOCS, 1994.

[5] Weiss, M., *Data Structures and C Programs* Addison-Wesley, 1997.

[6] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. *Extendible hashing—a fast access method for dynamic files*, September 1979, ACM Transactions on Database Systems, 4(3):315–344.

[7] Hellwig, C. *[PATCH] Radix-tree pagecache for 2.5*, January 2001, `http://www.ussg.iu.edu/hypermail/linux/kernel/0201.3/1234.html`

[8] Morton, A. *2.5.67-mm1*, April 2003, `http://www.uwsg.iu.edu/hypermail/linux/kernel/0304.1/0049.html`.

[9] Morton, A. *Re: 67-mjb2 vs 68-mjb1 (sdet degredation)*, April 2003, `http://www.cs.helsinki.fi/linux/linux-kernel/2003-16/0426.html`.

[10] Morton, A. *Re: [PATCH] Fixing address space lock contention in 2.6.11*, March 2005, `http://www.ussg.iu.edu/hypermail/linux/kernel/0503.0/1098.html`.

[11] McKenney, P., Appavoo, J., et al. *Read-Copy Update*, July 2001, Ottawa Linux Symposium.

[12] Pfaff, B. *GNU Libavl 2.0.2 Documentation* `http://www.stanford.edu/~blp/avl/libavl.html/`.

Trusted Computing and Linux

Kylene Hall
IBM
kylene@us.ibm.com

Tom Lendacky
IBM
toml@us.ibm.com

Emily Ratliff
IBM
emilyr@us.ibm.com

Kent Yoder
IBM
yoder1@us.ibm.com

Abstract

While Trusted Computing and Linux® may seem antithetical on the surface, Linux users can benefit from the security features, including system integrity and key confidentiality, provided by Trusted Computing. The purpose of this paper is to discuss the work that has been done to enable Linux users to make use of their Trusted Platform Module (TPM) in a non-evil manner. The paper describes the individual software components that are required to enable the use of the TPM, including the TPM device driver and TrouSerS, the Trusted Software Stack, and TPM management. Key concerns with Trusted Computing are highlighted along with what the Trusted Computing Group has done and what individual TPM owners can do to mitigate these concerns. Example beneficial uses for individuals and enterprises are discussed including eCryptfs and GnuPG usage of the TPM. There is a tremendous opportunity for enhanced security through enabling projects to use the TPM so there is a discussion on the most promising avenues.

1 Introduction

The Trusted Computing Group (TCG) released the first set of hardware and software specifications shortly after the creation of that group in 2003.[1] This year, a short two years later, 20 million computers will be sold containing a Trusted Platform Module (TPM) [Mohamed], which will largely go unused. Despite the controversy surrounding abuses potentially enabled by the TPM, Linux has the opportunity to build controls into the enablement of the Trusted Computing technology to help the end user control the TPM and take advantage of security gains that can be made by exercising the TPM properly. This paper will cover the pieces needed for a Linux user to begin to make use of the TPM.

This paper is organized into sections covering the goals of Trusted Computing, a brief introduction to Trusted Computing, the components required to make an operating system a trusted operating system from the TCG perspective, the current state of Trusted Computing, uses of the TPM, clarification of common technical misperceptions, and finally concludes with

[1] See [Fisher] and [TCGFAQ] for more history of the Trusted Computing Group.

a section on future work.

2 Goals of Trusted Computing

The Trusted Computing Group (TCG) has created the Trusted Computing specifications in response to growing security problems in the technology field.

"The purpose of TCG is to develop, define, and promote open, vendor-neutral industry specifications for trusted computing. These include hardware building block and software interface specifications across multiple platforms and operating environments. Implementation of these specifications will help manage data and digital identities more securely, protecting them from external software attack and physical theft. TCG specifications can also provide capabilities that can be used for more secure remote access by the user and enable the user's system to be used as a security token."[TCGBackground]

Fundamentally, the goal of the Trusted Computing Group's specifications is to increase assurance of trust by adding a level of verifiability beyond what is provided by the operating system. This does not reduce the requirement for a secure operating system.

3 Introduction to Trusted Computing

The Trusted Computing Group (TCG) has released specifications about the Trusted Platform Module (TPM), which is a "smartcard-like device," one per platform, typically realized in hardware that has a small amount of both volatile and non-volatile storage and cryptographic execution engines. Figure 1 shows

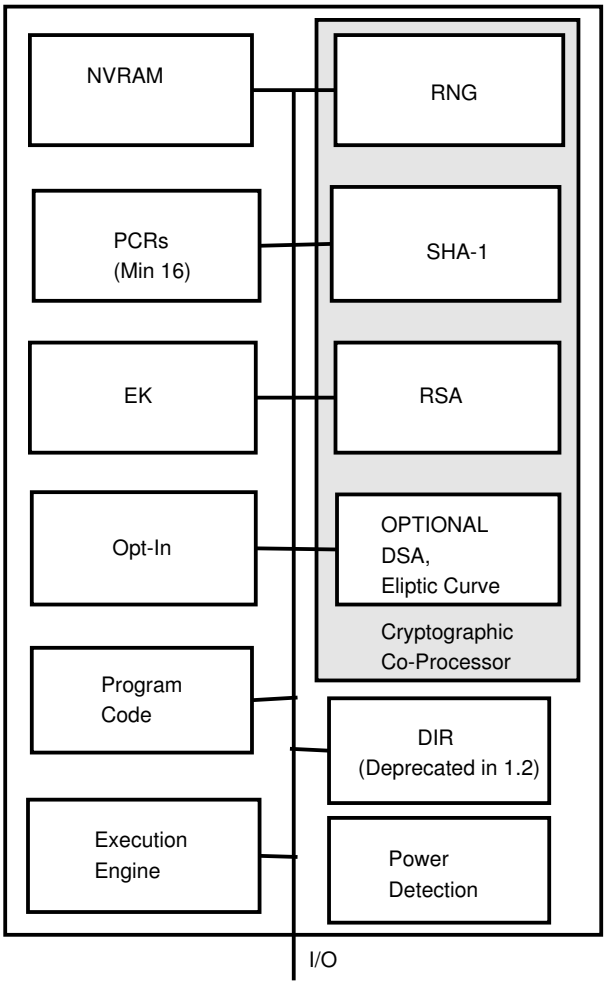

Figure 1: Trusted Platform Module

a logical view of a TPM. The TCG has also released a specification for APIs to allow programs to interact with the TPM. The next section details the components needed to create a completely enabled operating system. The interaction between the components is graphically shown in Figure 2.

For a rigorous treatment of Trusted Computing and how it compares to other hardware security designs, please read Sean W. Smith's "Trusted Computing Platforms Design and Applications" [Smith:2005].

3.1 Key Concepts

There are a few key concepts that are essential to understanding the Trusted Computing specifications.

3.1.1 Measurement

A measurement is a SHA-1 hash that is then stored in a Platform Configuration Register (PCR) within the TPM. Storing a value in a PCR can only be done through what is known as an extend operation. The extend operation takes the SHA-1 hash currently stored in the PCR, concatenates the new SHA-1 value to it, and performs a SHA-1 hash on that concatenated string. The resulting value is then stored in the PCR.

3.1.2 Roots of Trust

In the Trusted Computing Group's model, trusting the operating system is replaced by trusting the roots of trust. There are three roots of trust:

- root of trust for measurement
- root of trust for storage
- root of trust for reporting

The root of trust for measurement is the code that represents the "bottom turtle"[2]. The root of trust for measurement is not itself measured; it is expected to be very simple and immutable. It is the foundation of the chain of trust. It performs an initial PCR extend and then the performs the first measurement.

[2]This is an allusion to the folk knowledge of how the universe is supported. `http://en.wikipedia.org/wiki/Turtles_all_the_way_down`

The root of trust for storage is the area where the keys and platform measurements are stored. It is trusted to prevent tampering with this data.

The root of trust for reporting is the mechanism by which the measurements are reliably conveyed out of the root of trust for storage. This is the execution engine on the TPM.[TCGArch]

3.1.3 Chain of Trust

The chain of trust is a concept used by trusted computing that encompasses the idea that no code other than the root of trust for measurement may execute without first being measured. This is also known as transitive trust or inductive trust.

3.1.4 Attestation

Attestation is a mechanism for proving something about a system. The values of the PCRs are signed by an Attestation Identity Key and sent to the challenger along with the measurement log. To verify the results, the challenger must verify the signature, then verify the values of the PCRs by replaying the measurement log.

3.1.5 Binding Data to a TPM

Bound data is data that has been encrypted by a TPM using a key that is part of the root of trust for storage. Since the root of trust of storage is different for every TPM, the data can only be decrypted by the TPM that originally encrypted the data. If the key used is a migratable key, however, then it can be migrated to the root of trust for storage of a different TPM allowing the data to be decrypted by a different TPM.

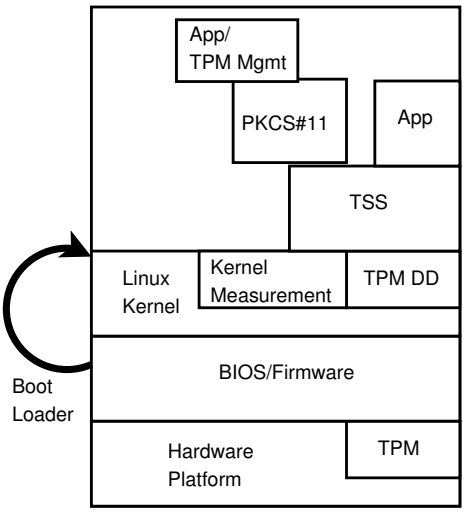

Figure 2: Trusted Computing Enabled Operating System

3.1.6 Sealing Data to a TPM

Sealed data is bound data that additionally records the values of selected PCRs at the time the data is encrypted. In addition to the restrictions associated with bound data, sealed data can only be decrypted when the selected PCRs have the same values they had at the time of encryption.

4 Components of Trusted Computing on Linux

Several components are required to enable an operating system to use the Trusted Computing concepts. These components are described in this section.

4.1 TPM

The Trusted Platform Module (TPM) is a hardware component that provides the ability to securely protect and store keys, certificates, passwords, and data in general. The TPM enables more secure storage of data through asymmetric key operations that include on-chip key generation (using a hardware random number generator), and public/private key pair encryption and signature operations. The TPM provides hardware-based protection of data because the private key used to protect the data is never exposed in the clear outside of the TPM. Additionally, the key is only valid on the TPM on which it was created unless created migratable and migrated by the user to a new TPM.

The TPM provides functionality to securely store hash values that represent platform configuration information. The secure reporting of these values, if authorized by the platform owner, enables verifiable attestation of a platform configuration. Data can also be protected under these values, requiring the platform to be in the same configuration to access the data as when the data was first protected.

The owner of the platform controls the TPM. There are initialization and management functions that allow the owner to turn on and off functionality, reset the TPM, and take ownership of the TPM. There are strong controls to protect the privacy of an owner and user.[3] The platform owner must opt-in. Any user, even if different from the owner, may opt-out.

Each TPM contains a unique Endorsement Key. This key can be used by a TPM owner to anonymously establish Attestation Identity Keys (AIKs). Since privacy concerns prevent the Endorsement Key from being used to sign data generated internally by the TPM, an AIK is used. An AIK is an alias to the Endorsement Key. The TPM owner controls the creation and activation of an AIK as well as the data associated with the AIK.[TCGMain],[TPM]

[3] See Section 7.1 for more details.

4.1.1 A Software-based TPM Emulator for Linux

If you don't have a machine that has a TPM but you'd like to start experimenting with Trusted Computing and the TSS API, a software TPM emulator can provide a development environment in which to test your program. While a software TPM will provide you with a development environment, it can't provide you with the "trust" that a hardware TPM can provide.

The advantage of having the TPM be a hardware component is the ability to begin measuring a system almost immediately at boot time. This is the start of the "chain of trust." By measuring as early in the boot cycle as possible, you lessen the chance that an untrusted component (hardware or software) can be introduced without being noticed. There must be an initial "trusted" measurement established, known as the root of trust for measurement, and the measurement "chain" must not be interruped.

With a software TPM emulator, you have delayed the initial measurement long into the boot cycle of the system. Many measurements have not occurred and so the trust of the system can not be fully validated. So while you would not want to rely on a software TPM to validate the trust of your systemm, it does provide you with a development environment to begin preparing to take advantage of trusted computing.

Mario Sasser, a student at the Swiss Federal Institute of Technology has created a TPM emulator that runs as a kernel module.[Strasser] It is not a full implementation of the specification and it is still under development. It is available from `http://www.infsec.ethz.ch/people/psevinc/` or `https://developer.berlios.de/projects/tpm-emulator`.

4.2 TPM Device Driver

The TPM device driver is a driver for the Linux kernel to communicate TPM commands and their results between the TCG Software Stack (TSS) and the TPM device. Today's TPMs are connected to the LPC bus. The TPM hardware is located by the driver from the PCI device for the LPC bus and attempts to read manufacturer specific information at manufacturer specific offsets from the standard TPM address. Since the TPM device can only handle one command at a time and the result must be cleared before another command is issued, the TPM device driver takes special care to provide that only one command is in-flight at a time and that the data is returned to only the requester. Rather than tie up all system resources with an ioctl, the command is transmitted and the result gathered into a driver buffer on a write call. Then the result is copied to the same user on a subsequent read call. This coupling of write and read calls is enforced by locks, the file structure's private data pointer and timeouts. At the direction of the Trusted Computing Group Specification, the TSS is the only interface allowed to communicate with the TPM thus, the driver only allows one open at a time, which is done by the TSS at boot time. The driver allows canceling an in-flight command with its sysfs file `cancel`. Other sysfs files provided by the driver are `pcrs` for reading current pcr values, `caps` for reading some basic capability information about the TPM such as manufacturer and version and `pubek` for reading the public portion of the Endorsement Key if allowed by the device. The current driver supports the Atmel and National Semiconductor version 1.1 TPMs, which are polled to determine when the result is available. The common functionality of the driver is in the `tpm` kernel module, and the vendor specifics are in a separate module. The driver is available on Sourceforge at `http://sourceforge.`

`net/projects/tpmdd/` under the project name tpmdd and has been in Linux kernel versions since 2.6.12.

4.3 TSS

The TCG Software Stack (TSS) is the API that applications use to interface with the TPM.

4.3.1 TSS Background

The TCG Software Stack (TSS)[TSS] is the set of software components that supports an application's use of a platform's TPM. The TSS is composed of a set of software modules and components that allow applications to communicate with a TPM.

The goals of the TSS are:

- Supply one entry point for applications to the TPM's functionality. (Provided by the TSS Service Provider Interface (The TSS API)).
- Provide synchronized access to the TPM. (Provided by the TSS Core Services Daemon(TCSD)).
- Hide issues such as byte ordering and alignment from the application. (Provided by the TSS Service Provider Interface (TSPI)).
- Manage TPM resources. (Provided by the TCSD).

All components of the TSS reside in user space, interfacing with the TPM through the TPM device driver.

TPM services provided through the TSS API are:

- RSA key pair generation
- RSA encryption and decryption using PKCS v1.5 and OAEP padding
- RSA sign/verify
- Extend data into the TPM's PCRs and log these events
- Seal data to arbitrary PCRs
- RNG
- RSA key storage

Applications will link with the TSP library, which provides them the TSS API and the underlying code necessary to connect to local and remote TCS daemons, which manage the resources of an individual TPM.

4.3.2 The TrouSerS project

The TrouSerS project aims to release a fully TSS 1.1 specification compliant stack, following up with releases for each successive release of the TSS spec. TrouSerS is released under the terms of the Common Public License, with a full API compliance test suite and example code (both licensed under the GPL) and documentation. TrouSerS was tested against the Atmel TPM on i386 Linux and a software TPM on PPC64. TrouSerS is available in source tarball form and from CVS at `http://trousers.sf.net/`.

4.3.3 Technical features not in the TSS specification

By utilizing udev.permissions for the TPM device file and creating a UID and GID just for the TSS, the TrouSerS TCS daemon runs without root owned resources.

For machines with no TPM support in the BIOS, TrouSerS supports an application level interface to the physical presence commands when the TCS daemon is executing in single user mode. This allows administrators to enable, disable, or reset their TPMs where a BIOS/firmware option is not available. This interface is automatically closed at the TCS level when the TCS daemon is not running in single user mode, or cannot determine the run level of the system.

In order to maintain logs of all PCR extend operations on a machine, TrouSerS supports a pluggable interface to retrieve event log data. Presumably, the log data would be provided by the Integrity Measurement Architecture (IMA) (see Section 4.6 below). As executable content is loaded and extended by the kernel, a log of each extend event is recorded and made available through sysfs. The data is then retrieved by the TCS Daemon on the next GetPcrEvent API call.

To maintain the integrity of BIOS and kernel controlled PCRs, TrouSerS supports configurable sets of PCRs that cannot be extended through the TSS. This is useful; for example in keeping users from extending BIOS controlled PCRs or for blocking access to an IMA controlled PCR.

TCP/IP sockets were chosen as the interface between TrouSerS' TSP and TCS daemon, for both local and remote access. This makes connecting to a TCSD locally and remotely essentially the same operation. Access control to the listening socket of the TCSD should be controlled with firewall rules. Access controls to the TCSD's internal functionality was implemented as a set of 'operations,' each of which enable a set of functions to be accessible to a remote user that will enable that user to accomplish the operation. For instance, enabling the seal operation allows a remote user to open and close a context, create authorization sessions, load a key, and seal data. By default, all functionality is available to local users and denied to remote users.

4.4 TPM Management

Some TPM management functionality was implemented in the tpm-mgmt package and the openCryptoki package. The tpm-mgmt package contains support for controlling the TPM (enabling, activating, and so on) and for initializing and utilizing the PKCS#11 support that is provided in the openCryptoki package.

4.4.1 Controlling the TPM

The owner of the platform has full control of the TPM residing on that platform. A TPM maintains three discrete states: enabled or disabled, active or inactive, and owned or unowned. The platform owner controls setting these states. These states, when combined, form eight operational modes. Each operational mode dictates what commands are available.

Typically, a TPM is shipped disabled, inactive and unowned. In this operational mode, a very limited set of commands is available. This limited set of commands consists mainly of self-test functions, capability functions and non-volatile storage functions. In order to take full advantage of the TPM, the platform owner must enable, activate, and take ownership of the TPM. Enabling and activating the TPM is typically performed using the platform BIOS or firmware. If the BIOS or firmware does not provide this support, but the TPM allows for the establishment of physical presence through software, then TrouSerS can be used to establish physical presence and accomplish the task of enabling and activating the TPM. Taking

ownership of the TPM sets the owner password, which is required to execute certain commands.

The tpm-mgmt package contains the commands that are used to control the TPM as described above, as well as perform other tasks.

4.4.2 PKCS#11 Support

The PKCS#11 standard defines an API interface used to interact with cryptographic devices. Through this API, cryptographic devices are represented as tokens, which provide applications a common way of viewing and accessing the functionality of the device. Providing a PKCS#11 interface allows applications that support the PKCS#11 API to take advantage of the TPM immediately.

The TPM PKCS#11 interface is implemented in the openCryptoki package as the TPM token. Each user defined to the system has their own private TPM token data store that can hold both public and private PKCS#11 objects. All private PKCS#11 objects are protected by the TPM's root of trust for storage. A symmetric key is used to encrypt all private PKCS#11 objects. The symmetric key is protected by an asymmetric TPM key that uses the user's PKCS#11 user login PIN as the key's authorization data. A user must be able to successfully login to the PKCS#11 token in order to use a private PKCS#11 object. The TPM token provides key generation, encryption and signature operations through the RSA, AES, triple DES (3DES), and DES mechanisms.

The following RSA mechanisms are supported (as defined in the PKCS#11 Cryptographic Token Interface Standard[PKCS11]):

- PKCS#1 RSA key pair generation
- PKCS#1 RSA
- PKCS#1 RSA signature with SHA-1 or MD5

The following mechanisms are supported AES, 3DES, and DES (as defined in the PKCS#11 Cryptographic Token Interface Standard[PKCS11]):

- Key generation
- Encryption and decryption in ECB, CBC or CBC with PKCS padding modes

The RSA mechanisms utilize the TSS to perform the required operations. By utilizing the TSS, all RSA private key operations are performed securely in the TPM. The symmetric mechanisms are provided to allow for the protection of data through symmetric encryption. The symmetric key used to protect the data is created on the TPM token and is thus, protected by the TPM. Since the key is protected by the TPM, the data is protected by the TPM.

Before any PKCS#11 token is able to be used it must be initialized. Since each user has their own TPM token data store, each user must perform this intialization step. Once the data store is initialized it can be used by applications supporting the PKCS#11 API.

The tpm-mgmt package contains commands to initialize the TPM token data store as well as perform other tasks. Some of the other tasks are:

- Import X509 certificates and/or RSA key pairs

  Existing certificates and/or key pairs can be stored in the data store to be used by applications.

- List the PKCS#11 objects in the data store

  In addition to any objects that you import, applications may have created or generated objects in the data store. tpm-mgmt lets you get a list of all the PKCS#11 objects that exist in the data store.

- Protect data using the "User Data Protection Key"

  Protect data by encrypting it with a random 256-bit AES key. The key is created as a PKCS#11 secret key object with an label attribute of "User Data Protection Key." This label attribute is used to obtain a PKCS#11 handle to the key and perform encryption, or decryption operations on the data.

- Change the PKCS#11 PINs (Security Officer and User)

  PKCS#11 tokens have security officer and a user PINs associated with them. It may be necessary or desirable to change one or both of these PINs at some point in time.

4.5 Boot Loader

To preserve the chain of trust beyond the boot loader, the boot loader must be instrumented to measure the kernel before it passes over control. The root of trust for measurement measured the BIOS before it transferred control, the BIOS measured the boot loader. Now the boot loader must measure the kernel. Seiji Munetoh and Y. Yamashita of IBM's Tokyo Research Lab have instrumented Grub v.0.94 and v.0.96 to perform the required measurements. Since Grub is a multi-stage boot loader, each stage measures the next before it transfers control. This is a slight simplification. Stage 1 is measured by the BIOS. Stage 1 measures the first sector of stage 1.5, which then measures the rest of stage 1.5 and stage 2. The configuration file is measured early with additional measurements of files referred to in configuration files taken in sequence. If stage 1.5 is not loaded, stage 1 measures the first sector of stage 2 instead and that sector measures the rest of stage 2. The grub measurements are stored in PCR 4, the grub configuration file measurement is stored in PCR 5, and the kernel measurement is stored in PCR 8. The PCRs used are configurable but the defaults meet the requirements of the TCG PC Specific Implementation Specification Version 1.1[TCGPC].

Lilo has also been instrumented to take the measurements by the Dartmouth Enforcer team. (See more detail about this project in Section 6.1.1).

4.6 Kernel Measurement Architecture—IMA

Reiner Sailer, and others of the IBM T.J. Watson Research Center have extended the chain of trust to the Linux kernel by implementing a measurement architecture for the kernel as a LSM.[SailerIMA] (Note: To effectively preserve the chain of trust, the LSM must be compiled into the kernel rather than dynamically loaded.) The `file_mmap` hook is used to perform the measurement on anything that is mapped executable before it is loaded into virtual memory. Kernel modules are measured just before they are loaded. The measurement is used to extend one of the PCRs numbered between 7-16, as configured in the kernel to allow for somewhat flexible PCR use. PCR 9

is the default. Other files that are read and interpreted, such as bash scripts or Apache configuration files, require application modifications to measure these files. Measurements are cached to reduce performance impact. Performance, usability, and bypass-protection are all addressed in the Sailer, et al. report. Enforcement is not part of this architecture. The measurements (and measurement log) are intended to be used by the challenger during remote attestation to determine the integrity of the system, rather than by the system to enforce a security policy.

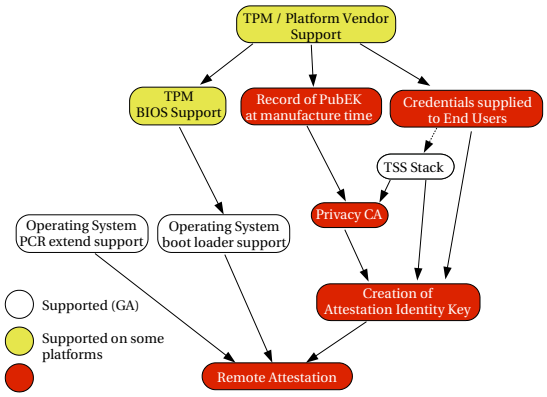

Figure 3: Dependencies for Full Trusted Computing Deployment

5 Trusted Computing on Linux Now and in the Future

Although version 1.1 TPMs provide many features usable today, significant hurdles exist to deploying the full capabilities of Trusted Computing outside a structured or corporate environment. Software that exists today basically enables the use of a TPM as one would use a smartcard. Other features, such as remote attestation, have more extensive requirements. The components required to implement remote attesation can be seen in Figure 3.

In order to implement remote attestation, TPM and Platform Vendor Support is required to:

- Put TPM support in the BIOS of shipping platforms (currently shipping).
- Record the Public Endorsement Key in some way (such as make a cryptographic hash) in order to identify whether a platform has a true TPM.
- Create and ship the TPM credential and the platform credential.

As long as the platform vendor has included TPM support in the BIOS, a corporate environment can work around the lack of the other elements by recording the PubEK as machines are deployed and maintaining a PKI internally. However, in order to enable remote attestation for general use by the public, a new infrastructure among hardware and software vendors must be created. This infrastructure would provide the credentials and a hash of the PubEK of shipping systems to Privacy CAs. The Privacy CAs differ from existing CAs in the key creation, certificate application, and certificate delivery mechanism, so new CAs are needed or existing CAs must implement the required software and procedures. At best, shipping platforms that fully support remote attestation are years away. Because of the lack of this infrastructure, no currently shipping platforms will have the capability to provide remote attestation for general use.

To make use of the more advanced features the TPM can provide, in addition to the infrastructure element listed above, a Linux distro would need to:

- Incorporate the measurement architecture

into the kernel.

- Ship measurement support for the boot loader.
- Include TrouSerS or some TCG Software Stack.
- Include attestation software.
- Include software for safe handling of the TPM and Platform credentials.

When this level of TPM hardware support is achieved, the groundwork will be laid to enable the software that will be used for attestation. Ideas for an interoperable attestation interface include a stand-alone attestation daemon and a modified TLS protocol that includes attestation. Until one of these solutions is specified and implemented, attestation solutions are ad hoc at best.

Finally, before general purpose remote attestation can be widely used, tools and best practice guidelines are needed to help define valid policies and maintain policy currency. Depending on the measurement architectures implemented by various operating systems, the policy becomes quite complex very quickly.

6 Example Uses of the TPM

Given the passive nature of the TPM device, the decision about its usefulness rests almost entirely on how one will use the device. Many of the doomsday scenarios surrounding the TPM device are based on scenarios involving software that Linux users will never agree to run on their hardware. In this section, some of the most promising uses of the TPM device are addressed. See also "Interesting Uses of Trusted Computing"[Anonymous] and "The Role of TPM in Enterprise Security"[Sailer] for more discussion around this topic. Anonymous notes in the first article "Before wide-scale use of TC for DRM, it will be necessary for the manufacturers, software vendors and content providers to get past a few tiny details, like setting up a global, universal, widely trusted and secure PKI. Hopefully readers . . . will understand that this is not exactly a trivial problem." The uses discussed below do not depend on full deployment of a complete Trusted Computing infrastructure but only on existing capabilities.

6.1 Beyond Measurement –Enforcement

A couple of examples of how the Trusted Computing measurements can be used to enforce a security policy exist and are described in this section.

6.1.1 Dartmouth's Enforcer

Enforcer is an LSM that measures each file as it is opened.[MacDonald] The measurement is compared against a database of previous measurements. File attributes (mtime, inode number, and so on) are also inspected. If the file has changed, the system will either log the condition, deny access to the file, panic the kernel, or "lock" the TPM (by extending the PCR used by Enforcer with random data, which makes decrypting data sealed to this PCR fail) based on the setting selected by the administrator. Enforcer does not require a TPM, but can optionally use the TPM to protect the database and configuration files. Enforcer also provides `helper`, which allows encrypting a loopback file system with a key protected by the TPM. Enforcer is available at `http://sourceforge.net/projects/enforcer/`.

6.1.2 Trusted Linux Client

The IMA kernel measurement architecture described previously provides no direct enforcement mechanisms. Dave Safford of IBM's T.J. Watson Research Center has proposed an extension to the concept that includes enforcement. The idea is to provide a series of LSMs that provide authenticated boot, encrypted home directories and file attribute checking. The first module validates the integrity of initrd and the kernel, and releases a TPM based kernel symmetric key. The key is used to derive keys for encrypted home directories via loopback file system and authenticated file attribute checking. The next module deals with extended attributes that are applied to every file including a file hash, MAC label, and others. The derived symmetric key is used to HMAC these attributes, and the value is checked and cached once at open/execute. A final module provides LOMAC style mandatory access control. See the presentation 'Putting Trust into Computing: Where does it Fit? —RSA Conference 2005' for an overview of this concept.[TCGRSA]

6.2 Enterprise Uses

Since the Trusted Computing Group is an industry led standards organization it is no surprise that compelling use cases exist for the enterprise.

6.2.1 Network attach

Enterprise networks are often described as 'hard and crunchy on the outside, but soft and chewy on the inside' reflecting the fact that they typically have very good perimeter defenses, but are less well protected from the inside. This poses a problem for enterprises that allow their employees to take mobile computing devices on the road and connect to non-protected networks. Viruses very often use unprotected mobile devices as a gateway device through which to invade corporate networks. To defeat viruses and worms that come in this way, more internal firewalls and choke points are architected into the network. A few vendors are now offering compliance checking software that challenge mobile devices when they attempt to reattach to the internal network; this is to prove that they meet corporate guidelines before allowing them to attach. This is typically done through agent software running on the mobile device. The agent software becomes the logical attack point.

Trusted Computing can make this compliance checking stronger. The Trusted Network Connect (TNC) subgroup of the Trusted Computing Group has released a specification[TNC] for client and server APIs that allow development of plugins for existing network attach products to do client integrity measurement and server-side verification of client integrity. The plugins add remote attestation capabilities to existing network attach products. The products continue to operate in their normal manner with the assurance that the client agents have not been subverted.

6.2.2 Systems management

Remote attestation is extremely useful when combined with systems management software. System integrity of the managed system is verified through remote attestation periodically, or on demand. Tied into the intrusion detection system, systematic integrity checking ensures that compromises can be quickly detected.

6.2.3 Common Criteria Compliance

Common Criteria evaluations are based on a well-defined and usually strict Security Target. Installing new software may cause the system to flip to an unevaluated mode. Remote attestation can be employed to confirm that all systems that are required to be Common Criteria compliant, retain adherence to the Security Target. This is one of the simpler uses of remote attestation since the policy to which the client must adhere is so static, strict, and well-defined that it eliminates the need for much policy management.

6.3 Uses by Individuals

The TPM can also be used to secure individual's computer and data.

6.3.1 TPM Keyring

The TPM Keyring application illustrates usage of the TSS API, and some of the properties of keys created with a TPM. TPM Keyring is licensed under the GPL and contains examples of how to wrap a software generated key with a TPM key, connect to local and remote TCS daemons, store and retrieve keys and encrypt and decrypt data using the TSS API. The source is available from CVS at `http://trousers.sf.net/`. TPM Keyring will wrap a software generated OpenSSL key with the Storage Root Key (SRK) of an arbitrary number of users. Once each user has a copy of this wrapped key, all users of the keyring can send secure messages to one another, but no user can give the key to anyone else, except the owner of the original OpenSSL key. Scripts are also provided to easily encrypt a symmetric key and use OpenSSL to encrypt large files.

You can imagine using `tpm_keyring` itself, or the concepts presented by `tpm_keyring` to create private and secure peer-to-peer networks.

Creating a New Keyring `tpm_keyring` generates a plaintext RSA key pair in memory and wraps the private key of that key with the public key of your TPM's root key. The plaintext RSA key is then encrypted with a password (that you are prompted for), and written to disk. Once the new key ring is created, you should move the encrypted software generated key to a safe place off your machine.

Adding Members to a Keyring After you've created a keyring, you'll probably want to add members so that you can start exchanging data. You'll need to bring your encrypted key file out of retirement from off-site backup in order to wrap it with your friend's TPM's root key. Contact this person and ask for their IP address or hostname. The public portion of your friend's root key will be pulled out of their TCS daemon's persistent storage and used to wrap your plaintext key. The resulting encrypted key is stored in your friend's persistent storage, with a UUID generated by hashing the name of the keyring you created. Let your friend know the name you gave the keyring so that they can import the key.

Importing a Key Once a friend has stored their key in your persistent store, you can import it so that `tpm_keyring` can use it. Run the import command with the same name of the key ring that your friend created and some hostname and alias pair to help you remember the friend who's keyring you're joining.

6.3.2 eCryptfs

The need for disk encryption is often overlooked but well motivated by security events in the news; for example, this article at `http://sanjose.bizjournals.com/sanjose/stories/2005/04/04/daily47.html` about a physical theft compromising 185,000 patients' medical records. eCryptfs [Halcrow:2005], being presented at OLS 2005 by Michael Halcrow, offers as an option, file encryption using TPM keys. In the case mentioned, if the information on disk had been encrypted with a TPM key, the data would not have been recoverable by the thief. Hot swappable drives and mobile storage being so easy to remove, in particular, benefit from encryption tied to a TPM key.

6.3.3 mod_ssl

Another use for the TPM is to provide secure storage for SSL private keys. Many system administrators face a problem of securely protecting the SSL private key and still being able to restart a web server as needed without human interaction. With the TPM, the private key can be bound, or optionally, sealed to a certain set of PCRs allowing it to be unsealed as necessary for starting SSL in a trusted environment on the expected platform.

6.3.4 GnuPG

Project Aegypten (`http://www.gnupg.org/aegypten/`) has extended GnuPG and other related projects so that GnuPG can use keys stored on smartcards. This can be extended to enable GnuPG to use keys stored on the TPM.

6.3.5 OpenSSH

Similarly, as the mod_ssl use case mentioned above, the TPM can be used to provide secure storage for SSH keys. In addition to the server key being protected, individuals can use their own TPM key to protect their SSH keys.

7 Pros and Cons of Trusted Computing

So much emotionally charged material has been written about Trusted Computing that it is difficult to separate the wheat from the chaff.

The seminal anti-TCG commentary is available from [Anderson], [RMS], [Schoen:2003], and [Moglen] with the seminal pro-TCG commentary available from [Safford].

Seth Schoen has written an excellent paper "EFF Comments on TCG Design Implementation and Usage Principles 0.95" with thoughtful, informed criticism of Trusted Computing. This paper makes the point "Many [criticisms] depend on what platform or operating system vendors do." [Schoen:2004]

Catherine Flick has written a comprehensive survey of the criticisms of Trusted Computing in her honor's thesis entitled "The Controversy over Trusted Computing." [Flick:2004]

This paper will address and attempt to clarify only the few technical issues that seem to come up repeatedly.

7.1 Privacy

There were many valid privacy concerns surrounding the 1.1 version of the TPM specification requiring 'trusted third parties' (PKI vendors) to issue AIKs. The concern was that the

trusted third party is able to link all pseudonymous AIKs back to a single Platform Credential. To address this concern, v. 1.2 now provides a new way for requesting AIKs called Direct Anonymous Attestation. DAA is beyond the scope of this paper, more information can be found in [Brickell]. In the v.1.1 timeframe, privacy concerns are mitigated by the fact that no manufacturer records a hash of the EK before shipping the TPM.

The measurement log, as maintained by the IMA kernel measurement architecture contains an entry about every executable that has run since boot. Like systems management data, this measurment log data may be considered sensitive data that should not be shared beyond the confines of the system, or perhaps the local network. A couple of solutions have been proposed for this problem. One very interesting solution calls for a compact verifier which verifies the targeted system and reports the results back to the challenger without leaking data. The verifier is a stock small entity with no private attributes. In this solution, the verifier would ideally be a small neighboring partition or part of the hypervisor[Garfinkel:2003]. Another solution calls for attestation based on abstract properties rather than complete knowledge of the system attributes. See [SadStu:2004] and [**?**].

7.2 TPM Malfunction

What happens to my encrypted data if the TPM on my motherboard dies? This depends on how the data was encrypted and what type of key was used to encrypt the data. When TPM keys are created, you have the option of making the key migratable. This implies a trade-off between security and availability so you are encouraged to consider their goal for each individual key. If the key was created migratable and the data is bound but not sealed to the TPM, you can import the key on a new TPM, restore the encrypted data from a backup, and use the key on the new TPM to access the data. If the key was not created as a migratable key or the data was sealed to the TPM, then the data will be lost. [4] Note that a backup of the migratable key must be made and stored in a safe place.

7.3 Secure Boot

Will trusted computing help me be able to perform secure boot as described by Arbaugh, and others[Arbaugh:1997] Arbaugh and others described “A Secure and Reliable Bootstrap Architecture” that is widely believed to be an inspiration for Trusted Computing. This paper describes the AEGIS architecture for establishing a chain of trust, driving the trust to lower levels of the system, and, based on those elements, secure boot. Trusted Computing supplies some elements of this architecture, but the TPM cannot completely replace the PROM board described in the paper. Commercially available TPMs currently do not have enough storage to contain the secure recovery code. Additionally, the infrastructural [5] and procedural hurdles, described in Section 5, would still have to be overcome. Trusted Computing enhanced BIOSes do not currently perform the verfication described in the paper, so the secure recovery has to be added to the BIOS implementation or enforced at a higher level than described in the paper.

[4] This is a slight simplification. If the PCR(s) selected for the seal operation on the new machine are exactly identical to the ones that the data was sealed to, then you can migrate the sealed data. Depending on the PCR chosen, to have the PCRs be the same the system, kernel, boot loader, and patch level of the two systems would have to be identical.

[5] The assumption of “the existence of a cryptographic certificate authority infrastructure” and the assumption that “some trusted source exists for recovery purposes.”

7.4 Kernel Lock-out

Does trusted computing lock me out of being able to boot my custom kernel? No, this functionality does not exist. The technical, infrastructural, and procedural hurdles, described in Section 5, would have to be overcome to enforce this technology on a global basis. Will this technology ever exist? There are cultural and political forces barring adoption of technology that takes aways the individual's right to run their operating system of choice on their general purpose computer. There is economic disincentive to forcibly limiting general purpose computing. The realization of this scenario depends more on political factors than technical capabilities.

7.5 Free and Open Source BIOS

Will I still be able to replace my computer's BIOS with a free BIOS? Trusted Computing does not prevent you from replacing your system BIOS with one of the free BIOS replacements, however, doing so currently violates the chain of trust. The TPM on the system can be used as a smartcard, but attestation would be broken. Free BIOS replacements can implement the relevant measurement architecture and maintain the chain of trust, if the boot block remains immutable and measures the new BIOS before it takes control of the system. The challenger during attestation would see that a different BIOS is loaded and can choose to trust the system, or not, based on their level of trust in the free BIOS.

7.6 Specific Additional Concerns

Is the Trusted Computing Group taking comments about specific concerns? The Trusted Computing Group has interacted with many people and organizations who have expressed concern with the group's specifications. Several of the concerns have resulted in changes to the TPM specification, for example, the introduction of Direct Anonymous Attestation, which solves many of the privacy problems that the BSI and individuals expressed. The Trusted Computing Group invites serious comments to be sent to admin@trustedcomputinggroup.org, or entered into their comment web page at `https://www.trustedcomputinggroup.org/about/contact_us/`.

8 Conclusion

This paper has quickly covered a great deal of ground from Trusted Computing definitions and components to uses and common concerns; no discussion about Trusted Computing and Linux is complete without citing Linus Torvald's famous email 'Flame Linus to a crisp!' proclaiming 'DRM is perfectly ok with Linux.' [6] Even though Linus considers DRM okay, the hope is that this paper makes clear that the uses of Trusted Computing are not limited to DRM, and that individual Linux users can use the TPM to improve their security. The Trusted Computing Group has shown itself willing to work with serious critiques and the Linux community is capabable of defending itself from abusive technologies being adopted. With estimates that more than 20 million computers have been sold containing a TPM, and the existence of open source drivers and libraries, let's put this technology to productive use in ways that are compatible with free and open source philosophies. While the infrastructure and software for complete support are future work items, that does not prevent users from

[6] `http://marc.theaimsgroup.com/?l=linux-kernel&m=105115686114064&w=2`

utilizing their TPM to gain secure storage for their personal keys and data through projects already available or proposed by this and other papers.

9 Legal Statement

References

[Anderson] Ross Anderson *'Trusted Computing' Frequently Asked Questions*, 2003, `http://www.cl.cam.ac.uk/users/rja14/tcpa-faq.html`

[Anonymous] Anonymous *Interesting Uses of Trusted Computing*, 2004, `http://invisiblog.com/1c801df4aee49232/article/0df117d5d9b32aea8bc23194ecc270ec`

[Arbaugh:1997] William A. Arbaugh, David J. Farber, and Jonathan M. Smith *A Secure and Reliable Bootstrap Architecture*, Proceedings of the IEEE Symposium on Security and Privacy, May 1997

[Brickell] E. Brickell, J. Camenisch, and L. Chen *Direct Anonymous Attestation*, Proceedings of 11th ACM Conference on Computer and Communications Security, 2004

[Fisher] Dennis Fisher, *Trusted Computing Group Forms*, eWeek, April 8, 2003, `http://www.eweek.com/article2/0,1759,1657467,00.asp`

[Flick:2004] Catherine Flick *The Controversy over Trusted Computing*, The University of Sydney, 2004, `http://luddite.cst.usyd.edu.au/~liedra/misc/Controversy_Over_Trusted_Computing.pdf`

[Garfinkel:2003] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh *Terra: A Virtual Machine-Based Platform for Trusted Computing*, Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003), October 2003

[Halcrow:2005] Michael Austin Halcrow *eCryptfs: An Enterprise-class Cryptographic Filesystem for Linux*, Ottawa Linux Symposium, 2005

[Haldar:2004] Vivek Haldar, Deepak Chandra, and Michael Franz *Semantic*

Remote attestation: A Virtual Machine Directed Approach to Trusted Computing, USENIX Virtual Machine Research and Technology Symposium, 2004, http://gandalf.ics.uci.edu/~haldar/pubs/trustedvm-tr.pdf

[MacDonald] Rich MacDonald, Sean Smith, John Marchesini, and Omen Wild *Bear: An Open-Source Virtual Secure Coprocessor based on TCPA*, Dartmouth College, August 2003, http://www.cs.dartmouth.edu/~sws/papers/msmw03.pdf

[Moglen] Eben Moglen *Free Software Matters: Untrustworthy Computing*, 2002, http://emoglen.law.columbia.edu/publications/lu-22.html

[Mohamed] Arif Mohamed, *Who Can You Trust?*, ComputerWeekly.com, April 26, 2005, http://www.computerweekly.com/articles/article.asp?liArticleID=138102&liArticleTypeID=20&liCategoryID=2&liChannelID=22&liFlavourID=1&sSearch=&nPage=1

[RMS] Richard Stallman *Can you trust your computer?*, 2002, http://www.gnu.org/philosophy/can-you-trust.html

[PKCS11] RSA Laboratories PKCS #11 v2.20: Cryptographic Token Interface Standard, 28 June 2004, ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf

[SadStu:2004] Ahmad-Reza Sadeghi and Christian Stueble *Property-based Attestation for Computing Platforms: Caring about properties, not mechanisms*, 20th Annual Computer Security Applications Conference, December 2004, http://www.prosec.rub.de/Publications/SadStu2004.pdf

[Safford] David Safford *TCPA Misinformation Rebuttal*, IBM T.J. Watson Research Center, 2002, http://www.research.ibm.com/gsal/tcpa/tcpa_rebuttal.pdf

[SailerIMA] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn, *Design and Implementation of a TCG-based Integrity Measurement Architecture*, 13th Usenix Security Symposium, August 2004

[Sailer] Reiner Sailer, Leendert van Doorn, and James P. Ward *The Role of TPM in Enterprise Security*, September 2004, https://www.trustedcomputinggroup.org/press/news_articles/rc23363.pdf

[Schoen:2003] Seth Schoen *Trusted Computing Promise and Risk*, EFF, 2003, http://www.eff.org/Infrastructure/trusted_computing/20031001_tc.pdf

[Schoen:2004] Seth Schoen *EFF Comments on TCG Design, Implementation and Usage Principles 0.95*, EFF, 2004, http://www.eff.org/Infrastructure/trusted_computing/20041004_eff_comments_tcg_principles.pdf

[Smith:2005] Sean W. Smith *Trusted Computing Platforms Design and*

Applicatons, Springer, 2005, ISBN 0-387-23916-2

[Strasser] Mario Strasser *A Software-based TPM Emulator for Linux*, Swiss Federal Institute of Technology, 2004, http://www.infsec.ethz.ch/people/psevinc/TPMEmulatorReport.pdf

[TCGArch] TCG *Trusted Computing Group Specification Architectural Overview, Revision 1.2*, April 28, 2004, http://www.trustedcomputinggroup.org/downloads/TCG_1_0_Architecture_Overview.pdf

[TCGBackground] TCG *Trusted Computing Group Backgrounder*, January 2005, http://www.trustedcomputinggroup.org/downloads/background_docs/TCGBackgrounder_revised_012605.pdf

[TCGFAQ] TCG *Trusted Computing Group Fact Sheet*, 2005, http://www.trustedcomputinggroup.org/downloads/background_docs/FACTSHEET_revised_020105.pdf

[TCGMain] TCG *TCG Main Specification Version 1.1b*, 2003, http://www.trustedcomputinggroup.org/downloads/specifications/TCPA_Main_Architecture_v1_1b.zip

[TCGPC] TCG *TCG PC Specific Implementation Specification Version 1.1*, August 18, 2003, http://www.trustedcomputinggroup.org/downloads/TCG_PCSpecification_v1_1.pdf

[TCGRSA] TCG *Putting Trust into Computing: Where does it Fit? - RSA Conference 2005*, https://www.trustedcomputinggroup.org/downloads/Putting_Trust_Into_Computing_Where_Does_It_Fit_021405.pdf

[TNC] TCG *TCG TNC Architecture Version 1.0*, 2005, http://www.trustedcomputinggroup.org/downloads/specifications/TNC_Architecture_v1_0_r4.pdf

[TPM] TCG *TCG TPM Specification Version 1.2*, 2004, http://www.trustedcomputinggroup.org/downloads/specifications/mainP1DP_rev85.zip

[TSS] TCG *TCG Software Stack Specification Version 1.1*, 2003, http://www.trustedcomputinggroup.org/downloads/TSS_Version__1.1.pdf

NPTL Stabilization Project

NPTL Tests and Trace

Sébastien DECUGIS
Bull S.A.
sebastien.decugis@bull.net

Tony REIX
Bull S.A.
tony.reix@bull.net

Abstract

Our project is a stabilization effort on the GNU libc thread library NPTL—Native POSIX Threading Library. To achieve this, we focused our work on extending the pool of open-source tests and on providing a tool for tracing the internal mechanisms of the library.

This paper introduces our work with a short status on test coverage of NPTL at the beginning of the project (February 2004). It explains how we built the prioritized list of NPTL routines to be tested. It then describes our methodology for designing tests in the following areas: conformance to POSIX standard, scalability, and stress. It also explains how we have simplified the use of the tests and the analysis of the results. Finally, it provides figures about our results, and it shows how NPTL has evolved during year 2004.

The paper goes on to explain how this NPTL Trace Tool can help NPTL users, and hackers, to understand and fix problems. It describes the features of the tool and presents our chosen architecture. Finally, it shows the current status of the project and the possible future extensions.

1 Introduction

NPTL library was first released on September 2002 and merged with the glibc about sixteen months later. It was meant from the beginning to replace the LinuxThreads implementation, and therefore become the standard thread library in GNU systems. The new library provides full conformance to the POSIX[1] requirements, including signal support, very good performance and scalability.

Porting from LinuxThreads to NPTL was intended to be transparent; however, there are several cases where software using NPTL must be modified. There are some documented changes, such as signal handling or `getpid()` behavior. There are also changes in the application dynamics, such as those caused by threads being created more quickly. A user application coded with incorrect assumptions about multi-threaded programming can fail because of some of the semantic changes; such problems are very difficult to debug. We had the opportunity to work with IBM on some of their internal *BugZilla* reports, and in many cases the problem appeared because of changes in appli-

[1] The POSIX® standard refers to the IEEE Std 1003.1, a.k.a. Single UNIX Specification [1] v3. The current version is the 2004 Edition and includes Technical Corrigendum 1 and 2. POSIX is a registered trademark of the IEEE, Inc.

cation dynamics. Last but not least, NPTL is still under development. New features are being added from time to time. Fixes and optimizations are also frequent. All these code modifications have the potential to introduce new bugs.

Before NPTL could be used reliably in complex applications on production systems, it needed more substantial testing and validation. Any production system providing reliable applications should not crash or hang simply because the threading library is not stable. On the other hand, the new library provides very good performance and therefore is of great interest for these same systems.

To continuously improve the stability and quality of NPTL as it evolves, as well as to shorten the stabilization period after each change, we developed a robust set of regression and stress tests. Ideally, these tests would be run frequently during NPTL development to look for regressions and the tests can be augmented as new features are added. These tests should cover as many APIs, arguments to the APIs, and threading semantics as possible. The tests must remain independent of the implementation of the threading library so that the tests will not need to be changed each time the implementation changes. We will see in the next chapter how were specified and developed a list of tests, how we tried to make these tests runs as simple and user-friendly as possible, and finally we will show NPTL evolution, from the test results point of view, through year 2004.

As we have seen previously, many of the problems developers have to face when they port an application from LinuxThreads to NPTL are due to bugs located in their application, not in NPTL. Bugs dealing with multi-threading are particularly difficult to isolate and reproduce most of the time. As an example, when you run the program step-by-step in a debugger, the thread creation time is totally different than when it runs outside the debugger. These bugs can also depend on the machine load, on a device access slowing only one of the threads, or a multitude of factors, resulting in weeks of research and testing for an application developer. Moreover, many POSIX standard interfaces are quite intricate, and many programmers do not test all return codes from NPTL routines. At best, an application which receives an unexpected error code may crash; at worst, the application may corrupt data silently.

To solve these issues, we have developed a trace tool for NPTL, called POSIX Threads Trace Tool (PTT). This tool keeps track of all NPTL related events, such as thread creation, lock acquisition, with little impact on the application. By tracing the library internals, we can understand the chain of events which lead to a hang or strange behavior in the application. We can also understand how the application is really using NPTL functions, measure lock contention, and optimize both the NPTL implementation and the application's use of NPTL. Finally, these traces can prove that a bug is in NPTL or in the kernel, rather than in the application. The third chapter of this paper is dedicated to this trace tool. It attempts to show the limitations of existing tools, then describes the features of our tool and how these features can be used efficiently to solve real situations. It also shows the tool internals and its current limitations and future directions.

The paper concludes with an overview of the remaining work to do on NPTL, NPTL tests, and NPTL trace tool, in order to obtain a production quality level in this open-source product. It shows the current use of the tests in the library and kernel development process. It also shows that this testing effort is necessarily not a "one-shot" project, and that more people should be involved in projects like this one. As for the trace, it shows how the trace tool can be extended into a dynamic code checker, or into

a profiling tool, with a minimal effort. It also deals with how this modified NPTL can be set up in a production environment, and why people should use this tool.

2 NPTL Tests

The first part of our project consisted of improving the test coverage for the NPTL library. Our goal was to be as exhaustive as possible, at least as far as POSIX requirements are concerned. We focused on the POSIX standard [1] among all standards the NPTL is supposed to conform to, because it is largely used on other platforms, and so is important for ensuring portability of an application, and because reference is made in the library name—Native *POSIX* Thread Library—which means it is the first standard one would expect NPTL to conform to.

2.1 Situation on March 2004

When we started our project in early 2004, we isolated three open-source projects which provided test cases for NPTL.

The first one is the *GNU lib C* project [glibc] itself. NPTL source tree contains test cases that can be run against the freshly compiled glibc by issuing the `make check` command. These tests—about 160 files at that time—are not documented at all and hardly commented. Their naming convention is the only hint to guess what each test is supposed to do. We had a hard time reading each test case and writing a short abstract on what the test is really doing. As a synthesis, these tests are mostly regression tests which test for very specific features, and test coverage for each library routine is far from adequate or complete. Moreover, the tests are often very close to NPTL internals, which means more maintenance when the library implementation changes. These tests are useful for the glibc developers, but are by design too closely linked to testing implementation specifics to be usable as a proof of reliability or indicator of conformance.

The second project we focused on is the *Open POSIX Test Suite* [OPTS]. This is a pure test project, with a lite harness—the only constraint on a test case is its return value—and a simple structure, at least for the regression tests. For each library routine, an XML file contains a set of assertions that describe the POSIX standard requirements for this routine, and then the test cases are named according to the assertion they are testing. Extracting the coverage information is quite straightforward from this structure. The test cases are also often well documented, with few exceptions where the comments do not match the content.

The third project we considered is the *Linux Test Project* [LTP]. This is the most used open-source test project for Linux, but it appeared that it provides very few test cases for NPTL, aside from those of the OPTS which is included. Moreover, the structure is more complex and the format for test cases is more rigid than in the OPTS.

After this analysis, we decided to release our test cases to the OPTS, as they would later be included in LTP with the complete OPTS new release. In situations where we would have to write implementation-dependent test cases, they would be submitted to the glibc project directly, but we did our best to avoid NPTL-internals dependent code, as it would require more maintenance.

2.2 Prioritized list

Our next step was to find what to test. NPTL contains more than 150 routines, so we had to

establish our priority list based on the following criteria:

1. functions which are used the most frequently;
2. functions which are complex enough to possibly contain bugs, based on their algorithm; and
3. functions which are not just a wrapper to the kernel—as we are not testing the kernel.

To find out which functions are the most used, we chose seven multi-threaded applications representative of several computer science domains where multi-threading is frequently used. The selected software were: two different *Java Virtual Machine*s; *JOnAS*, an open-source Java application server, compiled with *gcj*; the *Apache* web server; the *squid* web cache and proxy; the *MySQL* database server; and *GLucas*, a scientific software described in the next chapter. Each application was analyzed with the *nm* utility to find out which NPTL routines were used. We also included a personal opinion based on our past experience with each routine, to establish the list.

This work has resulted in a complete list of functions split into 4 groups, from the most important to test to the less important. The first group (most important) contains 15 functions, dealing with *threads*, *mutexes* and *condvars*. The second group contains 27 functions, dealing with *threads*, *signals*, *cancellation* and *semaphores*. The complete list is available on our website [2]. The remaining functions belong to groups three and four. Even if NPTL contains 150+ functions, many of those functions are only used to change a value in a structure (attribute), so the bug probability is really small. With groups 1, 2 and 3 we cover almost all the functions which can encounter problems. At this time, only groups 1 and 2 have been completely tested. There is still a great amount of work remaining to complete the test coverage—this will be detailed later in this paper.

2.3 Methodology

We had to design a method for test writing. We based it on the OPTS method.

For each library routine to test, the first step was to analyze the POSIX standard and extract each assertion that the function has to verify to be compliant. For some functions the standard appeared to be unclear or contradictory. In these cases, we opened requests for clarification in the Austin Revision Group [3], so that the next Technical Corrigendum for the standard would clarify the obscure parts.

In the next step, these assertions were compared to those already present in the OPTS, and the *assertions.xml* file was updated according to the differences we found. Most of the differences we encountered so far were due to the POSIX standard evolution since the OPTS was first released.

The third step in the design was to check each existing test case for a given assertion, find out possible errors, try to check that all situations were tested, and list the missing cases which had to be written. For some assertions, we also had to specify stress tests to be written in order to be exhaustive, or when we could not figure another way to test a particular feature. We also specified scalability tests to be written for some functions where scalability is important, even if this is more a quality of implementation issue than part of the POSIX standard.

At each step of this process, for each function, we posted an article in our project forum, publicly available and accessible from our website

[2]. This allowed other people to check how they could help or see the rationale for a particular test.

Once our design was complete, we had to write the test cases and submit them to the OPTS project. We wrote three kinds of test cases: *conformance*, *stress* and *scalability*.

A conformance test runs for a short period of time and returns a value representing its result: PASSED, FAILED, UNRESOLVED, etc. See the OPTS documentation for a detailed explanation of return codes.

A stress test runs forever until it is interrupted with SIGUSR1 (means success) or a problem occurs (means failure). Most of the stress tests are very resource-consuming and are meant to be run alone in the system. In this way, it is possible to identify the cause of a failure, when any occurs.

A scalability test loops on a given operation until the number of iterations is reached or until failure, and saves the duration of each iteration. Then, measures are parsed with a mathematical algorithm which tells if the function is scalable (constant duration) or not (duration depends on the changing parameter). The algorithm is based on the least squares method to model the results. The table of measures can also be output and used to generate a graph of the results with the *gnuplot* tool. Figure 1 gives an example of such a graphical output. It shows the duration of `sem_open()` and `sem_close()` operations with an increasing number of opened semaphores in the system. Other examples can be found in our forum.

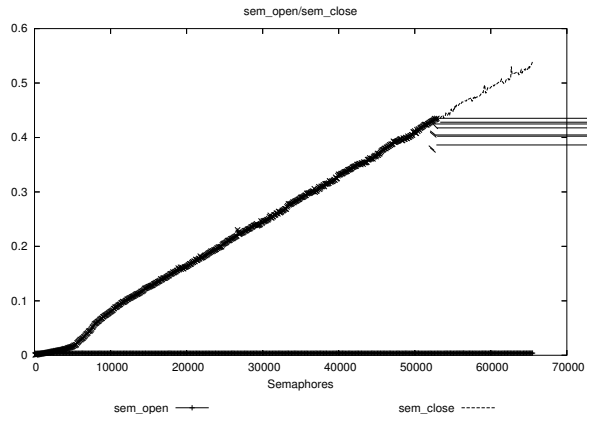

Figure 1: Graphical output sample

2.4 TSLogParser tool

To be useful, the tests must be run frequently, and the results must be easy to analyze and compare with other runs (references). Whereas running the tests is quite straightforward with OPTS (just set up the flags and run `make`), the analysis can be a real pain. As an example, after we completed the test writing for our first group of functions, we ran a complete test campaign on several Linux distributions with several hardware architectures—i686, PowerPC, ia64. We got a total of nine different configurations, and three runs on each configuration, which resulted in a total of about 50,000 test case results to digest. Needless to say, we needed automated tools to extract the useful information!

In many cases, comparing several runs and finding quickly what the differences are in detail is all we need. That supposes we have an arbitrary reference, to compare new code results to. But comparing huge log files is far from being easy. Using the *diff* tool is not a solution, as there are expected differences between the runs—order of test case execution, timestamps, random values— and a long time can be spent doing the analysis.

Another approach is to first make a synthesis of each run, and then only compare the synthesis. This is quite easy to achieve with tools such as *grep* and *wc*. The Scalable Test Platform (dis-

cussed in the next section), for example, uses this kind of summary tool. Anyway, this approach has some drawbacks. When a new failure appears, it is not possible to find out which test is failing. Also, if the success/failure distribution remains constant, while not involving the same individual tests, you won't see anything with your tool.

To address all these issues, we have designed a new tool: *TSLogParser* [4]. The main idea is to parse the log file of a test suite run and save the results and detailed information about each test into a database; and then be able to access all this information through a web interface. It allows filtering of results, to show only partial information or to access all details in just a few clicks. It also makes comparing several runs quite easy.

The structure of this tool has been designed to allow several kinds of test suites to be parsed and displayed the same way. The parser module which saves the log file into the database is written as a plug-in. The visualization and administration interfaces are not dependent on the test suite format. The current implementation is written in PHP and has been used with Apache and MySQL. It is able to compare up to 10 OPTS runs at once on a standard workstation. It also extracts statistical information from each run and allows filtering according to test status—for example one may want to hide all the successful tests or show only tests that end with a segmentation fault.

This tool has made the analysis of OPTS run results a fast and easy operation. It is a must-have—in our opinion—for anyone who is using the OPTS.

2.5 Scalable Test Platform

Another frequent issue in testing is that the active developers often lack the resources—time, hardware—to run complete test campaigns frequently. This can be solved, thanks to the *Scalable Test Platform* [STP] and *Patch Lifecycle Manager* [PLM] projects from *Open Source Development Labs* (OSDL).

PLM tracks the official kernel patches and allows uploading of new patches (either manually or automatically). STP allows people to request runs against tests, against any PLM patch, with a choice of Linux distributions and machine hardware. Our project contributed to STP by making the OPTS runnable through its interface. There is also a work in progress to bring the same patch feature that PLM provides for the glibc of the test system.

Once the requested test run completes, an email is sent to the requester with a summary of the results, and the complete log file is available for download. There is another work in progress to make the results available through the TSLogParser interface, because as we already discussed a summary can sometimes not contain enough information.

A very interesting feature of the PLM project is the ability to automatically pull new kernel patches and run a bunch of tests in STP—including the OPTS—against the new patched kernel. This allows very quick detection when new problems appear.

2.6 Situation on March 2005

After sixteen months of active work on this project, we are reaching the end of our credits. During this period, we were able to analyze and write test cases for all our 42 most important NPTL functions. A total of 246 conformance tests, 9 scalability tests and 16 stress tests have been written. These 42 functions correspond to 283 distinct assertions in POSIX, of which 246 (90%) are now covered by the OPTS and

135 (55% of OPTS) were contributed within our project. Figure 2 shows the evolution of the number of test cases (upper plot) and functions tested (lower plot) during our project. It only refers to our contribution, not to the complete OPTS project. The horizontal step during November 2004 corresponds to our first test campaign. The vertical step on February 2005 is due to semi-automated test generation for some signal-related functions. It is interesting to note that both plots are almost identical. This means that the amount of work required for each function to complete the OPTS work is almost the same for all functions. This needed work can be due to POSIX evolutions, as well as incomplete or invalid OPTS test cases.

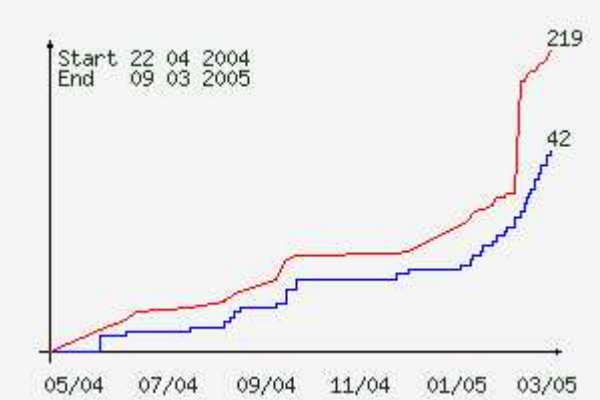

Figure 2: Project progression

Thanks to these test cases, a total of 22 defect reports have been issued—21 in the glibc and 1 in the kernel—most of which have been fixed in recent releases. The kernel defect deals with the scheduler and the `SCHED_RR` policy behavior on SMP machines. The glibc defects are either conformance bugs (wrong error code returned, bad `#include` files or symbol requirements), or functional bugs (flags role in `sigaction()`, behavior of timeouts with condvars), or else just bugs (segmentation faults, hangs, unexpected behaviors). We've also found a scalability issue with the function `sem_close()`, the duration of which depends on the number of opened semaphores.

In the meantime, 5 enhancement requests have been issued to the Austin Revision Group, about obscure or incomplete points in the POSIX standard. These requests addressed issues in `pthread_mutex_lock()`, `pthread_cond_wait()`, `pthread_cond_timedwait()`, `sigaction()`, and `sem_open()`. All have been accepted or are still pending.

The most important part of our project is not the number of bugs we have found, but the number of assertions which are now tested. For 42 functions we analyzed, almost *all* of what can be tested *is* now tested in OPTS. The test cases for these 42 functions cover all the current POSIX requirements.

2.7 NPTL Evolution over year 2004

As an example, we have run the current OPTS release with *Fedora Core 1* (FC1), *Fedora Core 2* (FC2) and *Fedora Core 3* (FC3) distributions, as well as an 'unstable' Fedora Core 3 update. After analyzing the results with the TSLogParser tool, we have come to find some interesting conclusions, detailed in the next paragraphs. This kind of analysis is very easy to achieve and can help tracking new bugs very quickly. Anyway, as the TSLogParser tool is interactive, we cannot reproduce its output in this document, and encourage the reader to check the tool web site [4] for examples, including the data discussed here.

AIO operations. Some test cases related to Asynchronous I/O operations, such as `aio_read()` or `aio_write()`, returned PASS with FC1 and FC2 and return FAIL or hang with the more recent distributions. This may indicate a bug in the new kernels or in the glibc.

Clock routines. Some tests related to the clock routines (`clock_settime()`, `nanosleep()`) did not pass in FC1, but things have been fixed since FC3.

Message queues. The message queues routines were not implemented in FC1, so the related tests reported a 'build failure' status. Everything is fine since FC2.

Sched routines. A few test cases related to the sched routines (`sched_setparam()`, `sched_setscheduler()`) won't compile in the latest FC3 update, whereas they passed in the previous releases.

There are some other test cases which would be worth a deeper investigation, but we won't enter into the details here. Reproducing these results is quite easy, and it would be valuable for Linux and the glibc that more people carry on this kind of work.

3 NPTL Trace

The second part of our work was dedicated to tracing NPTL.

3.1 Why Tracing?

Since more and more HyperThreaded or Multi-Core processors are available, it is expected that the design of many new applications will use multi-threading for running several tasks simultaneously and concurrently, in order to take profit of nearly all the available power of the machine.

Writing a portable multi-threaded application is a complex task: the POSIX Thread standard is not easy to understand. It provides ten kinds of objects: Thread, Mutex, Barrier, Conditional Variable, Semaphore, Spinlock, Timer, Read-Write lock, Message queues, and TLD (Thread Local Data). These objects are available under eighteen options: **BAR**, **CS**, **MSG**, **PS**, **RWL**, **SEM**, **SPI**, SS, **TCT**, **THR**, **TMO**, TPI, TPP, TPS, **TSA**, **TSH**, TSP, **TSS** [2] that may be supported or not by Operating Systems. (See [1] for the meaning of each option). NPTL provides about 150 different routines to manage the POSIX objects.

Also, "**anything can occur at any time**": a program must not assume that an Event A always occurs before—or after—Event B. That may be true on a small machine; but it will certainly be untrue some day on a bigger and faster machine at a customer site. That makes writing a multi-threaded application more complex than initially expected.

On Linux, NPTL is quite perfectly compliant with the POSIX Threads standard. Since several parts of the POSIX Threads standard are unspecified, they can be provided differently by two POSIX Threads libraries. So porting an application from another Operating System (though providing the same POSIX Threads objects and routines) to Linux may lead to bad surprises. Being able to quickly understand why an application behaves badly (hang, unexpected behavior, etc.) is critical for customers. Often, reproducing the problem in support labs is not possible since it may appear after days of computation. This may require sending a Linux guru to the customer site. Also, understanding quickly if the problem is in the application, in NPTL, or in the Linux Kernel is critical.

Analyzing a multi-threaded application showing a race condition or a hang with a debugger is not the right approach because it will certainly modify the way threads are scheduled, possibly causing the problem to disap-

[2] The options provided by recent GNU libc are highlighted in **bold**.

pear. The right approach is by using a less intrusive method, such as a trace tool. A NPTL trace tool enables recording of the most important multi-threading operations of an application or the main steps of NPTL with a minimal impact to the application (performances and flow of execution of threads). The trace can be analyzed once the problem has appeared and the application has stopped: it is **Post-Mortem Analysis**. If the impact on a critical application is acceptable, one can even continuously record the last few thousand traces so that analyzing a failure can be done when it occurs for the first time: it is **First Failure Data Capture**.

Why not use Linux Trace Toolkit [LTT]? First, LTT is designed to trace events in the kernel and not to trace programs in the user space. Second, LTT uses functions (like `write()`) that cannot be used when tracing NPTL. (See section 3.3.1 on page 120).

Why not simply use some *wrapper* enabling trace of only the calls of the application to NPTL routines? Because such a tool does not enable to analyze both the behavior of NPTL and that of the application. And, since it also requires to put in place a complex mechanism for collecting and storing traces, it is worth also tracing the behavior of NPTL routines, by adding traces inside its code.

So, as explained hereafter, we finally decided to design our own NPTL tracing tool.

3.2 Goals

At the beginning of 2004, when we started to add new tests for NPTL, we also started to study a NPTL trace tool. After discussing the design of such a tool with people involved in thread technology and in the glibc (IBM: F. Levine, E. Farchi; HP: J. Harrow; Intel: I. Perez-Gonzalez; etc.), we decided to propose to students from French Universities to study the architecture of the tool and to build it.

The POSIX Threads NPTL Trace Tool **[PTT]** has been designed to provide a solution to the requirements discussed previously. It addresses the three kinds of users described hereafter.

3.2.1 Users

We have studied the needs of three different kinds of users:

A **developer** in charge of writing, porting or maintaining a multi-threaded application. He mainly needs to see when his program calls NPTL routines and when it exits from them, with details about the parameters. He wants to be able to easily and quickly switch from a fast untraced NPTL to a traced NPTL, and vice-versa, without recompiling his application. When using the traced NPTL, the maximum acceptable decrease in the performances of his application is 10%.

A member of a **support team** that provides Linux skills to other people who write, test or use applications. This kind of user has skills about the Linux kernel and the GNU libc and he needs to see what is happening inside NPTL. Also he is very interested in generating traces at customer sites and to analyze them in his own offices.

A **hacker of NPTL**. Since analyzing why NPTL does not perform as expected is not an easy task, it is crucial to provide help. This way, more people could contribute to analyzing the behavior of NPTL and fix problems.

3.2.2 Features

Using PTT is a four step process:

1. build or get a traced NPTL library;
2. trace the application and build a binary trace;
3. translate the binary file into a text file that will be parsed by another program or that will be read manually;
4. analyze the trace, possibly with a tool helping to handle many objects and traces.

Several features are required:

- do not break the POSIX conformance rules (mainly cancellation).
- enable several people to trace different applications at the same time.
- handle large volumes of traces due to an application running days and weeks before the problem occurs: keep only last traces or manage very large trace files.
- give meaningful names to NPTL objects rather than hexadecimal addresses, since the application may create hundreds or thousands of objects of each kind.
- dynamically switch from a light trace to a richer or full trace.
- filter the decoded trace based on various criteria (name or kind of object, etc).
- start/stop the trace while the application is running, and provide solutions for handling incomplete traces.
- handle applications that fork new processes that must be traced.
- handle bad situations (hang, crash, kill).

3.3 Architecture

The main idea is to handle a buffer in shared memory: the threads of the application write the traces in the buffer, while a daemon (launched as a separate process) concurrently and periodically reads the traces in the buffer and writes them into the binary file. Traces are concurrently added by the threads into the buffer at the time the events occur. Figure 3 provides a simplified description of the architecture of PTT.

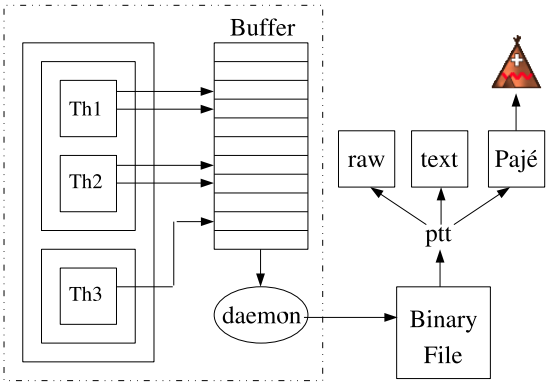

Figure 3: Architecture

3.3.1 POSIX Constraints

The architecture must take into account the following constraints.

- First, the **POSIX Threads standard** defines which routines can be a *Cancellation Point* (CP)[3]. POSIX defines three categories: the routines that shall be a CP,

[3] A Cancellation Point is a place where a thread can be canceled by means of `pthread_cancel()`. Such places appear when the cancellation *state* is set to *enabled*, and *type* is *deferred*.

those that cannot, and those that are undefined (free). It means that adding a CP into a routine that cannot have a CP is forbidden: routines like `printf()` cannot be called by trace code from inside NPTL routines. In few words, a NPTL trace mechanism can almost only write into memory !

- Second, tracing an application must have a minimum ***impact***. It means that the application must not run significantly slower and must not behave very differently than without the trace: the application must produce the same results and its threads should continue executing in the same order so that problems do not disappear.

3.3.2 Components

Events are written into a buffer. Then a daemon copies them to a binary file.

The basic component of the trace is an **event**. An event shows either a change in an attribute (state, owner, value, ...) of a NPTL object, or the calls (in / out) to any NPTL routine by the application. Sixty events have been defined for the four objects: Threads, Mutex, Barrier, CondVar. About 200 different events are expected to be defined when all routines are traced. As an example, eleven events have been defined for the Thread object: `THREAD_JOIN, _DETACH, _STATE_DEAD, _STATE_WAIT, _STATE_WAKE, _INIT, _CREATE_IN, _CREATE_OUT, _JOIN_IN, _JOIN_OUT, _SET_PD`. Each event is recorded in the buffer with useful data: time-stamp (for computing the elapsed time between two events), process Id, thread Id, and parameters. Events contain various amounts and kinds of data.

Adjacent events are grouped as a **trace point** in order to reduce the impact of the trace mechanism: only one call is done instead of two or more.

A circular **buffer** allocated in shared memory is used for storing the traces. If the buffer is not appropriately sized (too small for a given number of threads and processors), there is a risk of overflow: new traces are written over the oldest traces that the daemon is attempting to copy to the binary file. Buffer overflow is managed and produces a clear message. But its probability is nearly null, as explained hereafter.

A **daemon** is in charge of continuously monitoring the filling rate of the buffer. When a threshold is crossed, the daemon copies the traces to the binary file. One instance of the daemon is launched per application and behaves as the parent process of the application process.

One **binary file** is filled with traces for each traced application. It can be converted to text by means of a decoding tool. And its size can be greater than 2 Gigabytes.

3.3.3 Managing the Buffer

Correctly and efficiently managing the trace buffer was a quite complex task. Since using NPTL objects and routines (mutex) is forbidden, we used the atomic macros provided by the glibc.

We considered several solutions for managing the trace buffer:

- use two buffers: when one is full the buffers are switched and the threads write traces in the other one, enabling the daemon to save the traces to file without blocking the application threads, but with the risk of loosing traces.

- the same, but with blocking the threads and with no risk of loosing traces.
- use one buffer per processor in order to reduce the contention between traces.
- use one buffer per thread, suppressing all contention.
- use one buffer per process launched by the command to be traced.
- use one buffer for all processors, all processes and all threads launched by the command to be traced.

Each of these solutions has drawbacks and benefits about complexity, reliability and performance. We started looking in detail at the last solution. It appeared to be reliable, efficient, and not too complex, based on experiments we made on bi- and quad-processor machines.

The solution is based on the following two mechanisms:

1) When a thread needs to store trace data into the buffer, it first *reserves* the appropriate amount of space by increasing the *reserved* pointer in one **atomic** operation. Then it writes the trace data in the reserved space. And finally it increases the *written* pointer with the amount of written bytes by means of another **atomic** operation. With this approach, the buffer is never locked when threads reserve space and write traces.

2) The daemon continuously monitors the percentage of buffer already filled with traces. When the daemon decides that it is time to save the filled and reserved parts of the buffer, the daemon blocks all threads attempting to reserve more space in the buffer. Once all threads have completed writing events in the buffer (when *written* has reached *reserved*), the daemon releases the threads which restart reserving space in the buffer. Then the daemon writes the filled part of the buffer into the binary file. The goal is not to lose traces.

The figure 4 explains the main steps:

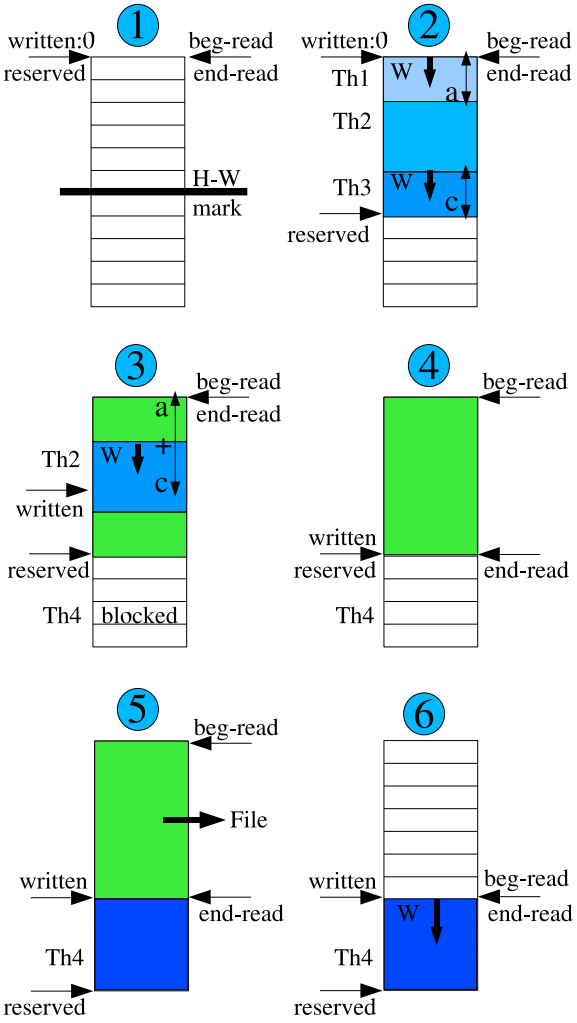

Figure 4: Buffer management

1) Start: No space has been reserved.

2) Threads 1, 2 and 3 have successively reserved the space they need for writing their trace. The reserved space has crossed the High-Water mark: the daemon now blocks the other threads attempting to reserve space. Threads

1 and 3 have started writing the trace data whereas thread 2 has not started yet.

3) Threads 1 and 3 have finished writing: an amount of *a+c* bytes of data has already been written. Thread 2 has started writing. Thread 4 is blocked.

4) Thread 2 has finished writing. Now the written space (*a+c+b*) is equal to the reserved space. The daemon knows which area must be saved to disk: Thread 4 is released.

5) The daemon is writing the trace data to the binary file. Thread 4 has reserved the needed space.

6) The daemon has finished writing the trace data. Thread 4 is writing its trace data.

An overflow may occur when threads write data in the buffer faster than the daemon empties it. Experiments have shown that it may appear only if the buffer is very small (let's say: 1 MB for one fast processor) and if the application is continuously writing traces due to many competing threads. Using a larger buffer is a good solution. By default, the threshold (indicating when it is time to empty the buffer) is set to half the size of the buffer. The size of the buffer for small and medium machines is computed as: $(MemSize * NberOfProcessors)/K$ where $K = 128$ by default. Thus, with a 1GB machine with 2 processors, the size of the buffer is 16 MB. We monitored the maximum usage of the buffer with various applications and the conclusion is that even an unrealistic application designed for writing PTT traces as fast as possible cannot overflow the buffer when K is 64. The applications we used never fill the buffer more than 60% before the daemon empties it. If needed, the user is able to use a more adequate buffer size, as a parameter given to the PTT launcher `ptt-view`.

If an overflow occurs, the threads of the application hang. After a time-out, the application is stopped by the daemon.

Other problems may also occur when a thread is canceled, hangs or dies.

- A thread can be canceled by means of the `pthread_cancel()` routine. The POSIX standard defines that an application can switch from and to two different cancellation modes: asynchronous or deferred (synchronous). In asynchronous mode, the thread can be canceled anywhere (if the cancellation state is *enabled*). In deferred mode, the thread can only be canceled in Cancellation Points.

  In order to guarantee that the trace data written in the buffer are always complete, the execution of the PTT trace mechanism is done in deferred mode (the previous cancellation *mode* is stored and then restored).

- A hang of the application can lead to 2 different cases. If an application thread hangs after it has reserved space in the trace buffer and before it has written its trace data, the daemon saves the last traces after waiting a time-out. If a thread hangs elsewhere, one must kill the application.

- When a thread runs into a Segmentation Fault or receives a kill signal, the daemon is warned and saves the last unsaved traces.

Moreover—as expected—once the application has completed its task and has returned, the daemon saves the last unsaved traces.

In order to simplify the design and to speed up the writing of traces into the buffer, all information to be stored within each event are a multiple of 32 bits.

Using syscalls like `gettimeofday()` to time-stamp the event introduces too much overhead. We must directly read a register of the machine whenever it is possible. This has been done on IA32 by using the TSC register. This will need to be studied for other architectures (PPC, IA64, ...) and for NUMA[4] machines where each node may have its own counter.

3.3.4 Using the patched NPTL

PTT is made of three parts:

- A patch that adds the PTT trace points into the NPTL routines.
- A patch that adds into NPTL the PTT code that writes the traces into the buffer.
- The code of the daemon and the four PTT commands.

PTT is delivered with instructions explaining how a version of NPTL can be patched and compiled. As explained above, no modification or recompilation of the application is required.

There are two cases for using the patched NPTL:

- For simple programs, it is easy to force the library loader to use the appropriate NPTL library. A script is delivered with PTT.
- For complex programs like JVMs, it is a bit more complex. The `java` command acts as a library loader: it looks at `/proc/self/exe` in order to find its path and name, then it loads libraries (`libjava.so`, ...) based on its path, and finally it reloads itself with `execve()`. So one cannot simply use `ld.so`.

  There are 3 solutions:

  1. If your system glibc is the same version as the patched one, then you can use `LD_PRELOAD`.
  2. You can edit the ELF header in order to change the library loader name/path. Not so easy...
  3. Or you can build a *chroot* environment with the patched library as default glibc.

  If the patched NPTL is delivered with a distribution, then the `LD_PRELOAD` solution seems appropriate.

3.3.5 Measures and Performances

We have measured the impact of PTT on several applications: GLucas, Volano™Mark[5] and SPECjbb2000[6] for Java, and an unrealistic program performing only calls to the tracing mechanism. We have also compared the impact of PTT with that of the `strace` command. All results are done with the subset of traced NPTL routines that were available in April: Threads, Mutexes, Barriers and CondVars. This means that the following results are preliminary and will probably be different once PTT is finalized.

On average, one call to the PTT trace mechanism leads to 30 bytes of trace data.

• **GLucas** [5] is an HPC[7] program dedicated to proving the primality of Mersenne numbers ($2^q - 1$). It is an open-source C program that implements a specific FFT[8] by means

[4] Non-Uniform Memory Access

[5] Volano™ is a trademark of Volano LLC. [6]

[6] SPECjbb® is a registered trademark of the Standard Performance Evaluation Corporation (SPEC®). [7]

[7] High Performance Computing

[8] Fast Fourier Transform

of threads. This is a perfect tool for measuring the impact of PTT: its consumption of multi-threading is much higher than a simple *producer-consumer* model, it can be configured to use as many threads as wanted and it can be launched for a variable amount of time,

• **Volano™Mark** [6] was designed for comparing JVMs when used by the Volano™ chat product. It is a pure Java server benchmark characterized by long-lasting network connections and high thread counts. It is an unofficial Java benchmark that can be configured to use many (thousands) threads for exchanging data between one client and one server by means of sockets. It creates client connections in groups of 20 (a *room*). It is a stress Java program which often makes a JVM crash or hang and which has been used by several studies of Linux performances in the past [9].

• **SPECjbb®2000** [7] is an official SPEC Java benchmark simulating a 3-tier system, mainly the middle tier (business logic and object manipulation). It uses a small number of threads (2 to 3 times the number of processors).

We have made measures on a 2x IA32 machine with 2.8 GHz processors. We observed that the maximum throughput before buffer overflow was obtained with the unrealistic application running one thread: ~1,800,000 traces per second. Due to contention, using more threads led to a lower throughput.

When running **GLucas** with 1000 iterations and with small (2×10^6) to medium (16×10^6) values for the exponent q , we measured that the system and user CPU cost of the daemon was negligible, less than 1 $^0/_{00}$ of the CPU time consumed by GLucas. The throughput of traces ranged between 5,000 and 50,000 traces per second: 40 times lower than the maximum.

When running **Volano™**Mark with 10 rooms, the results depended greatly on the JVM. It appeared that the three main JVMs available on ia32 do not use NPTL in the same way (this may also be due to the fact that only a subset of NPTL routines were traced at that time), leading to quite different results. First, the impact of using the patched NPTL with tracing disabled compared to using the original NPTL is nearly negligible: less than 2% with the fastest JVM, and less than $\sim$ 5% with the slowest one. Second, the impact of running the bench with the patched NPTL with full tracing compared to the original NPTL was about 16% with the fastest JVM and about 47% with the slowest one. Leading to a volume of traces (client + server) that depends on the JVM: from 215 MB to 1,000 MB.

When running **SPECjbb®2000** 65 times with 10 warehouses on a bi-processors machine, the impact of PTT could not be measured since it was lower than the precision of the measure.

We used the **strace** tool for tracing Volano™Mark in two ways. First, when tracing all system calls and only the Volano™ server, the performances were divided by 14.6. Second, when tracing only the calls to the `futex` system call and only the client, the performances were divided by 3.2. Although strace and PTT trace different things, this clearly shows that PTT is much lighter than strace.

3.3.6 Testing

PTT is delivered with a set of tests.

First, there are tests verifying that the features provided by PTT work fine. Examples: a program checks that the `fork()` is correctly handled; another one checks in detail concurrent accesses to the buffer; and a program checks that overloading the buffer and the daemon

leads to a nice message warning the end-user that he may loose traces.

Second, there are tests verifying in detail that the traces generated by each patched NPTL routine are correct.

Third, two versions of a *producer-consumer* model have been written, using condvars or semaphores.

GLucas and Java (Volano™Mark) are used for verifying that PTT does not modify the behavior of a large and complex application.

Also the **OPTS** is run in order to check that the PTT-patched NPTL is still compliant with the POSIX Threads standard.

3.4 User Interface

3.4.1 Commands

Several commands are delivered:

`ptt-trace` for launching the application and generating a binary trace file

`ptt-view` for translating the binary trace file into a human or machine readable text format—see Figure 5. (It will enable the end-user to filter the trace. Filters can be applied on: Process Id, Thread Id, name of POSIX Thread Objects, name of Events.)

`ptt-stat` for providing statistics about the use of POSIX Threads objects, etc

`ptt-paje` for translating the binary trace file into a Pajé trace file.

3.4.2 GUI

The analysis of the trace may be very difficult without the help of a graphical tool. Such a tool may simply help the user to navigate through the traces (filter information, find interacting objects, follow the status and the activity of objects, etc.); or it may also display traces in an easier-to-understand graphical way. Both directions are useful, but we decided to focus only on the second one, because we found a sophisticated open-source tool named Pajé that provides nearly all required features without the pain of designing and coding a tool dedicated to PTT.

Pajé [8] was designed for visualizing the traces of a parallel and distributed language (Athapascan) and was developed in a laboratory of the French Research Center IMAG in Grenoble. Pajé is flexible and scalable and can be used quite easily for visualizing the traces of any parallel or distributed system. It can provide views at different scales with different levels of details and one can navigate back and forth in a large file of traces. It is built on the GNUstep [11] platform: an object-oriented framework for desktop application development, based on the OpenStep specification originally created by NeXT—now Apple. Several important companies (France Telecom, ...) have already used Pajé for visualizing complex traces. Pajé is now available in the *sid* (unstable) Debian distribution and soon in the *sarge* (stable) Debian distribution.

We have done preliminary studies and experiments with Pajé, showing that it seems quite easy to produce traces in the format expected by Pajé.

The figure 6 is an example of how a trace could be visualized: the objects (threads, barriers, ...) appear as horizontal bars, with different colors according to their status; and the interactions between objects (when a thread creates or cancels other threads, etc.) are displayed as vertical arrows. The scenario of the example is: the main thread initializes a barrier (count=2) and creates a thread. Then the two threads call

```
Raw machine format:
0.001724:START_USER_FUNC   : 29336 : 0xb7ecb6b0
0.001908:BARRIER_INIT_IN   : 29336 : 0xb7ecb6b0 : 0x8049d28 : (nil) : 2
0.001909:BARRIER_INIT      : 29336 : 0xb7ecb6b0 : 0x8049d28 : 2
0.001909:BARRIER_INIT_OUT  : 29336 : 0xb7ecb6b0 : 0

Text human format:
0.001724 : Pid 29336, Thread 0xb7ecb6b0 starts user function
0.001908 : Pid 29336, Thread 0xb7ecb6b0 enters function pthread_barrier_init.
0.001909 : Pid 29336, Thread 0xb7ecb6b0 initializes barrier 0x8049d28, left=2
0.001909 : Pid 29336, Thread 0xb7ecb6b0 leaves function pthread_barrier_init.
```

Figure 5: An example of a trace written in human or machine readable text formats.

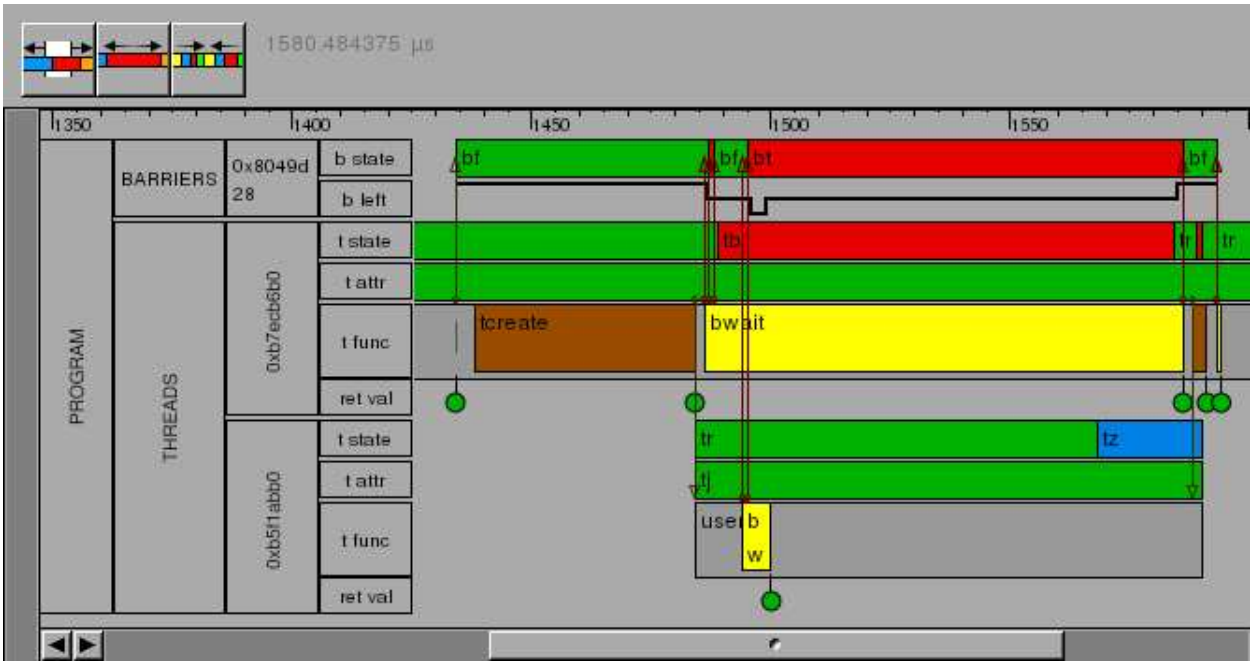

Figure 6: An example of visualizing a PTT trace with Pajé.

`pthread_barrier_wait`: the two threads are freed by the barrier. Finally, the main thread calls `pthread_thread_join` on the second thread and destroys the barrier.

The Pajé tool will enable the user of PTT to clearly see the interactions between the objects involved in his program. Pajé will help the developer of a multi-threaded application to see how his code executes in reality. He will be able to find possible dead-locks, understand which lock is blocking threads thus reducing the performances, and analyze bugs. For analyzing large traces, specific tools (naming, filtering, ...) must be designed and added in order to manage hundreds of objects and millions of events.

3.5 Status & Future work

Two students work on PTT up to mid July this year. Hereafter, we describe: the status of their work end of April; what they plan to provide in mid July; known limitations; and future potential tasks.

3.5.1 Status in April

At the end of April, PTT already provides the following:

- User and Internal documentations are available.
- PTT is quite reliable and efficient.
- A patch is available for the glibc 2.3.4 (and soon for 2.3.5).

The patch and the sources under CVS are available on SourceForge.net [10].

3.5.2 Expected Status in July

At the end of July, PTT should provide the following:

- Be reliable, efficient, and scalable;
- be available on 3 architectures: IA32, PPC, IA64; handle the most important NPTL objects and routines; provide basic filtering;
- and enable use of Pajé for visualizing small and medium volumes of traces.

3.5.3 Known Limitations

In order to know how much time has elapsed between two events, a time-stamp is recorded within each event. Since this time-stamp is obtained before the event space is reserved in the buffer, it may occur that an event appears in the buffer before older events. Although this could be fixed at the time of decoding the binary trace file, we consider that the error is negligible.

Time-stamping the events on NUMA machines: the actual solution does not take into account the time difference that may appear on such machines.

3.5.4 Next Steps

The main concern when tracing multi-threaded applications is to be able to link the information shown by the trace tool with the traced program. Even with only a dozen threads and mutexes, it is not easy for the user to link the traced thread he is looking at through PTT with the thread managed by his code. Being able to give a name to each instance of NPTL objects

is very important. Several ways should be studied and provided in order to replace the internal names (like: `0x401598c0`) by easily understandable names (like: `SocketThread_1`):

- automatically give the thread the name of the routine that was started when the thread was created,
- enable the user to iteratively give names to objects as the user recognizes the objects,
- enable reuse of some existing name table (JVMs).

PTT should be ported on other popular architectures. On machines using several time counters, like NUMA machines, the current version would deliver dates that sometimes could lead to mistakes. This needs to be solved.

Optimizations should be studied: manage the buffer differently; reduce the amount of data stored with each event. More work must be done in order to check the usability and scalability of PTT when used with big and complex applications on large machines with many and fast processors.

We expect people facing complex problems with multi-threaded applications to experiment with PTT, in order to find and fix remaining bugs, and to provide requirements for new features making PTT easier to use and more productive.

PTT could also be a basis for dynamically checking if the application is compliant with the POSIX Thread standard. It is so easy not to fulfill all the constraints of the standard.

The next step of the project is to prove that PTT is really a useful tool: it shortens the time needed for understanding a multi-threaded problem, it speeds up the work of Linux or Java support teams, and it simplifies the analysis of the behavior of NPTL when a misfunction is suspected.

Then PTT could be integrated into Linux Distros. The final goal is to have PTT accepted by the community and then integrated into the GNU libc.

3.5.5 Contributors

PTT has been designed by Sébastien Decugis, Mayeul Marguet, Tony Reix and the developers. The developers are: Nadège Griesser (ENSIMAG-Telecom, Grenoble), Laetitia Kameni-Djinou (UTC, Paris) and Matthieu Castet (ENSIMAG, Grenoble).

4 Conclusion

As we demonstrated in this document, our project has completed some of its objectives, but more work remains pending.

Our testing effort is not complete yet. We have tested only 42 functions of the 150 NPTL contains. Some of the remaining functions may contain bugs or at least are worth testing deeply. The remaining domains are *read-write locks*, *barriers*, *spinlocks*, *thread-specific data*, *timers*, and *message queues*.

Anticipating future problems by writing test cases before someone runs into a bug usually saves a lot of money for everybody. For this reason, we're calling for volunteers to continue our work and complete the testing. This work shall be a continued effort, because the POSIX Standard is changing regularly, therefore if the test suite is not updated regularly it will be deprecated sooner or later. To avoid this situation

for the OPTS, the best bet is to have many people use it.

The targeted users are mostly developers of POSIX-compliant implementations. Automating the use of OPTS is easy and, thanks to the TSLogParser tool, collecting and analyzing the results is also quite simple. The next step towards quality for NPTL is to have a real testing process integrated into its development cycle.

The glibc addition to the STP project may be a good solution to solve this, as it is already used for the kernel development and has proved to be useful by detecting new bugs very early in the process.

As we've already told about our Trace Tool, we need more beta testers to try it and give us their comments. This way, we should be able to develop smart tools to use the traces, for example by parsing them in order to find possible problems in threads synchronization or locks contention.

We will also be able to propose our tool to distribution makers, the final goal being that this trace tool be present on all systems. This way, debugging and profiling multi-threaded software will be much easier than it is currently. Is it a utopia? We don't think so...

References

[1] Single UNIX® Specification:
`http://www.unix.org/single_unix_specification/`

[2] NPTL Stabilization:
`http://nptl.bullopensource.org/`

[3] Austin Revision Group:
`http://www.opengroup.org/austin/`

[4] TSLogParser project:
`http://tslogparser.sourceforge.net/`

[5] GLucas:
`http://www.oxixares.com/glucas/`

[6] Volano™Mark:
`http://www.volano.com/`

[7] SPECjbb®2000:
`http://www.spec.org/jbb2000/`

[8] Pajé homepage:
`http://forge.objectweb.org/projects/paje/`

[9] Linux Kernel Performance Measurement and Evaluation (IBM):
`http://linuxperf.sourceforge.net/lwesf-duc_vianney-chat.pdf`

[10] PTT on SourceForge.net:
`http://sourceforge.net/projects/nptltracetool/`

[11] GNUstep project:
`http://www.gnustep.org/`

[glibc] GNU Lib C:
`http://www.gnu.org/software/libc/libc.html`

[OPTS] Open POSIX Test Suite:
`http://posixtest.sourceforge.net/`

[LTP] Linux Test Project:
`http://ltp.sourceforge.net/`

[STP] Scalable Test Platform:
`http://www.osdl.org/lab_activities/kernel_testing/stp`

[PLM] Patch Lifecycle Manager:
http://www.osdl.org/plm-cgi/plm

[LTT] Linux Trace Toolkit:
http://www.opersys.com/LTT/

[PTT] PTT (NPTL Traces) project:
http://nptltracetool.sourceforge.net/

Networking Driver Performance and Measurement - e1000 A Case Study

John A. Ronciak
Intel Corporation
john.ronciak@intel.com

Jesse Brandeburg
Intel Corporation
jesse.brandeburg@intel.com

Ganesh Venkatesan
Intel Corporation
ganesh.venkatesan@intel.com

Mitch Williams
Intel Corporation
mitch.a.williams@intel.com

Abstract

Networking performance is a popular topic in Linux and is becoming more critical for achieving good overall system performance. This paper takes a look at what was done in the e1000 driver to improve performance by (a) increasing throughput and (b) reducing of CPU utilization. A lot of work has gone into the e1000 Ethernet driver as well into the PRO/1000 Gigabit Ethernet hardware in regard to both of these performance attributes. This paper covers the major things that were done to both the driver and to the hardware to improve many of the aspects of Ethernet network performance. The paper covers performance improvements due to the contribution from the Linux community and from the Intel group responsible for both the driver and hardware. The paper describes optimizations to improve small packet performance for applications like packet routers, VoIP, etc. and those for standard and jumbo packets and how those modifications differs from the small packet optimizations. A discussion on the tools and utilities used to measure performance and ideas for other tools that could help to measure performance are presented. Some of the ideas may require help from the community for refinement and implementation.

Introduction

This paper will recount the history of e1000 Ethernet device driver regarding performance. The e1000 driver has a long history which includes numerous performance enhancements which occurred over the years. It also shows how the Linux community has been involved with trying to enhance the drivers' performance. The notable ones will be called out along with when new hardware features became available. The paper will also point out where more work is needed in regard to performance testing. There are lots of views on how to measure network performance. For various reasons we have had to use an expensive, closed source test tool to measure the network performance for the driver. We would like to engage with the Linux community to try to address this and come up with a strategy of having an open source measurement tool along with consistant testing methods.

This paper also identifies issues with the system and the stack that hinder performance. The performance data also indicates that there is room for improvement.

A brief history of the e1000 driver

The first generation of the Intel® PRO/1000 controllers demonstrated the limitation of the 32-bit 33MHz PCI bus. The controllers were able to saturate the bus causing slow response times for other devices in the system (like slow video updates). To work with this PCI bus bandwidth limitation, the driver team worked on identifying and eliminating inefficiencies. One of the first improvements we made was to try to reduce the number of DMA transactions across the PCI bus. This was done using some creative buffer coalescing of smaller fragments into larger ones. In some cases this was a dramatic change in the behavior of the controller on the system. This of course was a long time ago and the systems, both hardware and OS have changed considerably since then.

The next generation of the controller was a 64-bit 66MHz controller which definitely helped the overall performance. The throughput increased and the CPU utilization decreased just due to the bus restrictions being lifted. This was also when new offload features were being introduced into the OS. It was the first time that interrupt moderation was implemented. This implementation was fairly crude, based on a timer mechanism with a hard time-out time set, but it did work in different cases to decrease CPU utilization.

Then a number of different features like descriptor alignment to cache lines, a dynamic inter-frame gap mechanism and jumbo frames were introduced. The use of jumbo frames really helped transferring large amounts of data but did nothing to help normal or small sized frames. It also was a feature which required network infrastructure changes to be able to use, e.g. changes to switches and routers to support jumbo frames. Jumbo frames also required the system stacks to change. This took some time to get the issues all worked out but they do work well for certain environments. When used in single subnet LANs or clusters, jumbo frames work well.

Next came the more interesting offload of checksumming for TCP and IP. The IP offload didn't help much as it is only a checksum across twenty bytes of IP header. However, the TCP checksum offload really did show some performance increases and is widely used today. This came with little change to the stack to support it. The stack interface was designed with the flexibility for a feature like this. Kudos to the developers that worked on the stack back then.

NAPI was introduced by Jamal Hadi, Robert Olsson, et al at this time. The e1000 driver was one of the first drivers to support NAPI. It is still used as an example of how a driver should support NAPI. At first the development team was unconvinced that NAPI would give us much of a benefit in the general test case. The performance benefits were only expected for some edge case situations. As NAPI and our driver matured however, NAPI has shown to be a great performance booster in almost all cases. This will be shown in the performance data presented later in this paper.

Some of the last features to be added were TCP Segment Offload (TSO) and UDP fragment checksums. TSO took work from the stack maintainers as well as the e1000 development team to get implemented. This work continues as all the issues around using this have not yet been resolved. There was even a rewrite of the implementation which is currently under test (Dave Miller's TSO rewrite). The UDP

fragment checksum feature is another that required no change in the stack. It is however little used due to the lack of use of UDP checksumming.

The use of PCI Express has also helped to reduce the bottleneck seen with the PCI bus. The significantly larger data bandwidth of PCIe helps overcome limitations due to latencies/overheads compared to PCI/PCI-X buses. This will continue to get better as devices support more lanes on the PCI Express bus further reducing bandwidth bottlenecks.

There is a new initiative called Intel® I/O Acceleration Technology (I/OAT) which achieves the benefits of TCP Offload Engines (TOE) without any of the associated disadvantages. Analysis of where the packet processing cycles are spent was performed and features designed to help accelerate the packet processing. These features will be showing up over the next six to nine months. The features include Receive Side Scaling (RSS), Packet Split and Chipset DMA. Please see the [Leech/Grover] paper "Accelerating Network Receive Processing: Intel® I/O Acceleration Technolgy" presented here at the symposium. RSS is a feature which identifies TCP flows and passes this information to the driver via a hash value. This allows packets associated with a particular flow to be placed onto a certain queue for processing. The feature also includes multiple receive queues which are used to distribute the packet processing onto multiple CPUs. The packet split feature splits the protocol header in a packet from the payload data and places each into different buffers. This allows for the payload data buffers to be page-aligned and for the protocol headers to be placed into small buffers which can easily be cached to prevent cache thrash. All of these features are designed to reduce or eliminate the need for TOE. The main reason for this is that all of the I/OAT features will scale with processors and chipset technologies.

Performance

As stated above the definition of performance varies depending on the user. There are a lot of different ways and methods to test and measure network driver performance. There are basically two elements of performance that need to be looked at, throughput and CPU utilization. Also, in the case of small packet performance, where packet latency is important, the packet rate measured in packets per second is used as a third type of measurement. Throughput does a poor job of quantifying performance in this case.

One of the problems that exists regarding performance measurements is which tools should be used to measure the performance. Since there is no consistent open source tool, we use a closed source expensive tool. This is mostly a demand from our customers who want to be able to measure and compare the performance of the Intel hardware against other vendors on different Operating Systems. This tool, IxChariot by IXIA[1], is used for this reason. It does a good job of measuring throughput with lots of different types of traffic and loads but still does not do a good job of measuring CPU utilization. It also has the advantage that there are endpoints for a lot of different OSes. This gives you the ability to compare performance of different OSes using the same system and hardware. It would be nice to have and Open Source tool which could do the same thing. This is discussed in Section , "Where do we go from here."

There is an open source tool which can be used to test small packet performance. The tool is the packet generator or 'pktgen' and is a kernel module which is part of the Linux kernel. The tool is very useful for sending lots of packets with set timings. It is the tool of choice for

[1] Other brands and names may be claimed as the property of others.

anyone testing routing performance and routing configurations.

All of the data for this section was collected using Chariot on the same platform to reduce the number of variables to control except as noted.

The test platform specifications are:

- Blade Server
- Dual 2.8GHz Pentium® 4 Xeon™ CPUs, 512KB cache 1GB RAM
- Hyperthreading disabled
- Intel® 80546EB LAN-on-motherboard (PCI-X bus)
- Competition Network Interface Card in a PCI-X slot

The client platform specifications are:

- Dell[2] PowerEdge® 1550/1266
- Dual 1266MHz Pentium® III CPUs, 512KB cache, 1GB RAM,
- Red Hat[2] Enterprise Linux 3 with 2.4.20-8smp kernel,
- Intel® PRO/1000 adapters

Comparison of Different Driver Versions

The driver performance is compared for a number of different e1000 driver versions on the same OS version and the same hardware. The difference in performance seen in Figure 1 was due to the NAPI bug that Linux community found. It turns out that the bug was there for a long time and nobody noticed it. The bug was causing the driver to exit NAPI mode back into interrupt mode fairly often instead of staying in NAPI mode. Once corrected the number of interrupts taken was greatly reduced as it should be when using NAPI.

Comparison of Different Frames Sizes verses the Competition

Frame size has a lot to do with performance. Figure 2 shows the performance based on frame size against the competition. As the chart shows, frame size has a lot to do with the total throughput that can be reached as well as the needed CPU utilization. The frame sizes used were normal 1500 byte frames, 256 bytes frames and 9Kbyte jumbo frames.

NOTE: The competition could not accept a 256 byte MTU so 512 bytes were used for performance numbers for small packets.

Comparison of OS Versions

Figure 3 shows the performance comparison between OS versions including some different options for a specific kernel version. There was no reason why that version was picked other than it was the latest at the time of the tests. As can be seen from the chart in Figure 3, the 2.4 kernels performed better overall for pure throughput. This means that there is more improvement to be had with the 2.6 kernel for network performance. There is already new work on the TSO code within the stack which may have improved the performance already as the TSO code has been known to hurt the overall network throughput. The 2.4 kernels do not have TSO which could be accounting for at least some of the performance differences.

[2]Other brands and names may be claimed as the property of others.

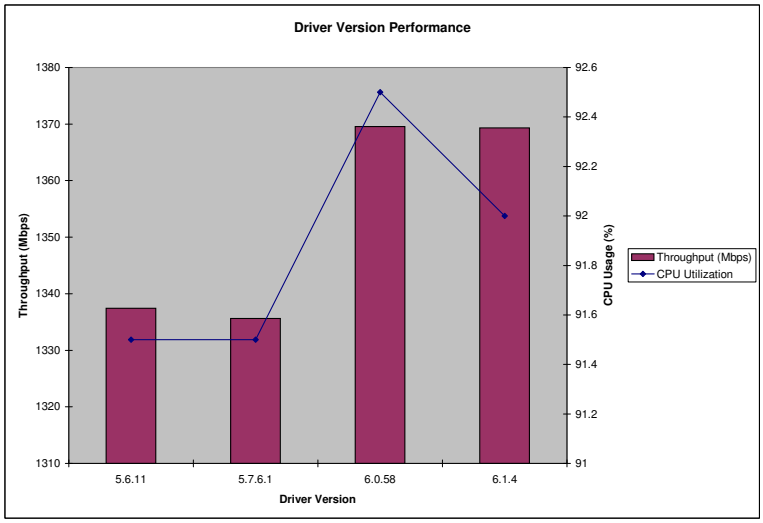

Figure 1: Different Driver Version Comparison

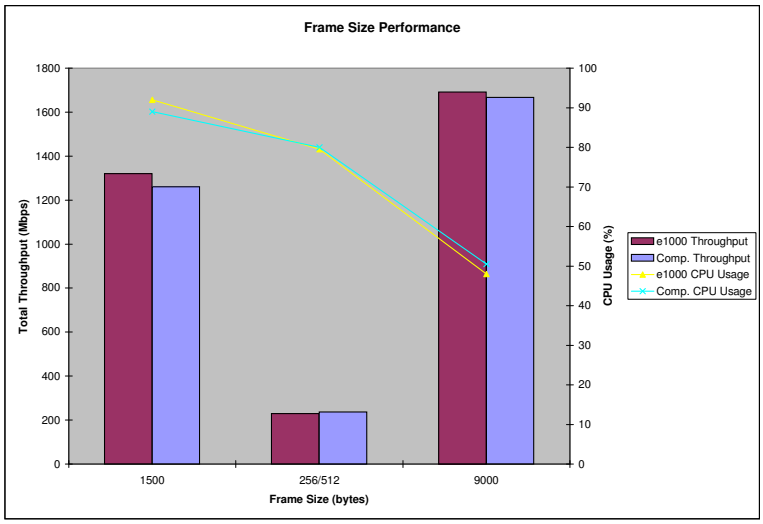

Figure 2: Frame Size Performance Against the Competition

Results of Tuning NAPI Parameters

Initial testing showed that with default NAPI settings, many packets were being dropped on receive due to lack of buffers. It also showed that TSO was being used only rarely (TSO was not being used by the stack to transmit).

It was also discovered that reducing the driver's weight setting from the default of 64 would eliminate the problem of dropped packets. Further reduction of the weight value, even to very small values, would continue to increase throughput. This is shown in Figure 4.

The explanation for these dropped packets is simple, because the weight is smaller, the driver iterates through its packet receive loop (in `e1000_clean_rx_irq`) fewer times, and hence writes the Receive Descriptor Tail regis-

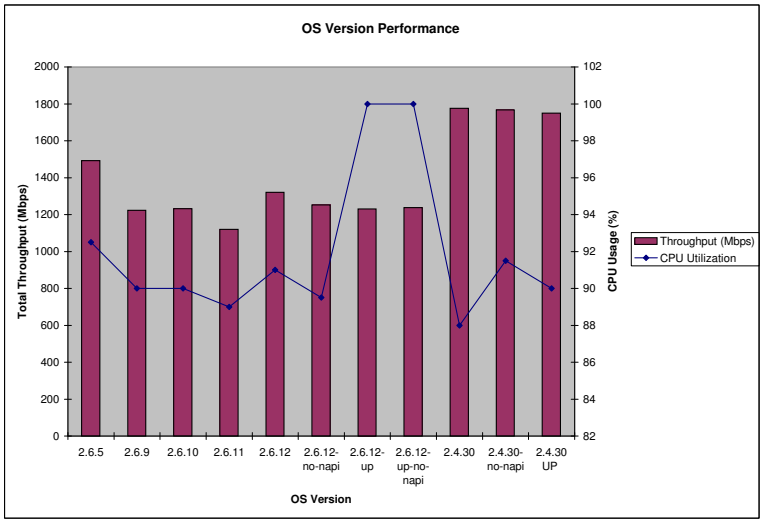

Figure 3: OS Version Performance

ter more often. This notifies the hardware that descriptors are available more often and eliminates the dropped packets.

It would be obvious to conclude that the increase in throughput can be explained by the dropped packets, but this turns out to not be the case. Indeed, one can eliminate dropped packets by reducing the weight down to 32, but the real increase in throughput doesn't come until you reduce it further to 16.

The answer appears to be latency. With the higher weights, the NAPI polling loop runs longer, which prevents the stack from running its own timers. With lower weights, the stack runs more often, and processes packets more often.

We also found two situations where NAPI doesn't do very well compared to normal interrupt mode. These are 1) when the NAPI poll time is too fast (less than time it takes to get a packet off the wire) and 2) when the processor is very fast and I/O bus is relatively slow. In both of these cases the driver keeps entering NAPI mode, then dropping back to interrupt mode since it looks like there is no work to do. This is a bad situation to get into as the driver has to take a very high number of interrupts to get the work done. Both of these situations need to be avoided and possibly have a different NAPI tuning parameter to set a minimum poll time. It could even be calculated and used dynamically over time.

Where the Community Helped

The Linux community has been very helpful over the years with getting fixes back to correct errors or to enhance performance. Most recently, Robert Olsson discovered the NAPI bug discussed earlier. This is just one of countless fixes that have come in over the years to make the driver faster and more stable. Thanks to all to have helped this effort.

Another area of performance that was helped by the Linux community was the e1000 small packet performance. There were a lot of comments/discussions in netdev that helped to get the driver to perform better with small packets. Again, some of the key ideas came from Robert

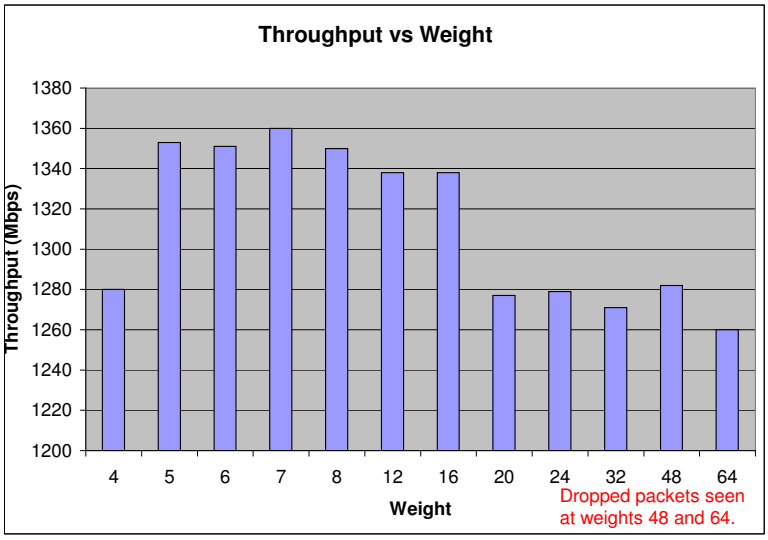

Figure 4: NAPI Tuning Performance Results

Olsson with the work he has done on packet routing. We also added different hardware features over the years to improve small packet performance. Our new RSS feature should help this as well since the hardware will be better able to scale with the number of processor in the system. It is important to note that e1000 benefitted a lot from interaction with the Open Source Community.

Where do we go from here

There are a number of different things that the community could help with. A test tool which can be used to measure performance across OS versions is needed. This will help in comparing performance under different OSes, different network controllers and even different versions of the same driver. The tool needs to be able to use all packet sizes and OS or driver features.

Another issue that should be addressed is the NAPI tuning as pointed out above. There are cases where NAPI actually hurts performance but with the correct tuning works much better.

Support the new I/OAT features which give most if not all the same benefits as TOE without the limitations and drawbacks. There are some kernel changes that need to be implemented to be able to support features like this and we would like for the Linux community to be involved in that work.

Conclusions

More work needs to be done to help the network performance get better on the 2.6 kernel. This won't happen overnight but will be a continuing process. It will get better with work from all of us. Also, work should continue to make NAPI work better in all cases. If it's in your business or personal interest to have better network performance, then it's up to you help make it better.

Thanks to all who have helped make everything perform better. Let us keep up the good work.

References

[Leech/Grover] Accelerating Network Receive Processing: Intel® I/O Acceleration Technolgy; Ottawa Linux Symposium 2005

nfsim: Untested code is buggy code

Rusty Russell
IBM Australia, OzLabs
rusty@rustcorp.com.au

Jeremy Kerr
IBM Australia, OzLabs
jk@ozlabs.org

Abstract

The netfilter simulation environment (`nfsim`) allows netfilter developers to build, run, and test their code without having to touch a real network, or having superuser privileges. On top of this, we have built a regression testsuite for netfilter and iptables.

Nfsim provides an emulated kernel environment in userspace, with a simulated IPv4 stack, as well as enhanced versions of standard kernel primitives such as locking and a proc filesystem. The kernel code is imported into the `nfsim` environment, and run as a userspace application with a scriptable command-line interface which can load and unload modules, add a route, inject packets, run iptables, control time, inspect `/proc`, and so forth.

More importantly we can test every single permutation of external failures automatically—for example, packet drops, kmalloc failures and timer deletion races. This makes it possible to check error paths that very rarely happen in real life.

This paper will discuss some of our experiences with `nfsim` and the progression of the netfilter testsuite as new features became available in the simulator, and the amazing effect on development. We will also show the techniques we used for exhaustive testing, and why these should be a part of every project.

1 Testing Netfilter Code

The netfilter code is complicated. Technically, netfilter is just the packet-interception and mangling framework implemented in each network protocol (IPv4, IPv6, ARP, Decnet and bridging code)[3]. IPv4 is the most complete implementation, with packet filtering, connection tracking and full dynamic Network Address Translation (NAT). Each of these, in turn, is extensible: dozens of modules exist within the tree to filter on different packet features, track different protocols, and perform NAT.

There were several occasions where code changes unintentionally broke extensions, and other times where large changes in the networking layer (such as non-linear `skb`s[1]) caused subtle bugs. Network testing which relies on users is generally poor, because no single user makes use of all the extensions, and intermittent network problems are never reported because users simply hit “Reload” to work around any problem. As an example, the Linux 2.2 masquerade code would fail on one in a thousand FTP connections, due to a control message being split over two packets. This was never reported.

[1] `skb`s are the kernel representation of network packets, and do not need to be in contiguous virtual memory.

2 The Existing Netfilter Testsuite

Netfilter had a testsuite from its early development. This testsuite used the ethertap[2] devices along with a set of helper programs; the tests themselves consisted of a series of shell scripts as shown in Figure 1.

Unfortunately, this kind of testing requires root privileges, a quiescent machine (no `ssh`-ing in to run the testsuite!) and a knowledge of shell slightly beyond cut-and-paste of other tests. The result was that the testsuite bit-rotted, and was no longer maintained after 2000.

2.1 Lack of Testing

The lack of thorough testing had pervasive effects on the netfilter project which only became clear as the lack was remedied. Most obviously, the quality of the code was not all that it could have been—the core functionality was solid, but the fringes contained longstanding and subtle bugs.

The less-noticed effect is the fear this knowledge induces in the developers: rewrites such as TCP window tracking take years to enter the kernel as the developers seek to slowly add users to test functionality. The result is a cycle of stagnation and patch backlog, followed by resignation and a lurch forward in functionality. It's also difficult to assess test coverage: whether users are actually running the changed code at all.

Various hairy parts of the NAT code had not been significantly altered since the initial implementation five years ago, and there are few developers who actually understand it: one of these, Krisztian Kovacs, found a nasty, previously unnoticed bug in 2004. This discovery caused Rusty to revisit this code, which in turn prompted the development of `nfsim`.

[2] ethertap devices are virtual network interfaces that allow userspace programs to inject packets into the network stack.

3 Testsuite Requirements

There are several requirements for a good testsuite here:

- It must be trivial to run, to encourage developers and others to run it regularly;
- It must be easy to write new tests, so non-core developers can contribute to testing efforts;
- It must be written in a language the developers understand, so they can extend it as necessary;
- It must have reasonable and measurable coverage;
- It should encourage use of modern debugging tools such as valgrind; and
- It must make developers *want* to use it.

4 The New Testsuite—nfsim

It was a long time before the authors had the opportunity to write a new testsuite. The aim of `nfsim` was to provide a userspace environment to import netfilter code (from a standard kernel tree) into, which can then be built and run as a standalone application. A command-line interface is given to allow events to be simulated in the kernel environment. For example:

- generate a packet (either from a device or the local network stack); or

```
tools/intercept PRE_ROUTING DROP 2 1 > $TMPFILE &
sleep 1

tools/gen_ip $TAP0NET.2 $TAP1NET.2 100 1 8 0 55 57 > /dev/tap0

if wait %tools/intercept; then :
else
  echo Intercept failed:
  tools/rcv_ip 1 1 < $TMPFILE
  exit 1
fi
```

Figure 1: Shell-script style test for old netfilter testsuite

- advance the system time; or
- inspect the kernel state (e.g., through the `/proc/` file system).

Upon this we can build a simple testsuite.

Figure 2 shows a simple configure-build-execute session of `nfsim`.

Help text is automatically generated from docbook XML comments in the source, which also form the man page and printable documentation. There is a "trivial" XML stripper which allows building if the required XML tools are not installed.

When the simulator is started, it has a default network setup consisting of a loopback interface and two ethernet interfaces on separate networks. This basic setup allows for the majority of testing scenarios, but can be easily reconfigured. Figure 3 shows the default network setup as shown by the `ifconfig` command.

The presence of this default network configuration was a decision of convenience over abstraction. It would be possible to have no interfaces configured at startup, but this would require each test to initialise the required network environment manually before running. From further experience, we have found that the significant majority of tests do not need to alter the default network setup.

Although the simulator can be used interactively, running predefined `nfsim` test scripts allows us to automate the testing process. At present, a netfilter regression testsuite is being developed in the main netfilter subversion repository.

To assist in automated testing, the builtin `expect` command allows us to expect a string to be matched in the output of a specific command that is to be executed. For example, the command:

```
expect gen_ip rcv:eth0
```

will expect the string "`rcv:eth0`" to be present in the output the next time that the `gen_ip` command (used to generate IPv4 packets) is invoked. If the expectation fails, the simulator will exit with a non-zero exit status. Figure 4 shows a simple `nfsim` test which generates a packet destined for an interface on the simulated machine, and fails if the packet is not seen entering and leaving the network stack.

```
$ ./configure --kerneldir=/home/rusty/devel/kernel/linux-2.6.12-rc4/
...
$ make
...
$ ./simulator --no-modules
core_init() completed
nfsim 0.2, Copyright (C) 2004 Jeremy Kerr, Rusty Russell
Nfsim comes with ABSOLUTELY NO WARRANTY; see COPYING.
This is free software, and you are welcome to redistribute
it under certain conditions; see COPYING for details.
initialisation done
> quit
$
```

Figure 2: Building and running `nfsim`

Note that there's a helpful `test-kernel-source` script in the `nfsim-testsuite/` directory. Given the source directory of a Linux kernel, builds `nfsim` for that kernel and runs all the tests. It has a simple caching system to avoid rebuilding `nfsim` unnecessarily.

During early development, a few benefits of `nfsim` appeared.

Firstly, compared to a complete kernel, build time was very short. Aside from the code under test, the only additional compilation involved the (relatively small) simulation environment.

Secondly, 'boot time' is great:

```
$ time ./simulator < /dev/null
real    0m0.006s
user    0m0.003s
sys     0m0.002s
```

4.1 The Simulation Environment

As more (simulated) functionality is required by netfilter modules, we needed to "bring in" more code from the kernel, which in turn depends on further code, leading to a large amount of dependencies. We needed to decide which code was simulated (reimplemented in `nfsim`), and which was imported from the kernel tree.

Reimplementing functionality in the simulator gives us more control over the "kernel." For example, by using simulated notifier lists, we are able to account for each register and deregister on all notifier chains, and detect mismatches. The drawback of reimplementation is that more `nfsim` code needs to be maintained; if the kernel's API changes, we need to update our local copy too. We also need to ensure that any behavioural differences between the real and simulated code do not cause incorrect test results.

Importing code allows us to bring in functionality 'for free,' and ensures that the imported functionality will mirror that of the kernel. However, the imported code will often require support in another area, meaning that further functionality will need to be imported or reimplemented.

For example, we were faced with the decision to either import or reimplement the IPv4 routing code. Importing would guarantee that we would deal with the 'real thing' when it came

```
> ifconfig
lo
        addr: 127.0.0.1  mask: 255.0.0.0  bcast: 127.255.255.255
        RX packets: 0 bytes: 0
        TX packets: 0 bytes: 0

eth0
        addr: 192.168.0.1  mask: 255.255.255.0  bcast: 192.168.0.255
        RX packets: 0 bytes: 0
        TX packets: 0 bytes: 0

eth1
        addr: 192.168.1.1  mask: 255.255.255.0  bcast: 192.168.1.255
        RX packets: 0 bytes: 0
        TX packets: 0 bytes: 0
```

Figure 3: Default network configuration of `nfsim`

```
# packet to local interface
expect gen_ip rcv:eth0
expect gen_ip send:LOCAL {IPv4 192.168.0.2 192.168.0.1 0 17 3 4}
gen_ip IF=eth0 192.168.0.2 192.168.0.1 0 udp 3 4
```

Figure 4: A simple `nfsim` test

time to test, but required a myriad of other components to be able to import. We decided to reimplement a (very simple) routing system, having the additional benefit of increased control over the routing tables and cache.

Generic functions, or functions strongly tied to kernel code were reimplemented. We have a single `kernelenv/kernelenv.c` file, which defines primitives such as `kmalloc()`, locking functions and lists. The kernel environment contains around 1100 lines of code.

IPv4 code is implemented in a separate module, with the intention of making a 'pluggable protocol' structure, with an IPv6 implementation following. The IPv4 module contains around 1700 lines of code, the majority being in routing and checksum functions.

Ideally, the simulation environment should be as clean and simple as possible; adding complexity here may cause problems when the code under test fails.

4.2 Interaction with Userspace Utilities

The most often-used method of interacting with netfilter code is through the `iptables` command, run from userspace. We needed some way of providing this interface, without either modifying iptables, or reimplementing it in the simulator.

To allow iptables to interface with netfilter code under test, we've developed a shared library, to be `LD_PRELOAD`-ed when running an unmodified `iptables` binary. The shared library intercepts calls to `{set,get}sockopt()`, and diverts these calls to the simulator.

4.3 Exhaustive Error Testing

During netfilter testing with an early version of `nfsim`, it became apparent that almost all of the error-handling code was not being exercised. A trivial example from `ip_tables.c`:

```
counters = vmalloc(countersize);
if (counters == NULL)
        return -ENOMEM;
```

Because we do not usually see out-of-memory problems in the simulator (nor while running in the kernel), the error path (where `counters` is `NULL`) will almost certainly never be tested.

In order to test this error, we need the `vmalloc()` to fail; other possible failures may be due to any number of possible external conditions when calling these 'risky' functions (such as `copy_{to,from}_user()`, semaphores or skb helpers).

Ideally, we would be able to simulate the failure of these risky functions in every possible combination.

One approach we considered is to save the state of the simulation when we reach a point of failure, test one case (perhaps the failure), restore to the previous state, then test the other case (success). This left us with the task of having to implement checkpointing to save the simulator state; while not impossible, it would have been a large amount of work.

The method we implemented is based on `fork()`. When we reach a risky function, we `fork()` the simulator, test the error case in the child process, and the normal case in the parent. This produces a binary tree of processes, with each node representing a risky function. To prevent an explosion in the number of processes, the parent blocks (in `wait()`) while the child executes[3].

The failure decision points are handled by a function named `should_i_fail()`. This handles the process creation, error testing and failure-path replay; the return value indicates whether or not the calling function should fail. Figure 5 shows the `nfsim` implementation of `vmalloc`, an example of a function that is prone to failure. The single (string) argument to `should_i_fail()` is unique per call site, and allows `nfsim` to track and replay failure patterns.

The placement of `should_i_fail()` calls needs to be carefully considered—while each failure test will increase test coverage, it can potentially double the test execution time. To prevent combinatorial increase in simulation time, `nfsim` also has a `should_i_fail_once()` function, which will test the failure case once only. We have used this for functions whose failure does not necessarily indicate an error, for example `try_module_get()`.

When performing this exhaustive error testing, we cannot expect a successful result from the test script; if we are deliberately failing a memory allocation, it is unreasonable to expect that the code will handle this without problems. Therefore, when running these failure tests, we don't require a successful test result, only that the code will handle the failure gracefully (and not cause a segmentation fault, for example). Running the simulator under valgrind[1] can be useful in this situation.

If a certain failure pattern causes unexpected problems, the sequence of failures is printed to allow the developer to trace the pattern, and can be replayed using the `--failpath`

[3] Although it would be possible to implement parallel failure testing on SMP machines by allowing a bounded number of concurrent processes.

```
struct sk_buff *alloc_skb(unsigned int size, int priority)
{
        if (should_i_fail(__func__))
                return NULL;

        return nfsim_skb(size);
}
```

Figure 5: Example of a risky function in `nfsim`

command-line option, running under valgrind or a debugger.

One problem that we encountered was the use of iptables while in exhaustive failure testing mode: we need to be able to `fork()` while interacting with iptables, but can not allow both resulting processes to continue to use the single iptables process. We have solved this by recording all interactions with iptables up until the `fork()`. When it comes time to execute the second case, a new iptables process is invoked, and we replay the recorded session. However, we intend to replace this with a system that causes the iptables process to fork with the simulator.

Additionally, the failure testing is very time-consuming. A full failure test of the 2.6.11 netfilter code takes 44 minutes on a 1.7GHz x86 machine, as opposed to 5 seconds when running without failure testing.

At present, the netfilter testsuite exercises 61% of the netfilter code, and 65% when running with exhaustive error checking. Although the increase in coverage is not large, we are now able to test small parts of code which are very difficult to reliably test in a running kernel. This found a number of long-standing failure-path bugs.

4.4 Benefits of Testing in Userspace

Because `nfsim` allows us to execute kernel code in userspace, we have access to a number of tools that aren't generally available for kernel development. We have been able to expose a few bugs by running `nfsim` under valgrind.

The GNU Coverage tool, `gcov`[2], has allowed us to find untested areas of netfilter code; this has been helpful to find which areas need attention when writing new tests.

Andrew Trigell's `talloc` library[4] gives us clean memory allocation routines, and allows for leak-checking in kernel allocations. The 'contexts' that `talloc` uses allows developers to identify the source of a memory leak.

5 Wider Kernel Testing: `kernsim`?

The `nfsim` technique could be usefully applied to other parts of the kernel to allow a Linux kernel testsuite to be developed, and speed quality kernel development. The Linux kernel is quite modular, and so this approach which worked so well for netfilter could work well for other sections of the kernel.

Currently `nfsim` is divided into `kernelenv`, `ipv4` and the `netfilter` (IPv4) code. The

first two are nfsim-specific testing implementations of the kernel equivalents, such as `kmalloc` and `spin_lock`. The latter is transplanted directly from the kernel source.

The design of a more complete `kernsim` would begin with dividing the kernel into other subsystems. Some divisions are obvious, such as the SCSI layer and VFS layer. Others are less obvious: the slab allocator, the IPv4 routing code, and the IPv4 socket layer are all potential subsystems. Subsystems can require other subsystems, for example the IPv4 socket layer requires the slab allocator and the IPv4 routing code.

For most of these subsystems, a simulated version of the subsystem needs to be written, which is a simplified canonical implementation, and contains additional sanity checks. A good example in `nfsim` is the packet generator which always generates maximally nonlinear `skb`s. A configuration language similar to the Linux kernel 'Kconfig' configuration system would then be used to select whether each subsystem should be the simulator version or imported from the kernel source. This allows testing of both the independent pieces and the combinations of pieces. The latter is required because the simulator implementations will necessarily be simplified.

The current `nfsim` commands are very network-oriented: they will require significant expansion, and probably introduction of a namespace of some kind to prevent overload.

5.1 Benefits of a `kernsim`

It is obvious to the `nfsim` authors that wider automated testing would help speed and smooth the continual redevelopment which occurs in the Linux kernel. It is not clear that the Linux developers' antipathy to testing can be overcome, however, so the burden of maintaining a `kernsim` would fall on a small external group of developers, rather than being included in the kernel source in a series of `test/` subdirectories.

There are other possibilities, including the suggestion by Andrew Tridgell that a host kernel helper could allow development of simple device drivers within `kernsim`. The potential for near-exhaustive testing of device drivers, including failure paths, against real devices is significant; including a simulator subsystem inside `kernsim` would make it even more attractive, allowing everyone to test the code.

6 Lessons Learnt from `nfsim`

Nfsim has proven to be a valuable tool for easy testing of the complex netfilter system. By providing an easy-to-run testsuite, we have been able to speed up development of new components, and increase developer confidence when making changes to existing functionality. Netfilter developers can now be more certain of any bugfixes, and avoid inadvertent regressions in related areas.

Unfortunately, persuading some developers to use a new tool has been more difficult than expected; we sometimes see testsuite failures with new versions of the Linux kernel. However, we are confident that `nfsim` will be adopted by a wider community to improve the quality of netfilter code. Ideally we will see almost all of the netfilter code covered by a nfsim test some time in the near future.

Adopting the simulation approach to testing is something that we hope other Linux kernel developers will take interest in, and use in their own projects.

Downloading `nfsim`

`nfsim` is available from:

`http://ozlabs.org/~jk/projects/nfsim/`

Legal Statement

References

[1] Valgrind Developers. Valgrind website. `http://valgrind.org/`.

[2] Free Software Foundation. GCOV — a Test Coverage Program. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[3] Netfilter Core Team. Netfilter/iptables website. `http://netfilter.org`.

[4] Andrew Tridgell. talloc website. `http://talloc.samba.org/`.

Hotplug Memory Redux

Joel Schopp, Dave Hansen, & Mike Kravetz
IBM Linux Technology Center
jschopp@austin.ibm.com, haveblue@us.ibm.com, kravetz@us.ibm.com

Hirokazu Takahashi & IWAMOTO Toshihiro
VA Linux Systems Japan
taka@valinux.co.jp, iwamoto@valinux.co.jp

Yasunori Goto & KAMEZAWA Hiroyuki
Fujitsu
y-goto@jp.fujitsu.com, kamezawa.hiroyu@jp.fujitsu.com

Matt Tolentino
Intel
matthew.e.tolentino@intel.com

Bob Picco
HP
bob.picco@hp.com

Abstract

Memory Hotplug is one of the most anticipated features in the Linux Kernel. The purposes of memory hotplug are memory replacement, dynamic workload management, or Capacity on Demand of Partitioned/Virtual machines. In this paper we discuss the history of Memory Hotplug and the LinuxVM including mistakes made along the way and technologies which have already been replaced. We also discuss the current state of the art in Memory Hotplug including user interfaces, CONFIG_SPARSEMEM, the no bitmap buddy allocator, free area splitting within zones, and memory migration on PPC64, x86-64, and IA64. Additionally, we give a brief discussion on the overlap between Memory Hotplug and other areas including memory defragmentation and NUMA memory management. Finally, we gaze into the crystal ball to the future of Memory Hotplug.

1 Introduction

At the 2004 Ottawa Linux Symposium Andrew Morton had this to say in the keynote:

"Some features do tend to encapsulate poorly and they have their little sticky fingers into lots of different places in the code base. An example which comes to mind is CPU hot plug, and memory hot unplug. We may not, we may end up not being able to accept such features at all, even if they're perfectly well written and perfectly well tested due to their long-term impact on the maintainability of those parts of the software which they touch, and also to the fact that very few developers are likely to even be able to regression test them." [1]

It has been one year since that statement. Andrew Morton is a clever man who knows that the way to get developers to do something is to tell them it can't be done. CPU hot plug has been accepted[15]. The goal of this paper is to lay out how developers have been planning and coding to prove the memory half of that statement wrong.

2 Motivation

Memory hotplug was named for the ability to literally plug and unplug physical memory from a machine and have the Operating System keep running.

In the case of plugging in new physical memory the motivation is being able to expand system resources while avoiding downtime. The proverbial example of the usefulness is the slashdot effect. In this example a sysadmin runs a machine which just got slashdotted. The sysadmin runs to the parts closet, grabs some RAM, opens the case, and puts the RAM into the computer. Linux then recognizes the RAM and starts using it. Suddenly, Apache runs much faster and keeps up with the increased traffic. No downtime is needed to shutdown, insert RAM, and reboot.

Conversely, unplugging physical memory is usually motivated by physical memory failing. Modern machines often have the ability to recover from certain physical memory errors and to use those errors to predict that the physical memory is likely to have an unrecoverable error in the future. With memory hotplug the memory can be automatically disabled. The disabled memory can then be removed and/or replaced at the system administrator's convenience without downtime to the machine [31].

However, the ability to plug and unplug physical memory has been around awhile and nobody has previously taken it upon themselves to write memory hotplug for the Linux kernel. Fast forward to today and we have most major hardware vendors paying developers to write memory hotplug. Some things have changed; capacity upgrade on demand, partitioning, and virtualization all have made the resources assigned to an operating system much more fluid.

Capacity Upgrade On Demand came on the leading edge of this new wave. Manufacturers of hardware thought of a very clever and useful way to sell more hardware. The manufacturer would give users more hardware than they paid for. This extra unpaid for hardware would be disabled, and could be enabled if the customer later decided to pay for it. If the customer never decided to pay for it then the hardware would sit unused. Users got an affordable seamless upgrade path for their machines. Hardware manufacturers sold enough of the extra hardware they had already shipped they still made a profit on it. In business terms it was a win-win.

Without hotplug, capacity upgrades still require a reboot. This is bad for users who have to delay upgrades for scheduled downtime. The delayed upgrades are bad for hardware manufacturers who don't get paid for unupgraded systems. With hotplug the upgrades can be done without delay or downtime. It is so convenient that the manufacturers can even entice users with free trials of the upgrades and the ability to upgrade temporarily for a fraction of the permanent upgrade price.

The idea of taking a large machine and dividing up its resources into smaller machines is known as partitioning. Linux looks at a partition like it is a dedicated machine. This brings us back to our slashdotting example from physical hotplug. The reason that example didn't drive users to want hotplug was that it was only useful if there was extra memory in a closet somewhere and the system administrator could

open the machine while it was running. With partitioning the physical memory is already in the machine, it's just probably being used by another partition. So now hotplug is needed twice. Once to remove the memory from a partition that isn't being slashdotted and again to add it to a partition that is. The system administrator could even do this "hotplug" remotely from a laptop in a coffee house. Better yet management software could automatically decide and move memory around where it was needed. Because memory would be allocated more efficiently users would need less of it, saving them some money. Hardware vendors might even encourage selling less hardware because they could sell the management software cheaper than they sold the extra hardware it replaces and still make more money.

Virtualization then takes partitioning to the next level by removing the strict dependency on physical resources [10][17]. At first glance it would seem that virtualization ends the need for hotplug because the resources aren't real anyway. This turns out not to be the case because of performance. For example, if a virtual partition is created with 4GB of virtual RAM the only way to increase that to 256GB and have Linux be able to use that RAM is to hotplug add 252GB of virtual RAM to Linux. On the other side of the coin, if a partition is using 256GB of virtual RAM and whatever is doing the virtualizing only has 4GB of real honest-to-goodness physical RAM to use, performance will make it unusable. In this case the virtualization engine would want to hotplug remove much of that virtual RAM.

So there are a variety of forces demanding memory hotplug from hardware vendors to software vendors. Some want it for reliability and uptime. Others want it for workload balancing and virtualization.

Thankfully for developers it is also an interesting problem technically. There are lots of difficult problems to be overcome to make memory hotplug a success, and if there is one thing a developer loves it is solving difficult problems.

3 CONFIG_SPARSEMEM

3.1 Nonlinear vs Sparsemem

Previous papers[6] have discussed the concept of nonlinear memory maps: handling systems which have non-trivial relationships between the kernel's virtual and physical address spaces.

In 2004, Dave McCracken from IBM created a quite complete implementation of nonlinear memory handling for the hotplug memory project. As presented in[6], this implementation solved two problems: separating the `mem_map[]` into smaller pieces, and the nonlinear layout.

The nonlinear layout component turned out to be quite an undertaking. Its implementation required changing the types of some core VM macros: `virt_to_page()` and `page_to_virt()`. It also required changing many core assumptions, especially in boot-time memory setup code, which impaired other development. However, the component that separated the `mem_map[]`s turned out to be relatively problem-free.

The decision was made to separate the two components. Nonlinear layouts are not required by simple memory addition. However, the split-out `mem_map[]`s are. The memory hotplug plan has always been to merge hot-add alone, before hot-remove, to minimize code impact at one time. The `mem_map[]` splitting feature was named sparsemem, short for sparse memory handling, and the nonlinear portion will not be implemented until hot-remove is needed.

3.2 What Does Sparsemem Do?

Sparsemem has several of the same design goals as DISCONTIGMEM, which is currently in use in the kernel for similar purposes. Both of them allow the kernel to efficiently handle gaps in its address space. The normal method for machines without memory gaps is to have a `struct page` for each physical page of RAM in memory. If there are gaps from things like PCI config space, there are `struct page`'s, but they are effectively unused.

Although a simple solution, simply not using structures like this can be an extreme waste of memory. Consider a system with 100 1GB DIMM slots that support hotplug. When the system is first booted, only 1 of these DIMM slots is populated. Later on, the owner decides to hotplug another DIMM, but puts it in slot 100 instead of slot 2. This creates a 98GB gap. On a 64-bit system, each `struct page` is 64 bytes.

$$(\frac{98GB}{4096\frac{bytes}{page}}) * (64\frac{bytes}{struct\,page}) \approx 1.5GB$$

The owner of the system might be slightly displeased at having a **net loss** of 500MB of memory once they plug in a new 1GB DIMM. Both sparsemem and discontigmem offer an alternative.

3.3 How Does Sparsemem Work?

Sparsemem uses an array to provide different `pfn_to_page()` translations for each "section" of physical memory. The sections are arbitrarily sized and determined at compile-time by each specific architecture. Each one of these sections effectively gets its own, tiny version of the `mem_map[]`.

However, one must also consider the storage cost of such an array which must represent every possible physical address. Let's take PPC64 as an example. Its sections are 16MB in size and there are, today, systems with 1TB of memory in a single system. To keep future expansion in mind (and for easy math), assume that the limit is 16TB. This means 2^{20} possible sections and, with 1 64-bit `mem_map[]` pointer per section, that's 8MB of memory used. Even on the smallest (256MB) configurations, this amount is a manageable price to pay for expandability all the way to 16TB.

In order to do quick `pfn_to_page()` operations, the index into the large array of the page's parent section is encoded in `page->flags`. Part of the sparsemem infrastructure enables sharing of these bits more dynamically (at compile-time) between the `page_zone()` and sparsemem operations.

However, on 32-bit architectures, the number of bits is quite limited, and may require growing the size of the `page->flags` type in certain conditions. Several things might force this to occur: a decrease in the size of each section (if you want to hotplug smaller, more granular, areas of memory), an increase in the physical address space (very unlikely on 32-bit platforms), or an increase in the number of consumed `page->flags`.

One thing to note is that, once sparsemem is present, the NUMA node information no longer needs to be stored in the `page->flags`. It might provide speed increases on certain platforms and will be stored there if there are unused bits. But, if there are inadaquate unused bits, an alternate (theoretically slower) mechanism is used; `page_zone(page)->zone_pgdat->node_id`.

3.4 What happens to Discontig?

As was noted earlier sparsemem and discontigmem have quite similar goals, although quite

different implementations. As implemented today, sparsemem replaces DISCONTIGMEM when enabled. It is hoped that SPARSEMEM can eventually become a complete replacement as it becomes more widely tested and graduates from experimental status.

A significant advantage sparsemem has over DISCONTIGMEM is that it's completely separated from CONFIG_NUMA. When producing this implementation, it became apparent that NUMA and DISCONTIG are often confused.

Another advantage is that sparse doesn't require each NUMA node's ranges to be contiguous. It can handle overlapping ranges between nodes with no problems, where DISCONTIGMEM currently throws away that memory.

Surprisingly, sparsemem also shows some marginal performance benefits over DISCONTIGMEM. The base causes need to be investigated more, but there is certainly potential here.

As of this writing there are ports for sparsemem on i386, PPC64, IA64, and x86_64.

4 No Bitmap Buddy Allocator

4.1 Why Remove the Bitmap?

When memory is hotplug added or removed, memory management structures have to be reallocated. The buddy allocator bitmap was one of these structures.

Reallocation of bitmaps for Memory Hotplug has the following problems:

- Bitmaps were one of the things which assumed that memory is linear. This assumption didn't fit SPARSEMEM and Memory Hotplug.
- For resizing, physically contiguous pages for new bitmaps were needed. This increased possibility of failure of Memory Hotplug because of difficulty of large size page allocation.
- Reallocation of bitmaps is complicated and computationally expensive

For Memory Hotplug, bitmaps presented a large obstacle to overcome. One proposed solution was dividing and moving bitmaps from zones to sections as was done with memmaps. The other proposed solution, eliminating bitmaps altogether, proved simpler than moving them.

4.2 Description of the Buddy Allocator

The buddy allocator is an memory allocator which coalesces pages into groups of 2^X length. X is usually 0-10 in Linux. Pages are coalesced into a group of length of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024. X is called an "order". Only a head page of a buddy is linked to `free_area[order]`.

This grouping is called a buddy. A pair of buddies in order X, which are length of 2^X, can be coalesced into a buddy of $2^{(X+1)}$ length. When a pair of buddies can be coalesced in order X, offset of 2 buddies are $2^{(X+1)} * Y$ and $2^{(X+1)} * Y + 2^X$. Hence, another buddy of a buddy in order X can be calculated as (Offset of a buddy) XOR $(1 << (X))$.

For example, page 4 (0x0100) can be coalesced with page 5 (0x0101) in order 0, page 6 (0x0110) in order 1, page 0 (0x0000) in order 2.

4.3 Bitmap Buddy Allocator

The role of bitmaps was to record whether a page's buddy in a particular order was free or

not. Consider a pair of buddies at $2^{(X+1)} * Y$ and $2^{(X+1)} * Y + 2^X$.

When `free_area[X].bitmap[Y]` was 1, one of the buddies was free. So `free_pages()` can determine whether a buddy can be coalesced or not from bitmaps. When both buddies were freed, they were coalesced and `free_area[X].bitmap[Y]` set to 0.

4.4 No Bitmap Buddy Allocator

When it comes to the no bitmap buddy allocator, instead of recording whether a page has its buddy or not in a bitmap, the free buddy's order is recorded in `memmap`. The following expression is used to check a buddy page's status:

```
page_count(page) == 0 &&
PG_private is set &&
page->private == X
```

The three elements that make up this expression are:

- When `page_count(page) == 0`, page is not used.
- Even if `page_count(page) == 0`, it's not sure that the page is linked to the free area. When a page is linked to the free area, `PG_private` is set.
- When `page_count(page)==0 && PG_private` is set, `page->private` indicates its order.

Here, offset of an another buddy of a buddy in order X can be calculated as (Offset of page) XOR 2^X. The following code is the core of no bitmap buddy allocator's coalescing routine:

```
struct page *base = zone->zone_mem_map;
int page_idx = page - base;
while ( order < MAX_ORDER ) {
  int buddy_idx = page_idx ^ (1 << order);
  struct page *buddy = base + buddy_idx;
  if (!(page_count(buddy) == 0 &&
        PagePrivate(buddy) &&
        buddy->private == order))
              break;
  remove buddy from zone->free_area[order]
  ClearPagePrivate(buddy);
  if (buddy_idx < page_idx)
        page_idx = buddy_idx;
  order++;
}
page = page_idx + base;
SetPagePrivate(page);
page->private = order;
link page to zone->free_area[order]
```

There is no significant performance difference either way between bitmap and no bitmap coalescing.

With SPARSEMEM, `base` in the above code is removed and following the relative offset calculation is used. Thus, the buddy allocator can manage sparse memory very well.[1]

```
page_idx = pfn_to_page(page);
buddy_idx = page_idx ^ (1 << order);
buddy = page + (buddy_idx - page_idx);
```

5 Free Area Splitting Within Zones

The buddy system provides an efficient algorithm for managing a set of pages within each zone [7][16][18][11]. Despite the proven effectiveness of the algorithm in its current form as used in the kernel, it is not possible to aggregate a subset of pages within a zone according to specific allocation types. As a result, two physically contiguous page frames (or sets of page frames) may satisfy allocation requests that are drastically different. For example, one page frame may contain data that is only temporarily used by an application while the other

[1] SPARSEMEM guarantees that memmap is contiguous at least up to MAX_ORDER.

is in use for a kernel device driver. While this is perfectly acceptable on most systems, this scenario presents a unique challenge on memory hotplug systems due to the variances in reclaiming pages that satisfy each allocation type.

One solution to this problem is to explicitly manage pages according to allocation request type. This approach avoids the need to radically alter existing page allocation and reclamation algorithms, but does require additional structure within each zone as well as modification of the existing algorithms.

5.1 Origin of Page Allocation Requests

Memory allocations originate from two distinct sources—user and kernel requests. User page allocations typically result from a write into a virtual page in the address space of a process that is not currently backed by physical memory. The kernel responds to the fault by allocating a physical page and mapping the virtual page to the physical page frame via page tables. However, when the system is under memory pressure, user pages may be paged out to disk in order to reclaim physical page frames for other higher priority requests or tasks. The algorithms and techniques used to accomplish this function constitute much of the virtual memory research conducted to date[22][23][24][25][26][27].

In Linux, user level allocations may be satisfied from pages contained in any zone, although they are preferably allocated from the HIGHMEM zone if that zone is employed by the architecture. This is reasonable considering these pages are not permanently mapped by the kernel. Architectures that do not employ the HIGHMEM zone direct user level allocations to one of the other two zone types, NORMAL or DMA. Unlike user allocations, kernel allocations must be satisfied from memory that is permanently mapped in the virtual address space. Once a suitable zone is chosen, an appropriately sized region is plucked from the respective free area list via the buddy algorithm without regard for whether it satisfies a kernel allocation or a user allocation.

5.2 Distinguishing Page Usage

During a page allocation, attributes are provided to the memory allocation interface functions. Each attribute provides a hint to the allocation algorithm in order to determine a suitable zone from which to extract pages; however, these hints are not necessarily provided to the buddy system. In other words, the region from which the allocation is satisified is only determined at a zone granularity. On systems such as PPC64 this may include the entirety of system memory! In order to enable the distinction of user allocation from kernel allocations within a zone, additional flags that specify whether the region must be provided to the buddy algorithm. These flags include:

- User Reclaimable
- Kernel Reclaimable
- Kernel Non-Reclaimable

Using these flags, the buddy allocation algorithm may further differentiate between page allocations and attempt to maintain regions that satisfy similar allocations and more significantly, have similar presence requirements.

5.3 Multiple Free Area Lists

Existing kernels employ one set of free area lists per zone as shown in figure1. In order

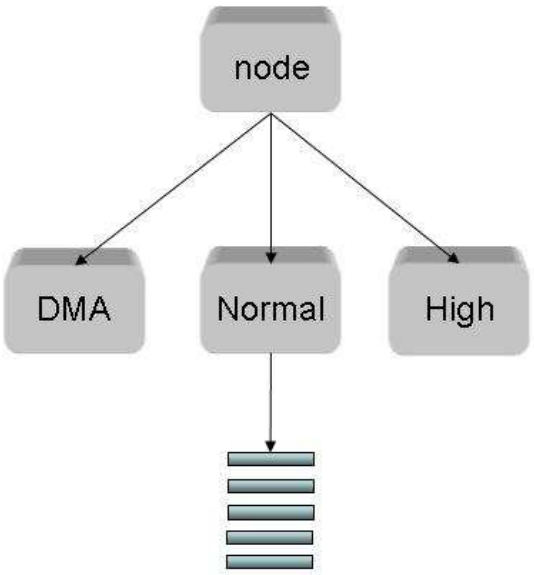

Figure 1: Existing free area list structure

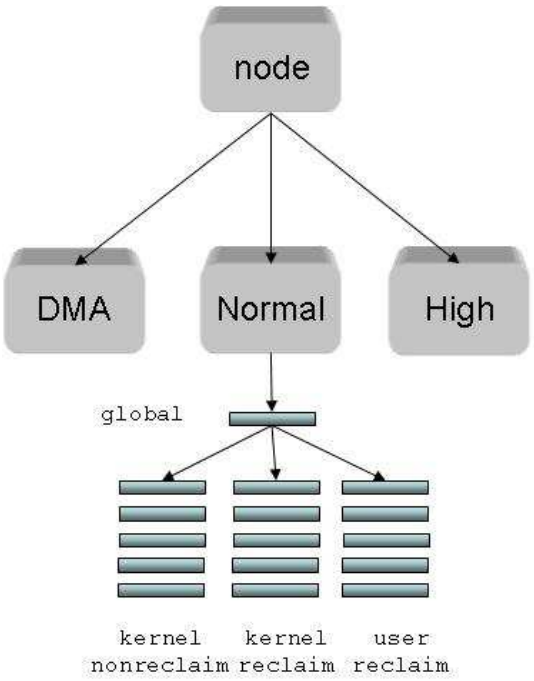

Figure 2: New free area list structure for fragmentation

to explicitly manage the user versus kernel distinction of memory with zones, multiple sets of free area lists are used within each zone, specifically one set of free area lists per allocation type. The basic strategy of the buddy algorithm remains unchanged despite this modification. Each set of free area lists employs the exact same splitting and coalescing steps during page allocation and reclamation operations. Therefore the functional integrity of the overall algorithm is maintained. The novelty of the approach involves the decision logic and accounting involved in directing allocations and free operations to the appropriate set of free area lists.

Mel Gorman posted a patch that implements exactly this approach in an attempt to minimize external memory fragmentation, a consistent issue with the buddy algorithm [19][8][7][11][16]. This approach introduces a new global free area list with MAX_ORDER sized memory regions and three new free area lists of size MAX_ORDER-1 as depicted in figure 2 below.

Although Mel's approach provides the basic infrastructure needed by memory hotplug, additional structure is required. In addition to the set of free area lists for each allocation type, an additional global free area list for contiguous regions of MAX_ORDER size is also maintained as depicted in figure three. The addition of this global list enables accounting for MAX_ORDER sized memory regions according to the capability to hotplug the region. Thus, during initialization, memory regions within each zone are directed to the appropriate global free area list based on the potential to hotplug the region at a later time. This translates directly to the type of allocation a page satisfies. For example, many kernel pages are pinned in memory and will never be freed. Hence, these pages will be obtained from the global *pinned* list. On the other hand nearly every user page may be reclaimed, so these pages will be obtained from the global *hotplug* list.

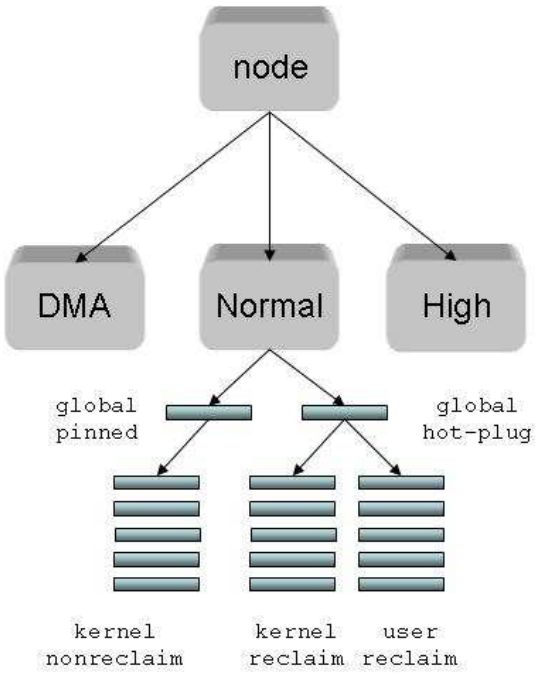

Figure 3: New free area list structure for memory hotplug

5.4 Memory Removal

The changes detailed in this section enable the isolation of sets of pages according to the type of allocation they may satisfy. Because user pages are relatively easy to reclaim, those page allocations will be directed to the regions that are maintained in the global *hotplug* free area list. During boot time or during a memory hot-add operation, the system firmware details which regions may be removed at runtime. This information provides the context for initializing the global *hotplug* list. As pages are allocated thus depleting the user and kernel reclaimable free area lists, additional MAX_ORDER regions may be derived from the global *hotplug* list. Similarly, the global *pinned* list provides pages to the kernel non-reclaimable lists upon depletion of available pages of sufficient size.

Because pages that may be hot-removed at runtime are isolated such that they satisfy user or kernel reclaimable allocations, memory removal is possible. This was not previously possible with the existing buddy algorithm due to the likelihood that a pinned kernel non-reclaimable page could be located within the range to be removed. Thus, kernels that only employ a subset of the potential zones may support hot-remove transparently.

5.5 Configurability

While satisfying allocation requests from discrete memory regions according to allocation type does enable removal of memory within zones, there is still the potential for one type of allocation to run out of pages due to the assignment of pages to each global free list. Mel dealt with this issue for the fragmentation problem by allowing allocations to *fallback* to another free area list should one become depleted[19].

While this is reasonable for most systems, it compromises the capability to remove memory should a non-reclaimable kernel allocation be satisfied by some set of pages in a hotplug region. As this type of fallback policy decision largely depends on the intended use of the system, one approach is to allow for the fallback decision logic to be configured by the system administrator. Therefore, systems that aren't likely to need to remove memory, even though the functionality is available, may allow the fallback to occur as the workload demands. Other systems in which memory removal is more critical may disable the fallback mechanism, thus preserving the integrity of hotplug memory regions.

6 Memory migration

6.1 Overview

In memory hotplug removal events, all the pages in some memory region must be freed in a timely fashion, while processes are running as usual. Memory migration achieves this by blocking page accesses and moves page contents to new locations.

Although using `shrink_list()` function—which is the core of `kswapd`—sounds simpler, it cannot reliably free pages and causes many disk I/Os. Additionally, the function cannot handle pages which aren't associated with any backing stores. Pages on a ramdisk are an example of this. The memory migration is designed to solve these issues. Page accesses aren't a problem because they are blocked, while `shrink_list()` cannot process pages that are being accessed. Unlike `shrink_list()`, most dirty pages can be processed without writing them back to disk.

6.2 Interface to Migrate Page

To migrate pages, create a list of pages to migrate and call the following function: `int try_to_migrate_pages(struct list_head *page_list)`

It returns zero on success, otherwise sets page_list to a list of pages that cannot migrate and returns a non-zero value. Callers must check return values and retry failed pages if necessary.

This function is primarily for memory hotplug remove, but also can be used for memory defragmentation (see Section 8.1) or process migration (see Section 8.2.3).

6.3 How does the memory migration work?

A memory migration operation consists of the following steps. The operation of anonymous pages is slightly different.

1. lock `oldpage`, which is the target page
2. allocate and lock `newpage`
3. modify `oldpage` entry in `page_mapping(oldpage)->page_tree` with `newpage`
4. invoke `try_to_unmap(oldpage, virtual_address_list)` to unmap `oldpage` from the process address spaces.
5. wait until `!PageWriteback(oldpage)`
6. write back `oldpage` if `oldpage` is dirty and `PagePrivate(oldpage)` and no file system specific method is available
7. wait until `page_count(oldpage)` drops to 2
8. `memcpy(newpage, oldpage, PAGE_SIZE)`
9. make `newpage` up to date
10. unlock `newpage` to wakeup the waiters
11. free `oldpage`

The key is to block accesses to the page under operation by modifying the `page_tree`. After the `page_tree` has been modified, no new access goes to `oldpage`. The accesses are redirected to `newpage` and blocked until the data is ready because it is locked and isn't up to date (Figure 4).

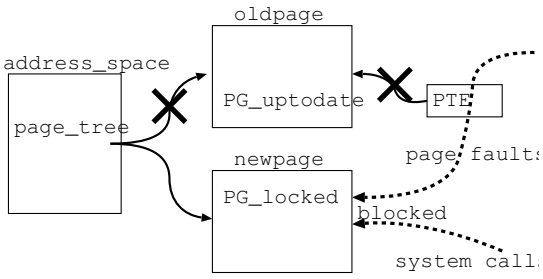

Figure 4: `page_tree` rewrite and page accesses

To handle `mlock()`ed pages, `try_to_unmap()` now takes two arguments. If the second argument is non-NULL, the function unmaps `mlock()ed` pages also and records unmapped virtual addresses, which are used to reestablish the PTEs when the migration completes.

Because the direct I/O code protects target pages with incremented `page_count`, memory migration doesn't interfere with the I/O.

In some cases, a memory migration operation needs to be rolled back and retried later. This is a bit tricky because it is likely that some processes have already looked up the `page_tree` and are waiting for its lock. Such processes need to discard `newpage` and look up the `page_tree` again, as `newpage` is now invalid.

6.3.1 Anonymous Memory Migration

The memory migration depends on `page_tree` lists of inodes, while anonymous pages may not correspond to any of them. This structure is strictly required to block all accesses to pages on it during migration.

Therefore, anonymous pages should be moved into the swap-cache prior to migrating them. After that these pages are placed in the `page_tree` of `swapper_space`, which manages all pages in the swap-cache. These pages can be migrated just like pages in the page-cache without any disk I/Os.

The important issue of systems without swap devices remains. To solve it, Marcelo Tosatti has proposed the idea of "migration cache"[2] and he's working on its implementation. The migration cache is very similar to the swap-cache except it doesn't require any swap devices.

6.4 Keeping Memory Management Aware of Memory Migration

The memory migration functionality is designed to fit the existing memory management semantics and most of the code works without a modification. However, the memory management code should satisfy the following rules:

- Multiple lookups of a page from its `page_tree` should be avoided. If a kernel function looks up a `page_tree` location multiple times, a memory migration operation can rewrite the `page_tree` in the meanwhile. When such a `page_tree` rewrite happens, it usually results in a deadlock between the kernel function and the memory migration operation. The memory migration implements a timeout mechanism to resolve such deadlocks, but it is preferable to remove the possibility of deadlocks by avoiding multiple `page_tree` lookups of the same page. Another option is to use non-hotremovable memory for such pages.
- The pages which may be grabbed for an unpredictably long time must be allocated from non-hotremovable memory,

even though it may be in the page-cache or anonymous memory. For instance, pages used as ring buffers for asynchronous input/output (AIO) events are pinned not to be freed.

- For a swap-cache page, its PG_swapcache flag bit needs checking after obtaining the page lock. This is due to how the memory migration is implemented for swap-cache pages and not directly related to unwinding. Such a check code is added in `do_swap_page()`.

- Functions that call `lock_page()` must be aware of the unwinding of memory migration. Basically, a page must be checked if it is still valid after every `lock_page()` call. If it isn't, one has to restart the operation from looking up the `page_tree` again. A good example of such restart is in `find_lock_page()`.

6.5 Tuning

6.5.1 Implementing File System Specific Methods for Memory Migration

Memory migration works regardless of file systems in use. However, it is desirable that file systems which are intensively used implement the helper functions which are described in this subsection.

There are various things that refer to pages, and some of these references need time consuming operations such as disk I/Os to complete in order for the reference to be dropped. This subsection focuses on one of these—the handling of dirty buffer structures pointed by `page->private`.

When a dirty page is associated with a buffer, the page must be made clean by issuing a write-back operation and the buffer must be freed unless there is an available file system specific operation defined.

The above operation can be too slow to be practical because it has to wait a writeback I/O completion. A file system specific operation can be defined to avoid this problem by implementing the `migrate_page()` method in the `address_space_operations` structure.

For example, the `buffer_head` structures that belong to ext2 file systems are handled by the `migrate_page_buffer` function. This function enables page migration without write-back operations by having `newpage` to take over the `buffer_head` structure pointed by `page->private`. It is implemented as follows:

- Wait until the `page_count` drops to the prescribed value (3 when the `PG_private` page flag is set, 2 otherwise). While waiting, issue the `try_to_unmap()` function calls.

- If the `PG_private` flag is set, process the `buffer_head` structure by calling the `generic_move_buffer()` function. The function waits until the `buffer_count` drops and the buffer lock is released. Then, it has `newpage` to take over the buffer_head structure by modifying `page->private`, `newpage->private` and the `b_page` member in the `buffer_head` structure. To adjust `page_count` due to the `buffer_head` structure, increment the `page_count` of `newpage` by one and decrement the one of `page` by one.

- At this point, the `page_count` of page is 2 regardless of the original state of the `PG_private` flag.

6.5.2 Combination With `shrink_list()`

Memory pressure caused by memory migration can be reduced. This memory pressure can cause reclaim of pages as replacements. Inactive pages are not worth migrating when the resultant migration causes other valid pages be reclaimed. This undesirable effect perturbs the LRUness of pages reclaimed. It would be preferable to just release these pages without migrating them.

The current implementation[3] invokes `shrink_list()` to release inactive pages and moves only active pages to new locations in case of memory hotplug removal.

6.6 Hugetlb Page Migration.

Due to certain workloads like databases and high performance computing (HPC) large page capability is critical for good performance. Because these pages are so critical to these workloads it follows that page migration must support migration of large pages to be widely used.

6.6.1 Interface to Migrate Hugetlb Pages

The prototype[5] interface for hugetlb migration seperates normal page migration from huge page migration.

When a caller notices the page needing to be migrated is a hugetlb page, it has to pass the page to `try_to_migrate_hugepage()`, migrating it without any system freeze or any process suspension.

6.6.2 Design of hugetlb page migration

The migration can be done in the same way for normal pages, using the same memory migration infrastructure. Luckily, it's not so hard to implement because Linux kernel manages large pages—often called hugetlb pages—via the pseudo file system known as hugetlbfs.

Linux kernel handles them in a similar manner as it handles normal pages in the page-cache. It inserts each of them into the `page_tree` of the associated inode in hugetlbfs and maps them into process address spaces using `mmap()` system call.

There is one additional requirement for migration of large pages. Demand paging against hugetlb pages must be blocked, with all accesses via process address spaces to pages under migration blocked in a pagefault handler until the migration is completed.

Therefore, the hugetlb page management related to demand paging feature has to be enhanced as follows:

- A pagefault handler for hugetlb pages must be implemented. The implementation[4] Chen, Kenneth W and Christoph Lameter are working on can be used with some modification, making the processes block in the pagefault handler if the page is locked. This is similar to what the pagefault handler for normal pages does.
- The function `try_to_unmap()` must be able to handle hugetlb pages to unmap them from process address spaces. This means `objrmap`—the object-based reverse mapping VM—also has to be introduced so that page table entries associated with any pages can be found easily.

Another interesting topic is hugetlb page allocation, which is almost impossible to do dynamically. Physically contiguous memory allo-

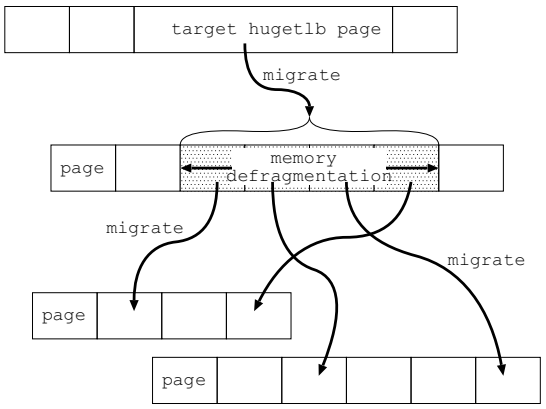

Figure 5: `hugetlb` page migration

cation is one of the well known issues remaining to be solved. The current hugetlb page management chooses the approach that all pages should be reserved at system start-up time.

Despite its current state, hugetlb page migration can not continue to use this approach. On demand allocation is strictly required. Fortunately, this is going to be solved with "Memory defragmentation" (see Section 8.1). Marcelo Tosatti is working on "Free area splitting within zones" effort (see section 5).

6.6.3 How `hugetlb` Page Migration Works

There really isn't much difference between hugetlb page migration and normal page migration. The following is the algorithm flow for this migration.

1. Allocate a new hugetlb page from the page allocator also known as the buddy allocator. This may require memory defragmentation to make a sufficient contiguous range (figure 5).
2. Lock the newly allocated page and keep it non-uptodate, without the `PG_uptodate` flag on it.
3. Replace a target hugetlb page with the new page on `page_tree` of the corresponding inode in hugetlbfs.
4. Unmap the target page from the process address spaces, clearing all page table entries mapping it.
5. Wait until all references on the target page are gone.
6. Copy from the target page to the new page.
7. Make the new page uptodate, setting the `PG_uptodate` flag on it.
8. Release the target page into the page allocator directly.
9. Unlock the new page to wake up all waiters.

6.7 Restriction

Under some rare situations, pages cannot migrate, and making those migrations functional would require too much code to be practical.

- NFS page-cache may have a non-responding NFS server. NFS I/O requests cannot complete if the server isn't responding. The pages with such outstanding NFS I/O requests cannot migrate. It is technically possible to handle this situation by updating all the references to an oldpage with ones to a newpage, but the code modification would be very large and probably not maintainable.
- Page-cache of which the file is used by `sendfile()` are also problematic. When a page-cache page is used by `sendfile()`, its page_count is kept raised until corresponding TCP packets are ACKed. This becomes a problem when a connection peer doesn't read data from the TCP connection.

- RDMA client/server memory use may also be an issue but further investigation is required.

6.8 Future Work

Currently, nonlinear mmaped pages[2] cannot migrate as `try_to_unmap()` doesn't unmap such pages. This must be addressed.

All file systems should have their own `migrate_page()` method. This will help performance considerably as the filesystems can make more intelligent decisions about their own data.

Kernel memory should be migratable too. A first approach would be migrating page table pages which consume significant memory. This migration should be reasonably straightforward.

7 Architecture Implementation Specifics

Memory hotplug has been implemented on many different architectures. Each of these architectures have unique hardware and consequently do memory management in different ways. They each present unique challenges and solutions that should be of interest to future implementators on the other architectures that currently don't support memory hotplug. Additionally, those whose architectures are already covered can better understand their own architectures by comparing them side by side with others.

[2]With `remap_file_pages()` system call, several pieces of a file can be mapped into one contiguous virtual memory.

7.1 PPC64 Implementation

The PPC64 architecture is perhaps the most mature with respect to the support of memory hotplug. This is because there are other operating systems that currently support memory hotplug on this architecture.

7.1.1 Logical Partition Environment

Operating Systems running on PPC64 operate in a Logical Partition (LPAR) of the machine. These LPARs are managed by a underlying level of firmware known as the hypervisor. The hypervisor manages access to the actual underlying hardware resources. It is possible to dynamically modify the resources associated with an LPAR. Such dynamically modifiable LPARS are known as Dynamic LPARS (DLPARs)[29].

Memory is one of the resources that can be dynamically added to or removed from a DLPAR on PPC64. In a PPC64 system, physical memory is divided into memory blocks that are then assigned to LPARs. The hypervisor performs remapping of real physical addresses to addresses that are given to the LPAR[3] These memory blocks with remapped addresses appear as physical memory to the Operating Systems in the LPAR. When an OS is started on an LPAR, the LPAR will have a set of memory blocks assigned to it. In addition, memory blocks can be added or removed to the LPAR while the OS is active. The size of memory blocks managed by the hypervisor is scaled based on the total amount of physical memory in the machine. The minimum size block

[3]To Linux the addresses given to it are considered physical addresses, but they are not in actuality physical addresses. This causes no end of confusion in developers conversations because developers get confused over what is a virtual address, physical address, remapped address, etc.

is 16MB[4]. As a result, the default SPARSEMEM section size for PPC64 is a relatively small 16MB.

7.1.2 Add/Remove Operations

On PPC64, the most common case of memory hotplug is not expected to be the actual addition or removal of DIMMs. Rather, memory blocks will be added to or removed from a DLPAR by the hypervisor. These add or remove operations are initiated on the Hardware Management Console (HMC). When memory is added to an LPAR, the HMC will notify a daemon running in the OS of the desire to add memory blocks. The daemon in turn makes a special system call that results in calls being made to the hypervisor. The hypervisor then makes additional memory blocks available to the OS. As part of the special system call processing, the physical address[5] of these new blocks is obtained. With the physical address known, scripts called via the daemon use the sysfs memory hotplug interface to create new memory sections associated with the memory blocks.

For memory remove operations, the HMC once again contacts the daemon running in the OS. The OS then executes a script that uses the sysfs interfaces to offline a memory section. Once a section is offlined, a special system call is made that results in calls to the hypervisor to isolate the memory from the DLPAR.

[4] 256MB is a more typical minimum block size. On some machines the user can actually change the minimum block size the machine will use

[5] This is not the real physical address, but the remapped address that Linux thinks is a real physical address

7.1.3 Single Zone and Defragmentation

PPC64 makes minimal use of memory zones. This is because DMA operations can be performed to any memory address. As a result, only a single DMA zone is created on PPC64 and no HIGHMEM or NORMAL zones. Of course, there may be multiple DMA zones (one per node) on a NUMA architecture. Having a single zone makes things simpler but it does nothing to segregate memory allocations of different types. For example, on architectures that support HIGHMEM, allocations for user pages mostly come from this zone. Having multiple zones provides a very basic level of segregation of different allocation types. Since we have no such luxury on PPC64, we must employ other methods to segregate allocation types. The memory defragmentation work done by Mel Gorman is a good starting point for this effort[19]. Mel's work segregates memory allocations on the natural MAX_ORDER PAGE size blocks managed by the page allocator. Because PPC64 has a relatively small default section size of 16 MB, it should be possible to extend this concept in an effort to segregate allocations to segment size blocks.

7.1.4 PPC64 Hypervisor Functionality

The PPC64 hypervisor provides functionality to aid in the removal of memory sections. The H_MIGRATE_DMA call aids in the remapping of DMA mapped pages. This call will selectively suspend bus traffic while migrating the contents of DMA mapped pages. It also modifies the Translation Control Entries (TCEs) used for DMA accesses. Such functionality will allow for the removal of *difficult* memory sections on PPC64.

7.2 x86-64 Implementation

Although much of the memory hot-plug infrastructure discussed in this paper, such as the *sparsemem* implementation, is generic across all platforms, architecture specific support is still required due to the variance in memory management requirements for specific processor architectures. Fortunately, the changes to the x86-64 Linux kernel beyond *sparsemem* to support memory hotplug have been minimized to the following:

- Kernel Page Table Initialization (capacity addition)
- ZONE_NORMAL selection
- Kernel Page Table Tear Down (capacity reduction)

The x86-64 kernel doesn't require the HIGHMEM zone due to the large virtual address space provided by the architecture [28][12]. Thus, new memory regions discovered during memory hot-add operations result in expansion of the NORMAL zone. Conversely, because the x86-64 kernel only uses the DMA and NORMAL zones, removal of memory within each zone as discussed in 5 is required.

Much of the development of the kernel support for memory hotplug has relied on *logical* memory add and remove operations, which has enabled the use of existing platforms for prototyping. However, the x86-64 kernel has been tested and used on real hardware that supports memory hotplug. Specifically, the x86-64 memory hotplug kernels have been tested on a recently released Intel Xeon®[6] platform that supports physical memory hotplug operations.

[6]Xeon is a registered trademark of the Intel Corporation

One of the key pieces of supporting physical memory hotplug is notification of memory capacity changes from the hardware/firmware. The ACPI specification outlines basic information on memory devices that is used to convey these changes to the kernel. Accordingly, in order to fully support physical memory hotplug in the kernel the x86-64 kernel uses the ACPI memory hotplug driver to field notifications from firmware and notify the VM of the addition or reduction at runtime using the same interface employed by the logical operations. Further information on the ACPI memory hotplug driver support in the kernel may be found in [21].

7.3 IA64 Implementation

IA64 is one of architectures where Memory Hotplug is eagerly desired. From the view of Memory Hotplug, IA64 linux has following characteristics:

- The memory layout of IA64 is very sparse with lots of holes.
- For managing holes, `VIRTUAL_MEM_MAP` is used in some configurations.
- MAX_ORDER is not 11 but 18.
- IA64 supports a physical address bits of 50

Early lmbench2 data has shown that SPARSEMEM performs equivalently to DISCONTIGMEM+VIRTUAL_MEM_MAP. The data was taken on a non-NUMA machine. Further work should be done with other benchmarks and NUMA hardware.

7.3.1 SPARSEMEM and VIRTUAL MEM MAP

The VM uses a `memmap[]`, a linear array of page structures. With DISCONTIGMEM, `memmap[]` is divided into several `node_mem_maps`. In general, `memmap[]` is allocated in physically contiguous pages at boot time.

The memory layout of IA64 is very sparse with lots of holes. Sometimes there are GBs of memory holes, even for a non-NUMA machine. In IA64 DISCONTIGMEM, a `vmemmap` is used to avoid wasting memory. A `vmemmap` is a `memmap` which uses contiguous region of virtual address instead of contiguous physical memory. [7]

It is useful to hide holes and to create sparse `memmap[]`s. It resides in region 5 of the virtual address space, which uses virtual page table [8] like vmalloc.

Unfortunately, VIRTUAL_MEM_MAP is quite complicated. Because of the complications VIRTUAL_MEM_MAP presents, early designs for MEMORY_HOTPLUG were too complicated to be successfully implemented. SPARSEMEM cleanly removes VIRTUAL_MEM_MAP and thus avoids the associated complexity altogether. Because SPARSEMEM is simpler than VIRTUAL_MEM_MAP it is a logical replacement for VIRTUAL_MEM_MAP for situations other than just hotplug. SPARSEMEM divides the whole memmap into the section's `section_memmap`s. All `section_memmap`s reside in region 7 of the virtual address space. Region 7 is an identity mapped segment and handled by the fast TLB miss handler with big page size. If a hole covers the whole section, section_memmap is not allocated. Holes in a section are treated as reserved pages. For example, an HP rx2600 with 4GB of memory has the available physical memory at two locations with sizes of 1Gb and 3Gb. For VIRTUAL_MEM_MAP the holes would be represented by empty virtual space with vmemmap. SPARSEMEM handles a hole which covers an entire section with an invalid section.

[7] VIRTUAL_MEM_MAP is configurable independent of DISCONTIGMEM

[8] VHPT, Virtual Hash Page Table, is a hardware supported function to fill TLB

7.3.2 SPARSEMEM NUMA

The `mem_section[]` array is on the BP's node. Because `pfn_to_page()` accesses it, a non BP node `pfn_to_page()` is slightly more expensive. Besides boot time the section array is modified only during a hotplug event. These events should happen infrequently. This frequently accessed but rarely changing data suggests replicating the array into all nodes in order to eliminate the non BP node penalty. Hotplug memory updates would have to notify each node of modifications to the array.

7.3.3 Size of Section and MAX_ORDER

One feature which is very aggresive on IA64 is the configuration parameter FORCE_MAX_ZONEORDER. This overwrites MAX_ORDER to 18. For a 16kb page size the resultant MAX_ORDER region is 4Gb(18+14). This is done for supporting 4Gb HugetlbFS. SPARSEMEM constrains $PAGE\_SIZE * 2^{(MAX\_ORDER-1)}$ to be less than or equal to section size. For HugetlbFS we have: (1)the smallest size of section is 4GB and (2)holes smaller than 4GB consume reserved page structures. 18 of MAX_ORDER seems to be rather optimistic

value for Memory Hotplug. Currently, configuation of FORCE_MAX_ZONEORDER is modified at compile time. At configuration time, if HUGETLB isn't selected, FORCE_MAX_ZONEORDER can be configured to $11-20$. If HUGETLB is selected, MAX_ORDER and SECTION_SIZE are adjusted to support 4Gb HUGETLB Page.

7.3.4 Vast 50 Bits Address Space of IA64

The IA64 architecture supports a physical address bit limit of 50, which can addresss up to 1 petabyte of memory. A section array with a 256Mb section size requires 32Mb of data to cover the whole address range. The Linux kernel by default is configured to only use 44 bits maximum, which can address 16 terabytes of memory. This only requires 512Kb of data to cover the whole address range. The number of bits used is configurable at compile time.

8 Overlap with other efforts

During the development of memory hotplug the developers discovered two surprising things.

- Parts of the memory hotplug code were very useful to those who don't care at all about memory hotplug.
- Code others were developing without so much as a thought of memory hotplug proved useful for memory hotplug.

This section attempts to briefly mention these surprising overlaps with other independent development without straying too far from the topic of memory hotplug.

8.1 Memory Defragmentation

The primary concern for memory defragmentation within the VM subsystem is at the page level. At the heart of this concern is the page allocator and management of contiguous groups of pages. Memory requests can be made for for sizes in the range of a single page up to $2^{(MAX\_ORDER-1)}$ contiguous pages. As time goes by, various size allocations are obtained and freed. The page allocator attempts to intelligently group adjacent pages via the use of buddy allocator as previously described. However, it still may become difficult to satisfy requests for large size allocations. When a suitable size block is not found on the free lists, an attempt is made to reclaim pages so that a sufficiently large block can assembled. Unfortunately, not all pages can be reclaimed. For example, those in use for kernel data. The free area splitting concepts previously discussed address this issue. By grouping pages based on usage characteristics, the likelihood that a large block of pages can be reclaimed and ultimately allocated is greatly increased.

With memory hotplug, removing a memory section is somewhat analogous to allocating all the pages within the memory section. This is because all pages within the section must be free (not in use) before the section can be removed. Therefore, the concept of free list splitting can also be applied to memory sections for memory removal operations. Unfortunately however, memory sections do not map directly to memory blocks managed by the page allocator. Rather, a memory section consists of multiple contiguous $2^{(MAX\_ORDER-1)}$ page size blocks. The number of blocks is dependent on architecture specific SECTION_SIZE and MAX_ORDER definitions. Future work within the memory hotplug project is to extend the concepts used to avoid fragmentation to that of memory section size blocks. This will increase

the likelihood that memory sections can be removed.

8.2 NUMA Memory Management

In a NUMA system, memory hotplug must consider the case where all of the memory on a node might be added/removed. Structures to manage the node must be updated.

In addition, a user can specify nodes which are used by a user's tasks by using `mbind()` or `set_mempolicy()` in order to support load balancing among cpusets/dynamic partitioning. Memory hotplug has to not only update mempolicy information, but also make interfaces for load balancing scripts to move memory contents from nodes to other appropriate nodes.

8.2.1 Hotplug of Management Structures for a Node.

Structures which manage memory of a node must be updated in order to hotplug the node This section describes some of the structures.

pgdat To reduce expensive cross node memory accesses, Linux uses `pgdat` structures which include zone and zonelists. These structures are allocated on each node's local memory in order to reduce access costs. If a new node is hotplug added, its `pgdat` structure should be allocated on its own node. Normally, there are no mm structures for the node until the pgdat is initialized, so pgdat has to be allocated by special routine early in the boot process. This allocation (getting a virtual address and mapping physical address to it) is like a `ioremap()`, but it should be mapped on cached area unlike `ioremap()`.

zonelist The zonelist is an array of zone addresses, and it is ordered by which zone should be used for its node. Its order is determined by access cost from a cpu to memory and the zone's attributes. This implies when a node with memory is hotplugged, all the node's zonelists which are being accessed must be updated. For updating, the options are:

- getting locks
- giving up reordering
- stop other cpus while updating

Stopping other cpus while updating may be the best way, because there is no impact on performance of page allocation unless a hotplug event is in progress. In addition, more structures than just zonelists need updating. For example, mempolicies of each process have to be updated to avoid using a removed node. To update them, the system has to remember all of the processes' mempolicies. Linux does not currently do this, so further development is necessary.

8.2.2 Scattered Clone Structures Among Nodes

Pgdat, which includes zone and zonelist, is used to manage its own node, but some of data structures' clones are allocated on each of the nodes for light weight accesses. One current example is `NODE_DATA()` on IA64 implementation. `NODE_DATA(node_id)` macro points to each `node_id`'s pgdat. In the IA64 implementation, `NODE_DATA(node_id)` is not just an array like it is in the IA32 implementation. This data is localized on each node and it can be obtained from `per_cpu` data. In this case, all of the nodes have clones of `pg_data_ptrs[]` array.

```
#define  local_node_data                    \
           local_cpu_data->node_data
#define  NODE_DATA(nid)                     \
           local_node_data->pg_data_ptrs[nid]
```

Besides `NODE_DATA()`, many other data structures which are often accessed are localized to each node. This implies that all of the node copies must also be updated when a hotplug event occurs. To update them, `stop_machine_run` may prove to be the best method of serializing access.

8.2.3 Process Migration on NUMA

It is important to determine what the best destination node is for migration of memory contents. This applies not only automatic migration, but also "manual page migration" as proposed by Ray Bryant at SGI. With manual page migration a load balancer script can specify the destination node for migrating existing memory. A potential interface would be a simple system call like `sys_migrate_pages(pid, oldnode, newnode)`.

However, if there are too many nodes (ex, 128 nodes) and tasks (ex, 256 processes) in the system, this system call will be called too frequently. Therefore, Ray Bryant is proposing an array interface to specify each node to avoid too many calls: `sys_migrate_pages(pid, count, old_nodes, new_nodes)`.

The arguments to `sys_migrate_pages()` `old_nodes` and `new_nodes` are the sets of source and destination nodes and count is the number of elements in each array. Therefore, a user can just call `sys_migrate_pages()` once for each task. If each task uses shared message blocks, there will be a large reduction in the number of system calls.

9 Conclusion

Hotplug Memory is real and achievable due to the dedication of its developers. This paper has shown that the issues with memory hotplug have been well thought out. Most of memory hotplug has already been implemented and is maintained in the `-mhp` tree[3]—broken down into the smallest possible independent pieces for continued development. These small pieces are released early and often. As individual pieces of this code become ready for consumption by the general public they are merged upstream. By maintaining this separate tree which is updated at least once per -rc release, hotplug developers have been able to test and stabilize an increasing amount of memory hotplug code. Thus, the pieces that get merged upstream are small, non-disruptive, and well tested.

Memory hotplug is a model of how a large, disruptive feature can be developed and merged into a continuously stable kernel. In fact, having a stable kernel has made development much more disciplined, debugging easier, conflicts with other developers easier to identify, feedback more thorough, and generally has been a blessing in disguise.

If in a parallel universe somewhere Andrew Morton gave his keynote today instead of a year ago I suspect he would say something different. The parallel universe Andrew Morton might say:

"Some features tend to be pervasive and have their little sticky fingers into lots of different places in the code base. An example of which comes to mind is CPU hot plug, and memory hot unplug. We may not be able to accept these features into a 2.7 development kernel due to their long-term impact on stabilizing that kernel. To make it easier on the developers of features like these we have decided to never have a

2.7 development kernel. Because of this, I expect CPU hot plug and memory hot unplug to be merged with relative ease as they reach the level of stability of the rest of the kernel."

10 Acknowledgments

Special thanks to Ray Bryant of SGI for his review on NUMA migration, New Energy and Industrial Technology Development Organization for funding some of the contributions from the authors who work at Fujitsu, Martin Bligh for his NUMA work and his work on IBMs test environment, Andy Whitcroft for his work on SPARSEMEM, Andrew Morton and the quilt developers for giving us something to manage all of these patches, Sourceforge for hosting our mailing list, OSDL for the work of the Hotplug SIG—especially their work on testing, and Martine Silbermann for valuable feedback.

11 Legal Statement

References

[1] `http://www.groklaw.net/article.php?story=20040802115731932`

[2] M. Tosatti (`marcelo.tosatti@cyclades.com`) (14 Oct 2004), *Patch: Migration Cache*. Email to Dave Hansen, Iwamoto Toshihiro, Hiroyuki Kamezawa, `linux-mm@kvack.org` (`http://lwn.net/Articles/106977/`)

[3] `http://sr71.net/patches/`

[4] C. Lameter (clameter@sgi.com) (21 Oct 2004) *Patch: Hugepages demand paging V1[0/4]: Discussion and Overview*. Email to Kenneth Chen, William Lee Irwin III, Ray Bryant, `linux-kernel@vger.kernel.org` (`http://lwn.net/Articles/107719/`)

[5] H. Takahashi, 2004 [online]. Linux memory hotplug for Hugepages. Available from: `http://people.valinux.co.jp/~taka/hpageremap.html` [Accessed 2004].

[6] D. Hansen, M. Kravetz, B. Christiansen, M. Tolentino. Hotplug Memory and the Linux VM. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Ontario, Canada, pages 278–294, July 2004.

[7] K. Knowlton. A Fast Storage Allocator. In *Communications of the ACM*, Vol. 8, Issue 10, pages 623–624, October 1965.

[8] M. Gorman. Understanding the Linux Virtual Memory Manager, Prentice Hall, NJ, 2004.

[9] W. Bolosky, R. Fitzgerald, M. Scott. Simple But Effective Techniques for

NUMA Memory Management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 19–31, 1989.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating System Principles*, pages 164–177, October 2003.

[11] E. Demaine, J. Munro. Fast Allocation and Deallocation with an Improved Buddy System. In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 84–96, 1999.

[12] Intel Corporation. 64-Bit Extension Technology Software Developer's Guide. 2004.

[13] K. Li and K. Peterson. Evaluation of Memory System Extensions. In *Proceedings of 18th Annual International Symposium on Computer Architecture*, pages 84–93, 1991.

[14] D. Mosberger, S. Eranian. ia-64 Linux Kernel Design and Implementation. Prentice Hall, NJ, 2002.

[15] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, S. Vaddagiri. Linux Kernel Hotpug CPU Support. In *Proceedings of the Ottawa Linux Symposium*, pages 467–479, July 2004.

[16] J. Peterson, T. Norman. Buddy Systems. In *Communications of the ACM*, Vol. 20, Issue 6, pages 421–431, June 1977.

[17] C. Waldspurger. Memory Resource Management in VMware ESX Server. In *Procedings of the 5th Symposium on Operating System Design and Implementation*, pages 181–194, 2002.

[18] M.S. Johnstone, P.R. Wilson. The Memory Fragmentation Problem: Solved? In *International Symposium on Memory Management*, pages 26–36, Vancouver, British Columbia, Canada, 1998.

[19] Linux Weekly News, 2005 [online]. Yet another approach to memory fragmentation. Available from `http://lwn.net/121618/` [Accessed Feb. 2005]

[20] ACPI Specification Version 3.0 `http://www.acpi.info/spec.htm`

[21] L. Brown, *et al*. The State of ACPI in the Linux Kernel. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Ontario, Canada, July 2005.

[22] B. Jacob, T. Mudge. Virtual Memory in Contemporary Microprocessors, IEEE Micro, pages 60–75, July 1998

[23] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, J. Chew. Machine Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, In *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[24] S. Hand. Self-Paging in the Nemesis Operating System, In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 73–86, February 1999.

[25] P. Denning. Virtual Memory, In *ACM Computing Surveys (CSUR)*, Vol. 2, Issue 3, pages 153–189, September 1970.

[26] A. Bensoussan, C. Clingen, R. Daley. The Multics Virtual Memory, In *Communications of the ACM*, Vol. 15, Issue 5, pages 308–318, May 1972.

[27] V. Abrossimov, M. Rozier. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 123–136, November 1989.

[28] A. Kleen. Porting Linux to x86-64. In *Proceedings of the Ottawa Linux Symposium,*, Ottawa, Ontario, Canada, July 2001.

[29] IBM. Dynamic Logical Partitioning in IBM eserver pSeries. October 2002.

[30] D.H. Brown Associates, Inc. Capacity on Demand A Requirement for the e-Business Environment. IBM White Paper. September 2003.

[31] L.D. Paulson. Computer System, Heal Thyself. In *IEEE Computer*, Vol. 35, Issue 8, August 2002.

Enhancements to Linux I/O Scheduling

Seetharami Seelam, Rodrigo Romero, Patricia Teller
University of Texas at El Paso
{seelam, romero, pteller}@cs.utep.edu

Bill Buros
IBM Linux Technology Center
wmb@us.ibm.com

Abstract

The Linux 2.6 release provides four disk I/O schedulers: deadline, anticipatory, noop, and completely fair queuing (CFQ), along with an option to select one of these four at boot time or runtime. The selection is based on *a priori* knowledge of the workload, file system, and I/O system hardware configuration, among other factors. The anticipatory scheduler (AS) is the default. Although the AS performs well under many situations, we have identified cases, under certain combinations of workloads, where the AS leads to process starvation. To mitigate this problem, we implemented an extension to the AS (called Cooperative AS or CAS) and compared its performance with the other four schedulers. This paper briefly describes the AS and the related deadline scheduler, highlighting their shortcomings; in addition, it gives a detailed description of the CAS. We report performance of all five schedulers on a set of workloads, which represent a wide range of I/O behavior. The study shows that (1) the CAS has an order of magnitude improvement in performance in cases where the AS leads to process starvation and (2) in several cases the CAS has performance comparable to that of the other schedulers. But, as the literature and this study reports, no one scheduler can provide the best possible performance for all workloads; accordingly, Linux provides four I/O schedulers from which to select. Even when dealing with just four, in systems that service concurrent workloads with different I/O behaviors, *a priori* selection of the scheduler with the best possible performance can be an intricate task. Dynamic selection based on workload needs, system configuration, and other parameters can address this challenge. Accordingly, we are developing metrics and heuristics that can be used for this purpose. The paper concludes with a description of our efforts in this direction, in particular, we present a characterization function, based on metrics related to system behavior and I/O requests, that can be used to measure and compare scheduling algorithm performance. This characterization function can be used to dynamically select an appropriate scheduler based on observed system behavior.

1 Introduction

The Linux 2.6 release provides four disk I/O schedulers: deadline, anticipatory, completely fair queuing (CFQ), and noop, along with an

option to select one of these at boot time or runtime. Selection is based on *a priori* knowledge of the workload, file system, and I/O system hardware configuration, among other factors. In the absence of a selection at boot time, the anticipatory scheduler (AS) is the default since it has been shown to perform better than the others under several circumstances [8, 9, 11].

To the best of our knowledge, there are no performance studies of these I/O schedulers under workloads comprised of concurrent I/O requests generated by different processes that exhibit different types of access patterns and methods. We call these types of workloads "mixed workloads." Such studies are of interest since, in contemporary multiprogramming/multiprocessor environments, it is quite natural to have several different types of I/O requests concurrently exercising the disk I/O subsystem. In such situations, it is expected that the I/O scheduler will not deprive any process of its required I/O resources even when the scheduler's performance goals are met by the processing of concurrent requests generated by other processes. In contrast, due to the anticipatory nature of the AS, there are situations, which we identify in this paper, when the AS leads to process starvation; these situations occur when a mixed workload stresses the disk I/O subsystem. Accordingly, this paper answers the following three questions and addresses a fourth one, which is posed in the next paragraph.

Q1. Are there mixed workloads that potentially can starve under the AS due to its anticipatory nature?

Q2. Can the AS be extended to prevent such starvation?

Q3. What is the impact of the extended scheduler on the execution time of some realistic benchmarks?

In this paper we also explore the idea of dynamic scheduler selection. In an effort to determine the best scheduler, [13] quantifies the performance of the four I/O schedulers for different workloads, file systems, and hardware configurations. The conclusion of the study is that there is "no silver bullet," i.e., none of the schedulers consistently provide the best possible performance under different workload, software, and hardware combinations. The study shows that (1) for the selected workloads and systems, the AS provides the best performance for sequential read requests executed on single disk hardware configurations; (2) for moderate hardware configurations (RAID systems with 2-5 disks), the deadline and CFQ schedulers perform better than the others; (3) the noop scheduler is particularly suitable for large RAID (e.g., RAID-0 with tens of disks) systems consisting of SCSI drives that have their own scheduling and caching mechanisms; and (4) the AS and deadline scheduler provide substantially good performance in single disk and 2-5 disk configurations; sometimes the AS performs better than the deadline scheduler and vice versa. The study infers that to get the best possible performance, scheduler selection should be dynamic. So, the final question we address in this paper is:

Q4. Can metrics be used to guide dynamic selection of I/O schedulers?

The paper is organized as follows. Section 2 describes the deadline and anticipatory schedulers, highlighting similarities and differences. The first and second questions are answered in Sections 3 and 4, respectively, by demonstrating that processes can potentially starve under the AS and presenting an algorithm that extends the AS to prevent process starvation. To answer the third question, Section 5 presents a comparative analysis of the deadline scheduler, AS, and extended AS for a set of mixed workloads. Furthermore, the execution times

of a set of benchmarks that simulate web, file, and mail servers, and metadata are executed under all five schedulers are compared. Finally, the fourth question is addressed in Section 6, which presents microbenchmark-based heuristics and metrics for I/O request characterization that can be used to guide dynamic scheduler selection. Sections 7, 8, and 9 conclude the paper by highlighting our future work, describing related work, and presenting conclusions, respectively.

2 Description of I/O Schedulers

This section describes two of the four schedulers provided by Linux 2.6, the deadline scheduler and the anticipatory scheduler (AS). The deadline scheduler is described first because the AS is built upon it. Similarities and differences between the two schedulers are highlighted. For a description of the CFQ and noop schedulers, refer to [13].

2.1 Deadline Scheduler

The deadline scheduler maintains two separate lists, one for read requests and one for write requests, which are ordered by logical block number—these are called the *sort lists*. During the enqueue phase, an incoming request is assigned an expiration time, also called *deadline*, and is inserted into one of the sort lists and one of two additional queues (one for reads and one for writes) ordered by expiration time—these are called the *fifo lists*. Scheduling a request to a disk drive involves inserting it into a dispatch list, which is ordered by block number, and deleting it from two lists, for example, the read fifo and read sort lists. Usually a set of contiguous requests is moved to the dispatch list. A request that requires a disk seek is counted as 16 contiguous requests. Requests are selected by the scheduler using the algorithm presented below.

Step 1: If there are write requests in the write sort list and the last two scheduled requests were selected using step 2 and/or step 3, then select a set of write requests from the write sort list and exit.

Step 2: If there are read requests with expired deadlines in the read fifo list, then select a set of read requests from this list and exit.

Step 3: If there are read requests in the read sort list, then select a set of read requests from this list and exit.

Step 4: If there are write requests in the write sort list, then select a set of write requests from this list and exit.

When the scheduler assigns deadlines, it gives a higher preference to reads; a read is satisfied within a specified period of time—500ms is the default—while a write has no strict deadline. In order to prevent write request starvation, which is possible under this policy, writes are scheduled after a certain number of reads.

The deadline scheduler is work-conserving—it schedules a request as soon as the previous request is serviced. This can lead to potential problems. For example, in many applications read requests are *synchronous*, i.e., successive read requests are separated by small chunks of computation, and, thus, successive read requests from a process are separated by a small delay. If p $(p > 1)$ processes of this type are executing concurrently, then if p requests, one from each process, arrive during a time interval, a work-conserving scheduler may first select a request from one process and then select a request from a different process. Consequently, the work-conserving nature of the deadline scheduler may result in deceptive

idleness [7], a condition in which the scheduler alternately selects requests from multiple processes that are accessing disjoint areas of the disk, resulting in a disk seek for each request. Such deceptive idleness can be eliminated by introducing into the scheduler a short delay before I/O request selection; during this time the scheduler waits for additional requests from the process that issued the previous request. Such schedulers are called non-work-conserving schedulers because they trade off disk utilization for throughput. The anticipatory scheduler, described next in Section 2.2, is an example of such a scheduler. The deceptive idleness problem, with respect to the deadline scheduler, is illustrated in Section 5, which presents experimental results for various microbenchmarks and real workloads. A study of the performance of the deadline scheduler under a range of workloads also is presented in the same section.

The Linux 2.6 deadline scheduler has several parameters that can be tuned to obtain better disk I/O performance. Some of these parameters are the deadline time for read requests (`read_expire`), the number of requests to move to the dispatch list (`fifo_batch`), and the number of times the I/O scheduler assigns preference to reads over writes (`write_starved`). For a complete description of the deadline scheduler and various tunable parameters, refer to [13].

2.2 Anticipatory Scheduler

The *seek-reducing anticipatory scheduler* is designed to minimize the number of seek operations in the presence of synchronous read requests and eliminate the deceptive idleness problem [7]. Due to some licensing issues [14], the Linux 2.6 implementation of the AS, which we refer to as *LAS*, is somewhat different from the general idea described in [7]. Nonetheless, the LAS follows the same basic idea, i.e., if the disk just serviced a read request from process p then stall the disk and wait (some period of time) for more requests from process p.

The LAS is comprised of three components: (1) the original, non-anticipatory disk scheduler, which is essentially the deadline scheduler algorithm with the deadlines associated with requests, (2) the anticipation core, and (3) the anticipation heuristic. The latter two serve read requests. After scheduling a request for dispatch, the deadline scheduler selects a pending I/O request for dispatch. In contrast, the LAS, selects a pending I/O request, using the same criteria as the deadline scheduler, and evaluates it via its anticipation heuristic.

The anticipation core maintains statistics related to all I/O requests and decaying frequency tables of exit probabilities, mean process seek distances, and mean process think times. The *exit probability* indicates the probability that an *anticipated request*, i.e., a request from the *anticipated process*, i.e., the process that generated the last request, will not arrive. Accordingly, it is decremented when an anticipated request arrives and is incremented otherwise, e.g., when a process terminates before generating a subsequent I/O request. If the exit probability exceeds a specified threshold, any request that arrives at the anticipation core is scheduled for dispatch. The seek distance (think time) is the difference between the logical block numbers (arrival times) of two consecutive requests. These metrics—exit probability, mean process seek distance, and mean process seek time—are used by the anticipation heuristic, in combination with current head position and requested head position, to determine anticipation time.

The anticipation heuristic evaluates whether to stall the disk for a specific period of time (*wait period* or *anticipation time*), in anticipation of a “better request", for example from the anticipated process, or to schedule the selected

request for dispatch. The anticipation heuristic used in the LAS is based on the shortest positioning time first (SPTF) scheduling policy. Given the current head position, it evaluaes which request, anticipated or selected, will potentially result in the shortest seek distance. This evaluation is made by calculating the positioning times for both requests. If the logical block of the selected request is close to the current head position, the heuristic returns zero, which causes the request to be scheduled for dispatch. Otherwise, the heuristic returns a positive integer, i.e., the anticipation time, the time to wait for an anticipated request. Since synchronous requests are initiated by a single process with interleaved computation, the process that issued the last request may soon issue a request for a nearby block.

During the anticipation time, which usually is small (a few milliseconds—6ms is the default) and can be adjusted, the scheduler waits for the anticipated request. If a new request arrives during the wait period, it is evaluated immediately with the anticipation heuristic. If it is the anticipated request, the scheduler inserts it into the dispatch list. Otherwise, the following algorithm is executed. If the algorithm does not result in the new request being scheduled for dispatch, the core continues to anticipate and the disk is kept idle; this leads to potential problems, some of which are described in Section 3.

Step 1: If anticipation has been turned off, e.g., as a result of a read request exceeding its deadline, then update process statistics, schedule the starving request for dispatch, and exit.

Step 2: If the anticipation time has expired then update process statistics, schedule the request for dispatch, and exit.

Step 3: If the anticipated process has terminated, update the exit probability, update process statistics, schedule the new request for dispatch, and exit.

Step 4: If the request is a read request that will access a logical block that is "close" to the current head position, then update process statistics, schedule the new request for dispatch, and exit. In this case, there is no incentive to wait for a "better" request.

Step 5: If the anticipated process just started I/O and the exit probability is greater than 50%, update process statistics, schedule the new request for dispatch, and exit; this process may exit soon, thus, there is no added benefit in further anticipation. This step creates some problems, further described in Section 3, when cooperative processes are executing concurrently with other processes.

Step 6: If the mean seek time of the anticipated process is greater than the anticipation time, update process statistics, schedule the request for dispatch, and exit.

Step 7: If the mean seek distance of the anticipated process is greater than the seek distance required to satisfy the new request, update process statistics, schedule the request for dispatch, and exit.

Unlike the deadline scheduler, the LAS allows limited back seeks. A back seek occurs when the position of the head is in front of the head position required to satisfy the selected request. In deadline and other work-conserving schedulers, such requests are placed at the end of the queue. There is some cost involved in back seeks, thus, the number of back seeks is limited to MAXBACK(1024*1024) sectors; see [13] for more information.

As described, the essential differences between the LAS and deadline scheduler are the anticipation core and heuristics, and back seeks. Performance of the LAS under a range of workloads is studied in Section 5, which highlights its performance problems.

3 Anticipatory Scheduler Problems

The anticipatory scheduler (LAS) algorithm is based on two assumptions: (1) synchronous disk requests are issued by individual processes [7] and, thus, anticipation occurs only with respect to the process that issued the last request; and (2) for anticipation to work properly, the anticipated process must be alive; if the anticipated process dies, there is no further anticipation for requests to nearby sectors. Instead, any request that arrives at the scheduler is scheduled for dispatch, irrespective of the requested head position and the current head position. These two assumptions hold true as long as synchronous requests are issued by individual processes. However, when a group of processes collectively issue synchronous requests, the above assumptions are faulty and can result in (1) faulty anticipation, but not necessarily bad disk throughput, and (2) a seek storm when multiple sets of short-lived groups of processes, which are created and terminated in a very short time interval, issue synchronous requests collectively and simultaneously to disjoint sets of disk area, resulting in poor disk throughput. We call processes that collectively issue synchronous requests to a nearby set of disk blocks *cooperative processes*. Examples of programs that generate cooperative processes include shell scripts that read the Linux source tree, different instances of *make* scripts that compile large programs and concurrently read a small set of source files, and different programs or processes that read several database records. We demonstrate related behavior and associated performance problems using the two examples below.

3.1 Concurrent Streaming and Chunk Read Programs

First, we demonstrate how the first assumption of the LAS can lead to process starvation. Consider two programs, A and B, presented in Figure 2. Program A generates a stream of synchronous read requests by a single process, while program B generates a sequence of dependent chunk read requests, each set of which is generated by a different process.

Assume that Program B is reading the top-level directory of the Linux source tree. The program reads all the files in the source tree, including those in the subdirectories, one file at a time, and does not read any file outside the top-level directory. Note that each file is read by a different process, i.e., when Program B is executed, a group of processes is created, one after the other, and each issues synchronous disk read requests. For this program, consider the performance effect of the first assumption, i.e., the per-process anticipation built into the LAS. Recall that LAS anticipation works only on a per-process basis and provides improved performance only under multiple outstanding requests that will access disjoint sets of disk blocks. When Program A or B is executed while no other processes are accessing the disk, anticipation does not reap a benefit because there is only a small set of pending I/O requests (due to prefetching) that are associated with the executing program. There are no disk head seeks that are targets for performance improvement.

Now consider executing both programs concurrently. Assume that they access disjoint disk blocks and the size of the `big-file` read by Program A is larger than that of the buffer cache. In this case, each read request results in a true disk access rather than a read from the buffered file copy. Since the two programs are executing concurrently, at any point in time there are at least two pending I/O requests, one generated by each of the processes. Program B sequentially creates multiple processes that access the disk and only a small set of the total number of I/O requests generated by Program

B corresponds to a single process; all read requests associated with a particular file are generated by one process. In contrast, the execution of Program A involves only one process that generates all I/O requests. Since the anticipation built into the LAS is associated with a process, it fails to exploit the disk spatial locality of reference of read requests generated by the execution of Program B; however, it works well for the requests generated by Program A. More important is the fact that concurrent execution of these two programs results in starvation of processes generated by Program B. Experimental evidence of this is presented in Section 5.

3.2 Concurrent Chunk Read Programs

This section demonstrates how the second assumption of the LAS can fail and, hence, lead to poor disk throughput. Consider the concurrent execution of two instances of Program B, instances 1 and 2, reading the top-level directory of two separate Linux source trees that are stored in disjoint sets of disk blocks. Assume that there are F files in each source tree. Accordingly, each instance of Program B creates F different processes sequentially, each of which reads a different file from the disk.

For this scenario, consider the performance effect of the second assumption, i.e., once the anticipated process terminates, anticipation for requests to nearby sectors ceases. When two instances of program B are executing concurrently, at any point in time there are at least two pending I/O requests, one generated by each program instance. Recall that requests to any one file correspond to only one process. In this case, the anticipation works well as long as only processess associated with one program instance, say instance 1, are reading files. When there are processess from the two instances reading files then the second assumption does not allow the scheduler to exploit the disk spatial locality of reference of read requests generated by another process associated with instance 1. For example, given pending I/O requests generated by two processes, one associated with instance 1 and one associated with instance 2, anticipation will work well for each process in isolation. However, once a process from one instance, say instance 1, terminates, even if there are pending requests from another process of instance 1, the scheduler schedules for dispatch a request of the process of instance 2. This results in a disk seek and anticipation on the instance 2 process that generated the request. This behavior iterates for the duration of the execution of the programs. As a result, instead of servicing all read requests corresponding to one source tree and, thus, minimizing disk seeks, an expensive sequence of seeks, caused by alternating between processes of the two instances of Program B, occurs. For this scenario, at least $2F - 1$ seeks are necessary to service the requests generated by both instances of Program B. As demonstrated, adherence to the second assumption of the LAS leads to seek storms that result in poor disk throughput. Experimental evidence of this problem is presented in Section 5.

4 Cooperative Anticipatory Scheduler

In this section we present an extension to the LAS that addresses the faulty assumptions described in Section 3 and, thus, solves the problems of potential process starvation and poor disk throughput. We call this scheduler the Cooperative Anticipatory Scheduler (CAS). To address potential problems, the notion of anticipation is broadened. When a request arrives at the anticipation core during an anticipation

time interval, irrespective of the state of the anticipated process (alive or dead) and irrespective of the process that generated the request, if the requested block is near the current head position, it is scheduled for dispatch and anticipation works on the process that generated the request. In this way, anticipation works not only on a single process, but on a group of processes that generate synchronous requests. Accordingly, the first assumption of the LAS and the associated problem of starvation of cooperative processes is eliminated. Since the state of the anticipated process is not taken into account in determining whether or not to schedule a new request for dispatch, short-lived cooperative processes accessing disjoint disk block sets do not prevent the scheduler from exploiting disk spatial locality of reference. Accordingly, the second assumption is broadened and the associated problem of reduced disk throughput is eliminated.

The CAS algorithm appears below. As in the LAS algorithm, during anticipation, if a request from the anticipated process arrives at the scheduler, it is scheduled for dispatch immediately. In contrast to the LAS, if the request is from a different process, before selecting the request for scheduling or anticipating for a better request, the following steps are performed in sequence.

Step 1: If anticipation has been turned off, e.g., as a result of a read request exceeding its deadline, then update process statistics, schedule the starving request for dispatch, and exit.

Step 2: If the anticipation time has elapsed, then schedule the new request, update process statistics and exit.

Step 3: If the anticipation time has not elapsed and the new request is a read that accesses a logical block number "close" to the current head position, schedule the request for dispatch and exit. A request is considered close if the requested block number is within some delta distance from the current head position or the process' mean seek distance is greater than the seek distance required to satisfy the request. Recall that this defines a request from a cooperative process. At this point in time the anticipated process could be alive or dead. If it is dead, update the statistics for the requesting process and increment the CAS *cooperative exit probability*, which indicates the existence of cooperative processes related to dead processes. If the anticipated process is alive, update the statistics for both processes and increment the cooperative exit probability.

Step 4: If the anticipated process is dead, update the system exit probability and if it is less than 50% then schedule the new request and exit. Note that this request is not from a cooperative process.

Step 5: If the anticipated process just started I/O, the system exit probability is greater than 50%, and the cooperative exit probability is less than 50%, schedule the new request and exit.

Step 6: If the mean think time of the anticipated process is greater than the anticipation time, schedule the new request and exit.

This concludes the extensions to the anticipatory scheduler aimed at solving the process starvation and reduced throughput problems.

5 Experimental Evaluation

This section first presents a comparative performance analysis, using a set of mixed workload microbenchmarks, of the deadline scheduler, LAS, and CAS. The workloads are described in Sections 5.4, 5.5, and 5.6. The goal of the analysis is to highlight some of the problems with the deadline scheduler and LAS, and to show

that the CAS indeed solves these problems. Second, we compare the execution times, under all five schedulers, of a set of benchmark profiles that simulate web, file, and mail servers, and metadata. A general description of these profiles is provided in Section 5.1 and individual workloads are described in Sections 5.7-5.10. The goal of this comparison is to show that the CAS, in fact, performs better or as good as the LAS under workloads with a wide range of characteristics. Using these benchmarks, we show that (1) the LAS can lead to process starvation and reduced disk throughput problems that can be mitigated by the CAS, and (2) under various workload scenarios, which are different from those used to demonstrate process starvation or reduced throughput, the CAS has performance comparable to the LAS.

5.1 Workload Description

The Flexible File System Benchmark (FFSB) infrastructure [6] is the workload generator used to simulate web, file, and mail servers, and metadata. The workloads are specified using profiles that are input to the FFSB infrastructure, which simulates the required I/O behavior. Initially, each profile is configured to create a total of 100,000 files in 100 directories. Each file ranges in size from 4 KB to 64 KB; the total size of the files exceeds the size of system memory so that the random *operations* (file read, write, append, create, or delete actions) are performed from disk and not from memory. File creation time is not counted in benchmark execution time. A profile is configured to create four threads that randomly execute a total of 80,000 operations (20,000 per thread) on files stored in different directories. Each profile is executed three times under each of the five schedulers on our experimental platform (described in Section 5.2). The average of the three execution times, as well as the standard deviation, are reported for each scheduler.

5.2 Experimental Platform

We conducted the following experiments on a dual-processor (2.28GHz Pentium 4 Xeon) system, with 1 GB main memory and 1 MB L2 cache, running Linux 2.6.9. Only a single processor is used in this study. In order to eliminate interference from operating system (OS) I/O requests, benchmark I/O accesses an external 7,200 RPM Maxtor 20 GB IDE disk, which is different from the disk hosting the OS. The external drive is configured with the `ext3` file system and, for every experiment, is unmounted and re-mounted to remove buffer cache effects.

5.3 Metrics

For the microbenchmark experiments, two application performance metrics, application execution time (in seconds) and aggregate disk throughput (in MB/s), are used to demonstrate the problems with different schedulers. With no other processes executing in the system (except daemons), I/O-intensive application execution time is inversely proportional to disk throughput. In such situations, the scheduler with the smallest application execution time is the best scheduler for that workload. In mixed workload scenarios, however, the execution time of any one application cannot be used to compare schedulers. Due to the non-work-conserving nature of the LAS and CAS, these schedulers, when serving I/O requests, introduce delays that favor one application over another, sometimes at the cost of increasing the execution times of other applications. Hence, in the presence of other I/O-intensive processes, the application execution time metric must be coupled with other metrics to quantify the relative merit of different schedulers. Consequently, we use the aggregate disk throughput metric in such scenarios. Application execution time indicates the performance of a single application

Program 1:
```
while true
do
   dd if=/dev/zero of=file \
      count=2048 bs=1M
done
```

Program 2:
```
time cat 200mb-file > /dev/null
```

Figure 1: Program 1—generates stream write requests; Program—2 generates stream read requests

and disk throughput indicates overall disk performance. Together, these two metrics help expose potential process starvation and reduced throughput problems with the LAS.

5.4 Experiment 1: Microbenchmarks—Streaming Writes and Reads

This experiment uses a mixed workload comprised of two microbenchmarks [9], shown in Figure 1, to compare the performance of the deadline scheduler, LAS, and CAS. It demonstrates the advantage of the LAS and CAS over the deadline scheduler in a mixed workload scenario. One microbenchmark, Program 1, generates a stream of write requests, while the other, Program 2, generates a stream of read requests. Note that the write requests generated by Program 1 are asynchronous and can be delayed to improve disk throughput. In contrast, Program 2 generates synchronous stream read requests that must be serviced as fast as possible.

When Programs 1 and 2 are executed concurrently under the three different schedulers, experimental results, i.e., application execution times and aggregate disk throughput, like those shown in Table 1 are attained. These results indicate the following. (1) For synchronous read requests, the LAS performs an order of magnitude better, in terms of execution time, and it provides 32% more disk throughput than the deadline scheduler. (2) The CAS has performance similar to that of the LAS.

The LAS and CAS provide better performance than the deadline scheduler by reducing unnecessary seeks and serving read requests as quickly as possible. For many such workloads, these schedulers improve request latency and aggregate disk throughput.

Scheduler	Execution Time (sec.)	Throughput (MB/s)
Deadline	129	25
LAS	10	33
CAS	9	33

Table 1: Performance of Programs 1 and 2 under the Deadline Scheduler, LAS, and CAS

5.5 Experiment 2: Microbenchmarks—Streaming and Chunk Reads

To compare the performance of the deadline scheduler, LAS, and CAS, illustrate the process starvation problem of the LAS, and show that the CAS solves this problem, this experiment uses a mixed workload microbenchmark comprised of two microbenchmarks [9], shown in Figure 2. One microbenchmark, Program A, generates a stream of read requests, while the other, Program B, generates a sequence of dependent chunk read requests. Concurrent execution of the two programs results in concurrent generation of read requests from each program. Thus, assume that the read requests of these two programs are interleaved. In general, the servicing of a read request from one of the programs will be followed by an expensive seek

Program A:
```
while true
do
   cat big-file > /dev/null
done
```

Program B:
```
time find . -type f -exec \
        cat '{}' ';' > /dev/null
```

Figure 2: Program—A generates stream read requests; Program—B generates chunk read requests

in order to service a request from the other program; this situation repeats until one program terminates. However, if a moderate number of requests are anticipated correctly, the number of expensive seeks is reduced. For each correct anticipation, two seek operations are eliminated; an incorrect anticipation costs a small delay. Accordingly, anticipation can be advantageous for a workload that generates dependent read requests, i.e., that exhibit disk spatial locality of reference. However, as described previously, the LAS can anticipate only if dependent read requests are from the same process. In this experiment the dependent read requests of Program A are from the same process, while the dependent chunk read requests of Program B are from different processes.

Assume that Program B is reading the top-level directory of the Linux source tree, as described in Section 3.1. In this case, the *find* command finds each file in the directory tree, then the *cat* command (spawned as a separate process) issues a read request to read the file, with the file name provided by the *find* process from the disk. The new *cat* process reads the entire file, then the file is closed. This sequence of actions is repeated until all files in the directory are read.

Note that, in this case, each file read operation is performed by a different process, while LAS anticipation works only on a per-process basis. Thus, if these processes are the only ones accessing the disk, there will be no delays due to seek operations to satisfy other processes. However, when run concurrently with Program A, the story is different, especially if, to eliminate disk cache effects, we assume that the `big-file` read by Program A is larger than the buffer cache. Note that during the execution of Program A a single process generates all read requests.

When these two programs are executed concurrently, anticipation works really well for the streaming reads of Program A but it does not work at all for the dependent chunk reads of Program B. The LAS is not able to recognize the dependent disk spatial locality of reference exhibited by the *cat* processes of Program B; this leads to starvation of these processes. In contrast, the CAS identifies this locality of reference and, thus, as shown in Table 2, provides better performance both in terms of execution time and aggregate disk throughput. In addition, it does not lead to process starvation.

Scheduler	Execution Time (sec.)	Throughput (MB/s)
Deadline	297	9
LAS	4767	35
CAS	255	34

Table 2: Performance of Program A and B under the Deadline Scheduler, LAS, and CAS

The results in Table 2 show the following. (1) The LAS results in very bad execution time; this is likely because LAS anticipation does not work for Program B and, even worse, it works really well for Program A, resulting in good disk utilization for Program A and a very small amount of disk time being allocated for requests from Program B. (2) The execution

time under the deadline scheduler is 16 times smaller than that under the LAS; this is likely because there is no anticipation in the deadline scheduler. (3) Aggregate disk throughput under the deadline scheduler is 3.9 times smaller than under the LAS; this is likely because LAS anticipation works really well for Program A. (4) The CAS alleviates the anticipation problems exhibited in the LAS for both dependent chunk reads (Program B) and dependent read workloads (Program A). As a result, CAS provides better execution time and aggregate disk throughput.

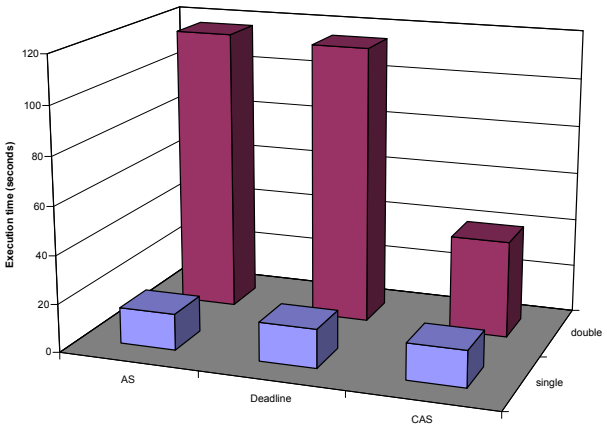

Figure 3: Reading the Linux Source: multiple, concurrent instances cause seek storms with the deadline scheduler and LAS, which are eliminated by the CAS

5.6 Experiment 3: Microbenchmarks—Chunk Reads

To illustrate the reduced disk throughput problem of the deadline scheduler and LAS and to further illustrate the performance of the CAS, this experiment first uses one instance of a microbenchmark that generates a sequence of dependent chunk reads and then uses two concurrently executing instances of the same program, Program B of Figure 2, that access disjoint Linux source trees. The results of this experiment are shown in Table 3 and Figure 3.

Scheduler	Throughput (MB/s)	
	1 Instance	2 Instances
Deadline	14.5	4.0
LAS	15.5	4.0
CAS	15.5	11.6

Table 3: Chunk Reads under the Deadline Scheduler, LAS, and CAS

As described before, in Program B a different *cat* process reads each of the files in the source tree, thus, each execution of the program generates, in sequence, multiple processes that have good disk spatial locality of reference. With two concurrently executing instances of Program B accessing disjoint sections of the disk, the deadline scheduler seeks back and forth several thousand times. The LAS is not able to identify the dependent read requests generated by the different *cat* processes and, thus, does not anticipate for them. As a result, like the deadline scheduler, the LAS becomes seek bound. In contrast, the CAS captures the dependencies and, thus, provides better disk throughput and execution time. Recall that, in this case, throughput is inversely proportional to execution time.

As shown in Figure 3 and Table 3, with one instance of Program B the three schedulers have a performance difference of about 7%. One would normally expect the execution time to double for two instances of the program, however, for the reasons described above the deadline scheduler, LAS, and CAS increase their execution times by a factor of 7, 7, and 2.5, respectively. Again, the CAS has a smaller factor increase (2.5) in execution time because it detects the dependencies among cooperative processes working in a localized area of disk and, thus, precludes the seek storms that occur otherwise in the deadline scheduler and LAS.

5.7 Experiment 4: Web Server Benchmark

This benchmark simulates the behavior of a web server by making read requests to randomly selected files of different sizes. The mean of the three execution times for each scheduler are reported in Figure 4 and Table 4. The numbers in the table are in seconds, and bold numbers indicate the scheduler with the best execution time for each benchmark. It is worthwhile to point out that the standard deviations of the results are less than 4% of the average values, which is small for all practical purposes. From the table we can conclude that the CAS has the best performance of all the schedulers in the case of random reads and the CFQ has the worst performance. The LAS has very good execution time performance which is comparable to that of CAS; it trails the CAS by less than 1%. The deadline, CFQ, and noop schedulers trail the CAS by 8%, 8.9%, and 6.5%, respectively.

Scheduler	Web Server	Mail Server	File Server	Meta Data
Deadline	924	118	1127	305
LAS	863	177	916	295
CAS	**855**	**109**	**890**	288
CFQ	931	112	1099	**253**
noop	910	125	1127	319

Table 4: Mean Execution Times (seconds) of Different Benchmark Programs

5.8 Experiment 5: File Server Benchmark

This benchmark simulates the behavior of a typical file server by making random read and write operations in the proportions of 80% and 20%, respectively. The average of the three execution times are reported in Figure 4 and Table 4. The standard deviations of the results are less than 4.5% of the average values. Here we

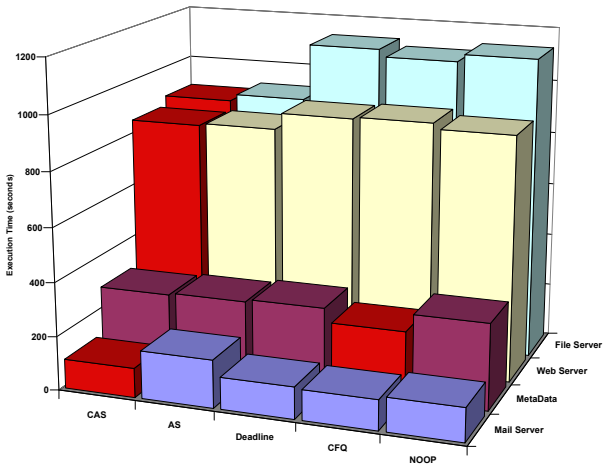

Figure 4: Mean Execution Time (seconds) on `ext3` File System

can conclude that the CAS has the best performance; the LAS trails the CAS by 2.9%; and the other schedulers trail the CAS by at least 23%.

5.9 Experiment 6: Mail Server Benchmark

This benchmark simulates the behavior of a typical mail server by executing random file read, create, and delete operations in the proportions of 40%, 40%, and 20%, respectively. The average of the three execution times are reported in Figure 4 and Table 4. The standard deviations of the results are less than 3.5% of the average values except for the LAS for which the standard deviation is about 11%. From these results we can conclude that the CAS has the best performance and the LAS has the worst performance, the LAS trails the CAS by more than 62%. The CFQ scheduler has very good execution time performance compared to the CAS; it trails by a little more than 3%. The deadline and noop schedulers trail the CAS by 8% and 14%, respectively.

5.10 Experiment 7: MetaData Program

This benchmark simulates the behavior of a typical MetaData program by executing random file create, write-append, and delete operations in the proportions of 40%, 40%, and 20%, respectively. Note that in this benchmark there are no read requests. The average of the three execution times are reported in Figure 4 and Table 4. The standard deviations of the results are less than 3.5% of the average values except for the noop scheduler for which the standard deviation is 7.7%. From these results, we can conclude that the CFQ scheduler has the best performance. The LAS trails the CAS by 2%. The deadline, LAS, CAS, and noop schedulers trail the CFQ, the best, scheduler by as much as 26%.

6 I/O Scheduler Characterization for Scheduler Selection

Our experimentation (e.g., see Figure 4) as well as the study in [13] reveals that no one scheduler can provide the best possible performance for different workload, software, and hardware combinations. A possible approach to this problem is to develop one scheduler that can best serve different types of these combinations, however, this may not be possible due to diverse workload requirements in real world systems [13, 15]. This issue is further complicated by the fact that workloads have orthogonal requirements. For example, some workloads, such as multimedia database applications, prefer fairness over performance, otherwise streaming video applications may suffer from discontinuity in picture quality. In contrast, server workloads, such as file servers, demand performance over fairness since small delays in serving individual requests are well tolerated. In order to satisfy these conflicting requirements, operating systems provide multiple I/O schedulers—each suitable for a different class of workloads—that can be selected, at boot time or runtime, based on workload characteristics.

The Linux 2.6.11 kernel provides four different schedulers and an option to select one of them at boot time for the entire I/O system and switch between them at runtime on a per-disk basis [2]. This selection is based on *a priori* understanding of workload characteristics, essentially by a system administrator. Moreover, the scheduler selection varies based on the hardware configuration of the disk (e.g., RAID setup), software configuration of the disk, i.e., file system, etc. Thus, static or dynamic scheduler selection is a daunting and intricate task. This is further complicated by two other factors. (1) Systems that execute different kinds of workloads concurrently (e.g., a web server and a file server)—that require, individually, a different scheduler to obtain best possible performance—may not provide best possible performance with a single scheduler selected at boot time or runtime. (2) Similarly, workloads with different phases, each phase with different I/O characteristics, will not be best served by *a priori* scheduler selection.

We propose a scheduler selection methodology that is based primarily on runtime workload characteristics, in particular the average request size. Ideally, dynamic scheduler selection would be transparent to system hardware and software. Moreover, a change in hardware or software configurations would be detected automatically and the scheduler selection methodology would re-evaluate the scheduler choice. With these goals in mind, we describe below ideas towards the realization of a related methodology.

We propose that runtime scheduler selection be based on *a priori* measurements of disk

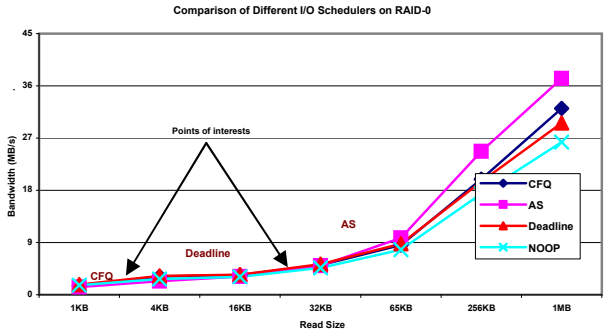

Figure 5: Scheduler Ranking using a Microbenchmark

throughput under the various schedulers and request sizes. These measurements are then used to generate a function that at runtime, given the current average request size, returns the scheduler that gives the best measured throughput for the specified disk. Using the four schedulers on our experimental system, described in Section 5.2, augmented by a RAID-0 device with four IDE 10 GB drives, we took *a priori* measurements by executing a program that creates and randomly reads data blocks of various sizes from several large files. The system is configured with the ext3 file system; it runs the Linux 2.6.11 kernel to permit switching schedulers. The ranking of the schedulers based on average request size and disk throughput is shown in Figure 5. Experiments using this proposed methodology to guide dynamic scheduler selection are in progress.

7 Future Work

Our cooperative anticipatory scheduler eliminates the starvation problem of the AS by scheduling requests from other processes that access disk blocks close to the current head position. It updates statistics related to requests on a per-process basis such that future-scheduling decisions can be made more appropriately. This scheduler has several tunable parameters, e.g., the amount of time a request spends in the queue before it is declared expired and the amount of time the disk is kept idle in anticipation of future requests.

Because we were interested in investigating the possible starvation problem and proposing a solution, we did not investigate the effects of changing these parameters; however, we have begun to do so. We are especially interested in studying the performance effects of the CAS, with various parameter sets, on different disk systems. Given that the study shows that different parameter sets provide better performance for systems with different disk sizes and configurations [13], a methodology for dynamically selecting parameters is our goal. Furthermore, we intend to experiment with maintaining other statistics that can aid in making scheduling decisions for better disk performance. An example statistic is the number of times a process consumes its anticipation time; if such a metric exceeds a certain threshold, it indicates that there is a mismatch between the workload access pattern and the scheduler and, hence, such a process should not be a candidate for anticipation.

With these types of advances, we will develop a methodology for automatic and dynamic I/O scheduler selection to meet application needs and to maximize disk throughput.

8 Related Work

To our knowledge the initial work on anticipatory scheduling, demonstrated on FreeBSD Unix, was done in [7]. Later, the general idea was implemented in the Linux kernel by Nick Piggin and was tested by Andrew Martin [11]. To our surprise, during the time we were exploring the I/O scheduler, the potential starva-

tion problem was reported to the Linux community independently on Linux mailing lists, however, no action was taken to fix the problem [3].

Workload dependent performance of the four I/O schedulers is presented in [13]; this work points out some performance problems with the anticipatory scheduler in the Linux operating system. There is work using genetic algorithms, i.e., natural evolution, selection, and mutation, to tune various I/O scheduler parameters to fit workload needs [12]. In [15] the authors explore the idea of using seek time, average waiting time in the queue, and the variance in average waiting time in a utility function that can be used to match schedulers to a wide range of workloads. This resulted in the development of a maximum performance two-policy algorithm that essentially consists of two schedulers, each suitable for different ranges of workloads. There also have been attempts [2] to include in the CFQ scheduler priorities and time slicing, analogous processor scheduler concepts, along with the anticipatory statistics. This new scheduler, called Time Sliced CFQ scheduler, incorporates the "good" ideas of other schedulers to provide the best possible performance; however, as noted in posts to the Linux mailing list, this may not work well in large RAID systems with Tagged Command Queuing.

To the best of our knowledge, we are the first to present a cooperative anticipatory scheduling algorithm that extends traditional anticipatory scheduling in such a way as to prevent process starvation, mitigate disk throughput problems due to limited anticipation, and present a preliminary methodology to rank schedulers and provide ideas to switch between them at runtime.

9 Conclusions

This paper identified a potential starvation problem and a reduced disk throughput problem in the anticipatory scheduler and proposed a cooperative anticipatory scheduling (CAS) algorithm that mitigates these problems. It also demonstrated that the CAS algorithm can provide significant improvements in application execution times as well as in disk throughput. At its core, the CAS algorithm extends the LAS by broadening anticipation of I/O requests; it gives scheduling priority to requests not only from the process that generated the last request but also to processes that are part of a cooperative process group. We implemented this methodology in Linux.

In addition, the paper evaluated performance for different workloads under the CAS and the four schedulers in Linux 2.6. Microbenchmarks were used to demonstrate the problems with the Linux 2.6 schedulers and the effectiveness of the solution, i.e., the CAS. It was shown that under the CAS web, mail, and file server benchmarks run as much as 62% faster.

Finally, the paper describes our efforts in ranking I/O schedulers based on system behavior as well as workload request characteristics. We hypothesize that these efforts will lead to a methodology that can be used to dynamically select I/O schedulers and, thus improve performance.

10 Acknowledgements

We are grateful to Nick Piggin for taking the time to answer a number of questions related to the implementation of the LAS and sharing some of his thoughts. Also, we thank Steven Pratt of IBM Linux Technology Center (LTC)

at IBM-Austin, TX for valuable discussions, Santhosh Rao of IBM LTC for his help with the FFSB benchmark, and Jayaraman Suresh Babu, UTEP, for his help with related experiments. This work is supported by the Department of Energy under Grant No. DE-FG02-04ER25622, an IBM SUR (Shared University Research) grant, and the University of Texas-El Paso.

11 Legal Statement

References

[1] Axboe, J., "Linux Block IO—Present and Future," *Proceedings of the Ottawa Linux Symposium 2004*, Ottawa, Ontario, Canada, July 21–24, 2004, pp. 51–62

[2] Axobe, J., "Linux: Modular I/O Schedulers," `http://kerneltrap.org/node/3851`

[3] Chatelle, J., "High Read Latency Test (Anticipatory I/O Scheduler)," `http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-02/5329.html`

[4] Corbet, J., "Anticipatory I/O Scheduling," `http://lwn.net/Articles/21274`

[5] Godard, S., "SYSSTAT Utilities Home Page," `http://perso.wanadoo.fr/sebastien.godard/`

[6] Heger, D., Jacobs, J., McCloskey, B., and Stultz, J., "Evaluating Systems Performance in the Context of Performance Paths," *IBM Technical White Paper*, IBM-Austin, TX, 2000

[7] Iyer, S., and Druschel, P., "Anticipatory Scheduling: a Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," *18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, Banff, Alberta, Canada, October 2001, pp. 117–130

[8] Love, R., *Linux Kernel Development*, Sams Publishing, 2004

[9] Love, R., "Kernel Korner: I/O Schedulers," *Linux Journal*, February 2004, 2004(118): p. 10

[10] Madhyastha, T., and Reed, D., "Exploiting Global Input/Output Access Pattern Classification," *Proceedings of SC '97*, San Jose, CA, November 1997, pp. 1–18

[11] Martin, A., "Linux: Where the Anticipatory Scheduler Shines," `http://www.kerneltrap.org/node.php?id=592`

[12] Moilanen, J., "Genetic Algorithms in the Kernel," `http://kerneltrap.org/node/4751`

[13] Pratt, S., and Heger, D., "Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers," *Proceedings of the Ottawa Linux*

Symposium 2004, Ottawa, Ontario, Canada, July 21–24, pp. 425–448

[14] Private communications with Nick Piggin

[15] Teory, T., and Pinkerton, T., "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM*, 1972, 15(3): pp. 177–184

Chip Multi Processing aware Linux Kernel Scheduler

Suresh Siddha
suresh.b.siddha@intel.com

Venkatesh Pallipadi
venkatesh.pallipadi@intel.com

Asit Mallick
asit.k.mallick@intel.com

Abstract

Recent advances in semiconductor manufacturing and engineering technologies have led to the inclusion of more than one CPU core in a single physical processor package. This, popularly know as Chip Multi Processing (CMP), allows multiple instruction streams to execute at the same time. CMP is in addition to today's Simultaneous Multi Threading (SMT) capabilities, like Intel® Hyper-Threading Technology which allows a processor to present itself as two logical processors, resulting in best use of execution resources. With CMP, today's Linux Kernel will deliver instantaneous performance improvement. In this paper, we will explore ideas for further improving peak performance and power savings by making the Linux Kernel Scheduler CMP aware.

1 Introduction

To meet the growing requirements of processor performance, processor architects are looking at new technologies and features focusing on enhanced performance at a lower power dissipation. One such technology is Simultaneous Multi-Threading (SMT). Hyper-Threading (HT) Technology[5] introduced in 2002, is Intel's implementation of SMT. HT delivers two logical processors running on the same execution core, sharing all the resources like functional execution units and cache hierarchy. This approach interleaves the execution of two instruction streams, making the most effective use of processor resources. It maximizes the performance vs. transistor count and power consumption.

Recent advances in semiconductor manufacturing and engineering technologies are leading to rapid increase in transistor count on a die. For example, forthcoming Itanium® family processor code named Montecito will have more than 1.7 billion transistors on a die! As the next logical step to SMT, these extra transistors are put to effective use by including more than one execution core with in a single physical processor package. This is popularly known as Chip Multi Processing (CMP). Depending on the number of execution cores in a package, it's either called a dual-core[4] (two execution cores) or multi-core (more than two execution cores) capable processors. In multi-threading and multi-tasking environment, CMP allows for significant improvement in performance at the system level.

In this paper, in Section 2 we will look at an overview of CMP and some implementation examples. Section 3 will talk about the generic OS scheduler optimization opportunities that are appropriate in CMP environment.

Linux Kernel Scheduler implementation details of these optimizations will be dwelled in Section 4. We will close the paper with a brief look at CMP trends in future generation processors.

2 Chip Multi Processing

In a Chip Multi Processing capable physical processor package, more than one execution core reside in a physical package. Each core has its own resources (architectural state, registers, execution units, up-to a certain level of cache, etc.). Shared resources between the cores in a physical package vary depending on the implementation. Some of the implementation examples are

a) each core could have a portion of on-die cache (for example L1) exclusively for itself and then have a portion of on-die cache (for example L2 and above) that is shared between the cores. An example of this is the upcoming first mobile dual-core processor from Intel, code named Yonah.

b) each core having its own on-die cache hierarchy and its own communication path to the Front Side Bus (FSB). An example of this is the Intel® Pentium® D processor.

Figure 1 shows a simplified block diagram of a physical package which is CMP capable, where two execution cores reside in one physical package, sharing the L2 cache and front side bus resources.

A physical package can be both CMP and SMT capable. In that case, each core in the physical package can in turn contain more than one logical thread. For example, a dual-core with HT will enable a single physical package to appear as four logical processors, capable of running four processes or threads simultaneously. Figure 2 shows an example of a CMP with two

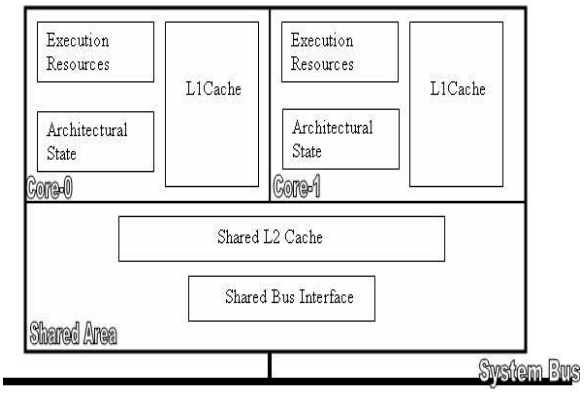

Figure 1: CMP implementation with two cores sharing L2 cache and Bus interface

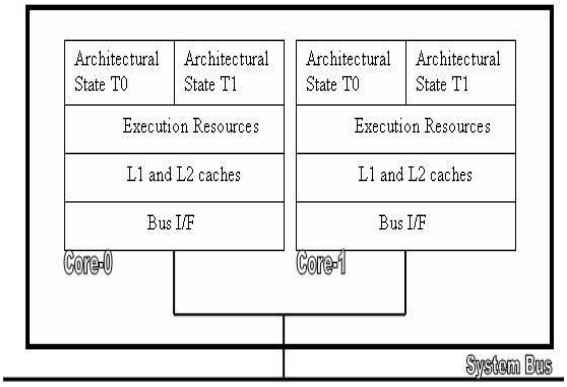

Figure 2: CMP implementation with two cores, each having two logical threads. Each core has their own cache hierarchy and communication path to FSB.

logical threads in each core and with each core having their own cache hierarchy and their own communication path to the FSB. An example of this is the Intel® Pentium® D Extreme Edition processor.

3 CMP Optimization opportunities

A multi-threaded application that scales well and is optimized on SMP systems will have an instantaneous performance benefit from CMP because of these extra logical processors coming from cores and threads. Even if the appli-

cation is not multi-threaded, it can still take advantage of these extra logical processors in a multi-tasking environment.

CMP also brings in new optimization opportunities which will further improve the system performance. One of the optimization opportunity is in the area of Operating System (OS) scheduler. Making the OS scheduler CMP aware will result in an improved peak performance and power savings.

In general, OS scheduler will try to equally distribute the load among all the available processors. In a CMP environment, OS scheduler can be further optimized by looking at micro architectural information(like L2 cache misses, Cycles Per Instruction (CPI), ...) of the running tasks. OS scheduler can decide which tasks can be scheduled on same core/package and which can't be scheduled together based on this micro architectural information. Based on these decisions, scheduler tries to decrease the resource contentions in a CPU core or a package and thereby resulting in increased throughput. In the past, some work[10, 9] has been done in this area and because of the complexities involved (like what micro architectural information need to be tracked for each task and issues in incorporating this processor architecture specific information into generic OS scheduler) this work is not quite ready for the inclusion in today's Operating Systems.

We will not address the micro architectural information based scheduler optimizations in this paper. Instead this paper talks about the OS CMP scheduler optimization opportunities in the case where the system is lightly loaded (i.e., the number of runnable tasks in the system are less compared to the number of available processors in the system). These optimization opportunities are simple and straight forward to leverage in today's Operating Systems and will help in improving peak performance or power savings.

3.1 Opportunities for improving peak performance

In a CMP implementation where there are no shared resources between cores sharing a physical package, cores are very similar to individual CPU packages found in a multi-processor environment. OS scheduler which is optimized for SMT and SMP will be sufficient for delivering peak performance in this case.

However, in most of the CMP implementations, to make best use of the resources cores in a physical package will share some of the resources (like some portion of cache hierarchy, FSB resources, ...). In this case, kernel scheduler should schedule tasks in such a way that it minimizes the resource contention, maximizes the system throughput and acts fair between equal priority tasks.

Let's consider a system with four physical CPU packages. Assume that each CPU package has two cores sharing the last level cache and FSB queue. Let's further assume that there are four runnable tasks, with two tasks scheduled on package 0, one each on package 1, 2 and package 3 being idle. Tasks scheduled on package 0 will contend for last level cache shared between cores, resulting in lower throughput. If all the tasks are FSB intensive (like for example Streams benchmark), because of the shared FSB resources between cores, FSB bandwidth for each of the two tasks in package 0 will be half of what individual tasks get on package 1 and 2. This scheduling decision isn't quite right both from throughput and fairness perspective. The best possible scheduling decision will be to schedule the four available tasks on the four different packages. This will result in each task having independent, full access to last level shared cache in the package and each will get fair share of the FSB bandwidth.

On CMP with shared resources between cores

in a physical package, for peak performance scheduler must distribute the load equally among all the packages. This is similar to SMT scheduler optimizations in todays operating systems.

3.2 Opportunities for improving power savings

Power management is a key feature in today's processors across all market segments. Different power saving mechanisms like P-states and C-States are being employed to save more power. The configuration and control information of these power saving mechanisms are exported through Advanced Configuration and Power Interface (ACPI)[2]. Operating System directed Configuration and Power Management (OSPM) uses these controls to achieve desired balance between performance and power.

ACPI defines the power state of processors and are designated as C0, C1, C2, C3, . . . , Cn. The C0 power state is an active power state where the CPU executes instructions. The C1 through Cn power states are processor sleeping (idle) states where the processor consumes less power and dissipates less heat.

While in the C0 state, ACPI allows the performance of the processor to be altered through performance state (P-state) transitions. Each P-state will be associated with a typical power dissipation value which depends on the operating voltage and frequency of that P-state. Using this, a CPU can consume different amounts of power while providing varying performance at C0 (running) state. At a given P-state, CPU can transit to numerically higher numbered C-states in idle conditions. In general, numerically higher the P states (i.e., lower the CPU voltage) and C-states, the lesser will be power consumed, heat dissipated.

3.2.1 CMP implications on P and C-states

P-states

In a CMP configuration, typically all cores in one physical package will share the same voltage plane. Because of this, a CPU package will transition to a higher P-state, only when all cores in the package can make this transition. P-state coordination between cores can be either implemented by hardware or software. With this mechanism, P-state transition requests from cores in a package will be coordinated, causing the package to transition to target state only when the transition is guaranteed to not lead to incorrect or non-optimal performance state. If one core is busy running a task, this coordination will ensure that other idle cores in that package can't enter lower power P-states, resulting in the complete package at the highest power P-state for optimal performance. In general, this coordination will ensure that a processor package frequency will be the numerically lowest P-state (highest voltage and frequency) among all the logical processors in the processor package.

C-states

In a CMP configuration with shared resources between the cores, processor package can be broken up into different blocks, one block for each execution core and one common block representing the shared resources between all the cores (as shown in Figure 1). Depending on the implementation, each core block can independently enter some/all of the C-state's. The common block will always reside in the numerically lowest (highest power) C-state of all the cores. For example, if one core is in C1 and other core is in C0, shared block will reside in C0.

3.2.2 Scheduling policy for power savings

Let's consider a system having two physical packages, with each package having two cores sharing the last level cache and FSB resources. If there are two runnable tasks, as observed in the Section 3.1 peak performance will be achieved when these two tasks are scheduled on different packages. But, because of the P-state coordination, we are restricting idle cores in both the packages to run at higher power P-state. Similarly the shared block in both the packages will reside in higher power C0 state (because of one busy core) and depending on the implementation, idle cores in both the packages may not be able to enter the available lowest power C-state. This will result in non-optimal power savings.

Instead, if the scheduler picks the same package for both the tasks, other package with all cores being idle, will transition slowly into the lowest power P and C-state, resulting in more power savings. But as the cores share last level cache, scheduling both the tasks to the same package, will not lead to optimal behavior from performance perspective. Performance impact will depend on the behavior of the tasks and shared resources between the cores. In this particular example, if the tasks are not memory/cache intensive, performance impact will be very minimal. In general, more power can be saved with relatively smaller impact on performance by scheduling them on the same package.

On CMP with no shared resources between the cores in a physical package, scheduler should distribute the load among the cores in a package first, before looking for an idle package. As a result, more power will be saved with no impact on performance.

4 Linux Kernel Scheduler enhancements

Process scheduler in 2.6 Linux Kernel is based on hierarchical scheduler domains constructed dynamically depending on the processor topology in the system. Each domain contains a list of CPU groups having a common property. Load balancer runs at each domain level and scheduling decisions happen between the CPU groups at any given domain.

All the references to "Current Linux Kernel" in the coming sections, stands for version 2.6.12-rc5[6]. Current Linux Kernel domain scheduler is aware of three different domains representing SMT (called `cpu_domain`), SMP (called `phys_domain`) and NUMA (called `node_domain`). Current Linux kernel has core detection capabilities for x86, x86_64, ia64 architectures. This will place all CPU cores in a node into different sched groups in SMP scheduler domain, even though they reside in different physical packages. The first step naturally is to add a new scheduler domain representing CMP (called `core_domain`). This will help the kernel scheduler identify the cores sharing a given physical package. This will enable the implementation of scheduling policies highlighted in Section 3.

Figure 3 shows the scheduler domain hierarchy setup with current Linux Kernel on a system having two physical packages. Each package has two cores and each core having two logical threads. Figure 4 shows the scheduler domain hierarchy setup with the new CMP scheduler domain.

4.1 Scheduler enhancements for improving peak performance

As noted in Section 3.1, when the CPU cores in a physical package share resources, peak per-

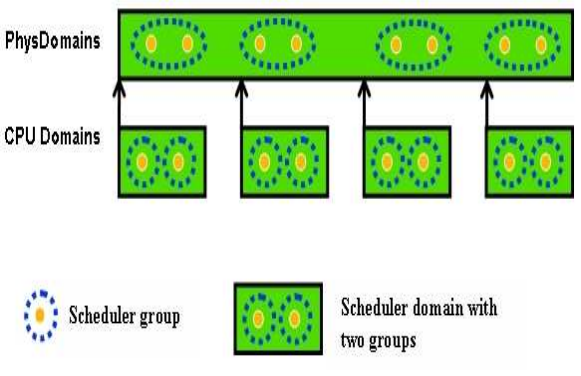

Figure 3: Scheduler domain hierarchy with current Linux Kernel on a system having two physical packages, each having two cores and each core having two logical threads.

formance will be achieved when the load is distributed uniformly among all physical packages. Following subsections will look into the enhancements required for implementing this policy.

4.1.1 Active load balance in presence of CMP and SMT

With SMT and SMP domains in current Linux Kernel, load balance at SMP domain will help in detecting a situation where all the SMT siblings in one physical package are completely idle and more than one SMT sibling is busy in another physical package. Load balance on processors in idle package will detect this situation and will kick active load balance on one of the non idle SMT siblings in the busiest package. Active load balance then looks for a package with all the SMT threads being idle and pushes the task (which was just running before active load balance got kicked in) to one of the siblings of the selected idle package, resulting in optimal performance.

Similarly in the presence of new scheduler domain for CMP, load balance in SMP domain will help detect a situation where more than one core in a package is busy, with another package being completely idle. Similar to the above, active load balance will get kicked on one of the non-idle cores in the busiest package. In the presence of SMT and CMP, active load balance needs to pick up an idle package if one is available; otherwise it needs to pick up an idle core. This will result in load being uniformly distributed among all the packages in a SMP domain and all the cores with in a package.

In pre 2.6.12 -mm kernels, there is a change in active load balance code which leverage the domain scheduler topology more effectively. Instead of looking for an idle package, active load balance code is modified in such a way, that it simply moves the load to the processor which detects the imbalance. In some of the cases[1] this will take few extra hops in finding a correct processor destination for a process but because of simplicity reasons this was pursued. This modification to active load balance also works in the presence of both SMT and CMP.

Figures 4 and 5 show how active balance plays a role in distributing the load equally among the physical packages and CPU cores in presence of CMP and SMT. Figure 6 shows how the new active balance will help in distributing the load equally among the physical packages, even though there is no idle package available. This will help from the fairness perspective.

4.1.2 cpu_power selection

One of the key parameters of a scheduler domain is the scheduler group's `cpu_power`. It represents effective CPU horsepower of the scheduler group and it depends on the underneath domain characteristics. With SMP and SMT domains in current Linux Kernel, `cpu_power` of sched groups in the SMP domain is

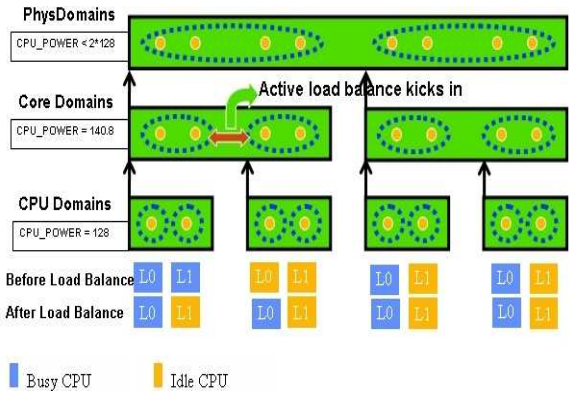

Figure 4: Demonstration of active load balance with 4 tasks, on a system having two physical packages, each having two cores and each core having two logical threads. Active load balance kicks in at the core domain for the first package, distributing the load equally among the cores

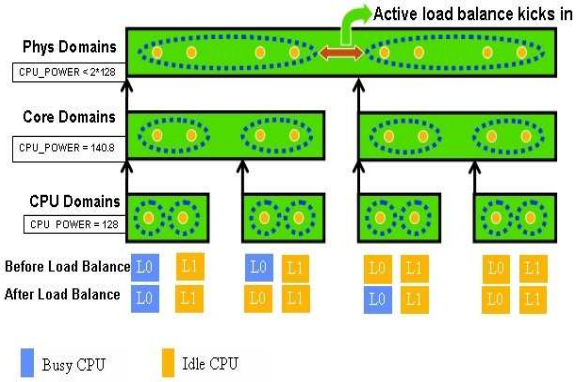

Figure 5: Demonstration of active load balance with 2 tasks, on a system having two physical packages, each having two cores and each core having two logical threads. Active load balance kicks in at SMP domain between the two physical packages, distributing the load equally among the physical packages

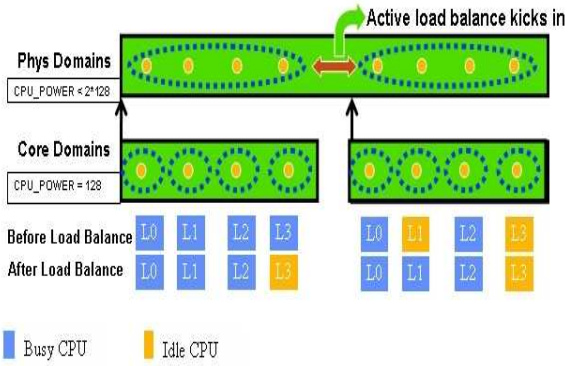

Figure 6: Demonstration of active load balance with 6 tasks, on a system having two physical packages, each having four cores. Active load balance kicks in at SMP domain between the two physical packages, distributing the load equally among the physical packages

calculated with the assumption that each extra logical processor in the physical package will contribute 10% to the `cpu_power` of the physical package.

With the new CMP domain, `cpu_power` for CMP domains scheduler group will be same as `cpu_power` of schedule group in current Linux Kernel's SMP domain (as the underneath SMT domain will remain same). Because of the new CMP domain underneath, new `cpu_power` for SMP domains sched group needs to be selected.

If the cores in a physical package don't share resources, then the `cpu_power` of groups in SMP domain, will simply be the horsepower sum of all the cores in that physical package. On the other hand, if the cores in a physical package share resources, then the `cpu_power` of groups in SMP domain has to be smaller than the no resource sharing case. We will discuss more about this in the power saving Sections 4.2.1 and 4.2.2 and determine how much smaller this needs to be for the peak performance mode policy.

4.1.3 exec, fork balance

Pre 2.6.12 mm kernels has exec, fork balance[3] introduced by Nick Piggin. Setting `SD_BALANCE_{EXEC, FORK}` flags to domains SMP and above, will enable exec, fork balance. Because of this, whenever a new process gets created, it will start on the idlest package and idlest core with in that package. This will remove the dependency on the active load balance to select the correct physical package, CPU core for a new task. This makes the process of picking the right processor more optimal as it happens at the time of task creation, instead of happening after a task starts running on a wrong CPU.

exec, fork balance will select the optimal CPU at the beginning itself and if dynamics change later during the process run, active load balance will kick in and distribute the load equally among the physical packages and the CPU cores with in them.

4.2 Scheduler enhancements for improving power savings

As observed in Section 3.2, when the system is lightly loaded, optimal power savings can be achieved when all the cores in a physical package are completely loaded before distributing the load to another idle package.

When the cores in a physical package share resources, this scheduling policy will slightly impact the peak performance. Performance impact will depend on the application behavior, shared resources between cores and the number of cores in a physical package. When the cores don't share resources, this scheduling policy will result in an improved power savings with no impact on peak performance.

For the CMP implementations which don't share resources between cores, we can make this power savings policy as default. For the other CMP implementations, we can allow the administrator to choose a scheduling policy offering either peak performance (covered in Section 4.1) or improved power savings. Depending on the requirements one can select either of these policies.

Following subsections highlights the changes required in kernel scheduler for implementing improved power savings policy on CMP.

4.2.1 cpu_power selection

The first step in implementing this power savings policy is to allow the system under light load conditions to go into the state with one physical package having more than one core busy and with another physical package being completely idle. Using scheduler group's `cpu_power` in SMP domain and with modifications to load balance, we can achieve this.

In the presence of CMP domain, we will set `cpu_power` of scheduling group in SMP domain to the sum of all the cores horsepower in that physical package. And if the load balance is modified such that, the maximum load in a physical package can grow up to the `cpu_power` of that scheduling group, then the system can enter a state, where one physical package has all its cores busy and another physical package in the system being completely idle.

We will leave the `cpu_power` for the CMP domain as before (same as the one used for SMP domain in the current Linux Kernel) and this will result in active load balance when it sees a situation where more than one SMT thread in a core is busy, with another core being completely idle. As the performance contribution by SMT is not as large as CMP, this behavior will be retained in power saving mode as well.

4.2.2 Active load balance

Next step in implementing this power savings policy is to detect the situation where there are multiple packages being busy, each having lot of idle cores and move the complete load into minimal number of packages for optimal power savings (this minimal number depends on the number of tasks running and number of cores in each physical package).

Let's take an example where there are two packages in the system, each having two cores. There can be a situation where there are two runnable tasks in the system and each end up running on a core in two different packages, with one core in each package being idle. This situation needs to be detected and the complete load needs to be moved into one physical package, for more power savings.

For detecting this situation, scheduler will calculate watt wastage for each scheduling group in SMP domain. Watt wastage represents number of idle cores in a non-idle physical package. This is an indirect indication of wasted power by idle cores in each physical package so that non-idle cores in that package run unaffected. Watt wastage will be zero when all the cores in a package are completely idle or completely busy. Scheduler can try to minimize watt wastage at SMP domain, by moving the running tasks between the groups. During the load balance at SMP domain level, if the normal load balance doesn't detect any imbalance, idle core (in a package which is not wasting much power compared to others in SMP domain) can run this power saving scheduling policy and see if it can pull a task (using active load balance) from a package which is wasting lot of power.

In the last example, idle core in package 0 can detect this situation and can pickup the load from busiest core in package 1. To prevent the idle core in package 1 doing the same thing to the busiest core in package 0 (causing unnecessary ping-pong) load balance algorithm needs to follow the ordering. Figure 7 shows a demonstration of this active load balance, which will result in improved power savings.

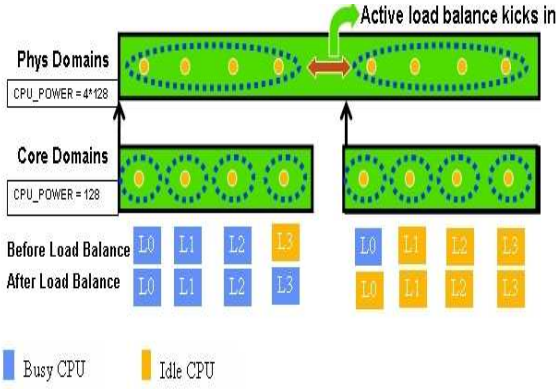

Figure 7: Demonstration of active load balance for improved power savings with 4 tasks, on a system having two physical packages, each having four cores. Active load balance kicks in between the two physical packages, resulting in movement of the complete load to one physical package, resulting in improved power savings

As the number of cores residing in a physical package increase, shared resources between the cores will become bottle neck. As the contention for the resources increase, power saving scheduling policy will result in an increased impact on peak performance. As shown in Figure 7, moving the complete load to one physical package will indeed consume lesser power compared to keeping both the packages busy. But if the cores residing in a package share last level cache, the impact of sharing the last level cache by 4 tasks may outweigh the power saving. To limit such performance impact, we can let the administrator choose the allowed watt wastage for each package. Allowed watt wastage is an indirect indication of the scheduling group's horsepower. `cpu_power` of the scheduling group in SMP domain can be mod-

ified proportionately based on the allowed watt wastage. Load balance modifications in Section 4.2.1 will limit the maximum load that a package can pickup (under light load conditions) and hence the impact to peak performance. More power will be saved with smaller allowed watt wastage. In the case shown in Figure 7, administrator for example can say, under light load conditions don't overload one physical package with more than 2 tasks.

Setting the scheduler groups `cpu_power` of SMP domain to the sum of all the cores horsepower (i.e., allowed watt wastage is zero) will result in a package picking up the max load depending on the number of cores. This will result in maximum power saving. Setting the `cpu_power` to a value less than the combined horsepower of two cores (i.e., allowed watt wastage is one less than the number of cores in a physical package) will distribute the load equally among the physical packages. This will result in peak performance. Any value for `cpu_power` in between will limit the impact to peak performance and hence the power savings.

Administrator can select the peak performance or the power savings policy by setting appropriate value to the scheduler group's `cpu_power` in SMP domain.

4.2.3 exec, fork balance

`SD_BALANCE_{EXEC, FORK}` flags need to be reset for domains SMP and above, causing the new process to be started in the same physical package. Normal load balance will kick in when the load of a package is more than the package's horsepower (`cpu_power`) and there is an imbalance with respect to another physical package.

5 Summary & Future work

CMP related scheduler enhancements discussed in this paper fits naturally to the 2.6 Linux Kernel Domain Scheduler environment. Depending on the requirements, administrator can select the peak performance or power saving scheduler policy. We have prototyped peak performance policy discussed in this paper. We are currently experimenting with the power saving policy, so that it behaves as expected under the presence of CMP, SMT and under the light, heavy load conditions. Once we complete the performance tuning and analysis with real world workloads, these patches will hit the Linux Kernel Mailing List.

For the future generation CMP implementations, researchers and scientists are experimenting[8] with "many tens of cores, potentially even hundreds of cores per package and these cores supporting tens, hundreds, maybe even thousands of simultaneous execution threads." Probably we can extend Moore's law[7] to CMP and can dare say that number of cores per die will double approximately every two years. This sounds plausible for the coming decade at least. With more CPU cores per physical package, kernel scheduler optimizations addressed in this paper will become critical. In future, more experiments and work need to be focused on bringing micro architectural information based scheduling to the mainline.

Acknowledgments

Many thanks to the colleague's at Intel Open Source Technology Center for their continuous support.

Thanks to Nick Piggin and Ingo Molnar for always providing quick comments on our scheduler patches.

References

[1] Active load balance modification in pre 2.6.12 -mm kernels. http://www.ussg.iu.edu/hypermail/linux/kernel/0503.1/0057.html.

[2] Advanced configuration and power interface spec 3.0. http://www.acpi.info/DOWNLOADS/ACPIspec30.pdf.

[3] Balance on exec and fork in pre 2.6.12 -mm kernels. http://www.ussg.iu.edu/hypermail/linux/kernel/0502.3/0037.html.

[4] Intel dual-core processors. http://www.intel.com/technology/computing/dual-core.

[5] Intel hyper-threading technology. http://www.intel.com/technology/hyperthread.

[6] Linux kernel. http://www.kernel.org.

[7] Moore's law. http://www.intel.com/research/silicon/mooreslaw.htm.

[8] Processor and platform evolution for the next decade. http://www.intel.com/technology/techresearch/idf/platform-2015-keynote.htm.

[9] Daniel Nussbaum Alexandra Fedorova, Christopher Small and Margo Seltzer. *Chip Multithreading Systems Need a New Operating System Scheduler*. SIGOPS, ACM, 2004.

[10] Jun Nakajima and Venkatesh Pallipadi. *Enhancements for Hyper-Threading Technology in the operating System: Seeking the Optimal Scheduling*. WIESS, USENIX, December 2002.

SeqHoundRWeb.py: interface to a comprehensive online bioinformatics resource

Peter St. Onge
University of Toronto
pete@{seul.org|economics.utoronto.ca}

Paul Osman
paul@eval.ca

Abstract

In the post-genomic era, getting useful answers to challenging biological questions often demands significant expertise and resources not only to acquire the requisite biological data but also to manage it. The storage required to maintain a workable genomic or proteomic database is usually out of reach for most biologists. Some toolsets already exist to facilitate some aspects of data analysis, and others for access to particular data stores (e.g., NCBI Toolkit), but there is a substantial learning curve to these tools and installation is often non-trival. SeqHoundRWeb.py grew out of a common frustration in bioinformatics—the initiate bioinformaticist often has substantial biological knowledge, but little experience in computing; Python is often held up as a good first scripting language to learn, and in our experience new users can be productive fairly rapidly.

Introduction

The discovery of DNA by Watson and Crick marked the beginnings of massive upheaval in biology, and ultimately, in the ways biologists work. Research today, in the so-called post-genomic era, has embraced computing technology as never before, with repercussions affecting all areas of biology[15].

One of the greatest hurdles a novice bioinformaticist must face is the learning curve when learning an approach to biology that does not involve the use of any of the classical or well-known laboratory techniques.

Perl is probably the most commonly used bioinformatics language currently[13], and has substantial and rich object-oriented facilities to deal with biological data[2]. While Perl is highly versatile and effective in the hands of experienced bioinformaticists[14], my experience shows that initiates to programming through academic bioinformatics courses often have considerable difficulty understanding the breadth of Perl approaches and idioms sufficiently for it to be useful.

Like Perl, other languages have seen specialized facilities to handle biological information develop considerably. Java[1], Ruby[4], Python[3], amongst others, all have have been used successfully in research projects. In particular, use of Python is becoming increasingly commonplace in research[12].

A second issue is the ability to obtain and effectively exploit data in order to test the research hypotheses of interest. Fortunately, there are many research sites provid-

ing substantial data for research use, including the US National Center for Biotechnology Information[7] (NCBI), the Gene Ontology Consortium[5] (GO) and the European Bioinformatics Institute[8] (EBI). Each of these sites host a considerable amount of data, in both size and breadth, typically database table dumps allowing others to reconstitute and further develop the data for individual research needs. Although the data files are normally well under a gigabyte in size (compressed), typical processing requires some skill even for simple parsing and data work up, particularly because of the file sizes involved.

Not surprisingly, most research questions tend to be more complex and require more robust approaches to managing data, such as relational databases (e.g. MySQL and PostGreSQL) which in turn often mean having substantial hardware and some system administration skills.

Other types of data processing, such as genome-level comparisons between species through Basic Local Alignment Search Tool[10] (BLAST) can be highly CPU intensive for hours or even days depending on the size of the genomes being searched and the underlying hardware. As this data updates on an ongoing basis, the need to rebuild result sets dictates the availability of large quantities of substantial computing power.

One project to assemble much of the data from these various sources and carry out many of the more demanding data process steps was SeqHound[11], merging biological sequence (genomic and proteomic), taxonomy, annotation and 3-D structure within an object-oriented database management system and exposed via a web-based API. Although most of SeqHound is F/OSS, hosting it locally locally would be difficult for most researchers without substantial hardware and technical expertise, as the current (as of Oct '04) recommendation is to have a system able to store some 650 GB of data[9].

In order to make access to SeqHound simpler for novice bioinformaticists, my approach was to trade off some speed for flexibility, and build a Python wrapper around SeqHound's HTTP API, exposing much of the rich data provided by SeqHound to the ease-of-use of Python and its language features.

Overview

The primary prerequisites for SeqHoundRWeb.py are the urllib and os modules, so this means that SeqHoundRWeb.py should be able to work on any platform supported by Python. Installation of the package will be via the typical Python installation techniques, and we expect that it will be made available through the normal distribution channels soon. Until then, the code can be imported into Python, as shown below as long as the location of the SeqHoundRWeb.py file is provided—novice Python users can take full advantage of the functions provided in this module without the need for root or Administrator-level access!

Table 1 shows a straightforward example of retrieval of a GenInfo (GI) identifier given a particular accession number for a hypothetical protein for a species of rat (specifically, the Norway Rat, *Rattus norvegicus*).

Acknowledgements

This project came about thanks to a number of people: Alexander Ignachenko, Shaun Ghanny, Robin Haw, Henry Ling, Bianca Tong, Kayu Chin, Thomas Kislinger, and Ata Ghavidel all provided constructive criticism and support, for

```
import os
import SeqHoundRWeb

# Set the proper URL for seqhound
os.environ["SEQHOUNDSITE"] = "http://seqhound.blueprint.org"

accs = [
   "CAA28783",
   "CAA28784",
   "CAA28786"
]

for acc in accs:
   result = SeqHoundRWeb.SeqHoundFindAcc([acc])
   if result[0] == 'SEQHOUND_OK': # found it
       print result[1]
```

Table 1: Simple SeqHoundRWeb Example - Rat

which I remain grateful. Katerina Michalickova, Michel Dumontier and others from the Hogue Lab at Mount Sinai Hospital for creating SeqHound and exposing its functionality via HTTP. An initial proof of concept, which became the basis for this project, was built in the Emili Lab at the University of Toronto, and I would like to thank the Department of Economics for allowing me the opportunity to continue working on it as part of my professional responsibilities. Thanks also to Alex Brotman, Ales Hvezda and others from the SEUL Project for their keen eyes in finding mistakes in the text. Any errors or omissions are, of course, my own[6].

References

[1] BioJava web site. http://www.biojava.org/.

[2] BioPerl web site. http://bio.perl.org/.

[3] BioPython web site. http://www.biopython.org/.

[4] BioRuby web site. http://www.bioruby.org/.

[5] Gene Ontology Consortium FTP site. http://archive.godatabase.org/latest-full/.

[6] It's All Pete's Fault website. http://www.itsallpetesfault.org/.

[7] NCBI FTP site. http://www.ncbi.nlm.nih.gov/Ftp/.

[8] Catherine Brooksbank, Evelyn Camon, Midori A. Harris, Michele Magrane, Maria Jesus Martin, Nicola Mulder, Claire O'Donovan, Helen Parkinson, Mary Ann Tuli, Rolf Apweiler, Ewan Birney, Alvis Brazma, Kim Henrick, Rodrigo Lopez, Guenter Stoesser, Peter Stoehr, and Graham Cameron. The European Bioinformatics Institute's data resources. *Nucl. Acids Res.*, 31(1):43–50, 2003.

[9] Ian Donaldson, Katerina Michalickova, Hao Lieu, Renan Cavero, Michel Dumontier, Doron Betel, Ruth Isserlin, Marc Dumontier, Michael Matan, Rong Yao, Zhe Wang, Victor Gu, and Elizabeth Burgess. The SeqHound Manual, Release 3.01, October 2004.

[10] Mark Yandell Ian Korf and Joseph Bedell. *BLAST*. O'Reilly, 2003.

[11] Katerina Michalickova, Gary Bader, Michel Dumontier, Hao Lieu, Doron Betel, Ruth Isserlin, and Christopher Hogue. Seqhound: biological sequence and structure database as a platform for bioinformatics research. *BMC Bioinformatics*, 3(1):32, 2002.

[12] J. Daniel Navarro, Vidya Niranjan, Suraj Peri, Chandra Kiran Jonnalagadda, and Akhilesh Pandey. From biological databases to platforms for biomedical discovery. *Trends in Biotechnology*, 21(6):263–268, 2003.

[13] James Tisdall. *Beginning Perl for Bioinformatics*. O'Reilly, 2001.

[14] James D. Tisdall. *Mastering Perl for Bioinformatics*. O'Reilly, 2003.

[15] Johnathan D. Wren. Engineering in genomics. *IEEE Engineering in Medicine and Biology Magazine*, pages 87–98, March/April 2004.

Ho Hum, Yet Another Memory Allocator...

Do We Need Another Dynamic Per-CPU Allocator?

Ravikiran G Thirumalai
kiran.th@gmail.com
Dipankar Sarma
Linux Technology Center, IBM India Software Lab
dipankar@in.ibm.com
Manfred Spraul
manfred@colorfullife.com

Abstract

The Linux® kernel currently incorporates a minimalistic slab-based dynamic per-CPU memory allocator. While the current allocator exists with some applications in the form of block layer statistics and network layer statistics, the current implementation has issues. Apart from the fact that it is not even guaranteed to be correct on all architectures, the current implementation is slow, fragments, and does not do true node local allocation. A new per-CPU allocator has to be fast, work well with its static sibling, minimize fragmentation, co-exist with some arch-specific tricks for per-CPU variables and get initialized early enough during boot up for some users like the slab subsystem. In this paper, we describe a new per-CPU allocator that addresses all issues mentioned above, along with possible uses of this allocator in cache friendly reference counters (bigrefs), slab head arrays, and performance benefits due to these applications.

1 Introduction

The Linux kernel has a number of allocators, including the page allocator for allocating physical pages, the slab allocator for allocating objects with caching, and vmalloc allocator. Each of these allocators provide ways to manage kernel memory in different ways. With the introduction of symmetric multi-processing (SMP) support in the Linux kernel, managing data that are rarely shared among processors became important. While statically allocated per-CPU data has been around for a while, support for dynamic allocation of per-CPU data was added during the development of 2.6 kernel. Dynamic allocation allowed per-CPU data to be used within dynamically allocated data structures making it more flexible for users.

The dynamic per-CPU allocator in the 2.6 kernel was, however only the first step toward better management of per-CPU data. It was a compromise given that the use of dynamically allocated per-CPU data was limited. But with the need for per-CPU data increasing and support for NUMA becoming important, we decided to revisit the issue.

In this paper, we present a new dynamic per-CPU data allocator that saves memory by interleaving objects of the same processor, supports allocating objects from memory close to the CPUs (for NUMA platforms), works during early boot and is independent of the slab allocator. We also show it allows implementation of more complex synchronization primitives like distributed reference counters. We discuss some preliminary results and future course of action.

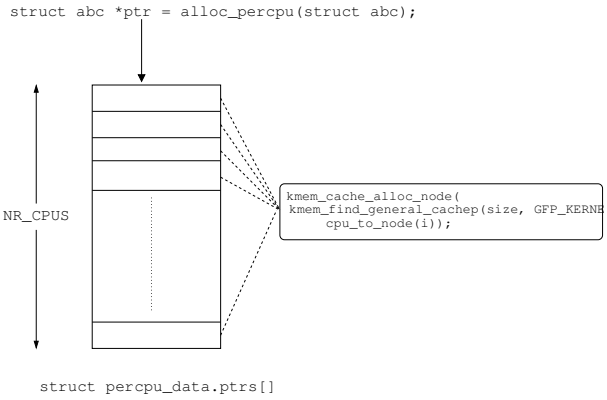

Figure 1: Current allocator

2 Background

Over the years, CPU speed has been increasing at a much faster rate then speed of memory access. This is even more important in multi-processor systems where accessing memory shared between the processors could be significantly more costly if the corresponding cache line is not available in that processor's cache.

Operation	Cost (ns)
Instruction	0.7
Clock Cycle	1.4
L2 Cache Hit	12.9
Main Memory	162.4

Table 1: 700 MHz P-III Operation Costs

Table 1 shows the cost of memory operations on a 700 MHz Pentium™ III processor. When global data is shared between processors, cache lines bouncing between processors reduce memory bandwidth and thereby negatively impact scalability. As scalability improved during the development of the 2.6 kernel, the need for efficient management of infrequently shared data also increased. The first step towards this was interleaved static per-CPU areas proposed by Rusty Russell [3]. This allowed better management of cache lines by sharing them between CPU-local versions of different objects. As the need for per-CPU data increased beyond what could be declared statically, the first RFC for a dynamic per-CPU data allocator was proposed [6] along with a reference implementation [7].

Subsequent discussions led to a simplified implementation of a dynamic allocator in the 2.5 kernel as shown in Figure 1. This allocator provides an interface `alloc_percpu()` that returns a pointer cookie. The pointer cookie is the address of an array of pointers to CPU-local objects each corresponding to a CPU in the system. The array and the CPU-local objects are allocated from the slab. Simplicity was the most important factor with this allocator, but it clearly had a number of problems.

1. The slab allocations are no longer padded to cache line boundaries. This means that the current implementation would lead to false sharing.
2. An additional memory access (array of CPU-local object pointers) has a performance penalty, mostly due to associativity miss.
3. The array itself is not NUMA-friendly.

4. It wastes space.

We therefore implemented a new dynamic allocator that worked around the problems of the current one. This allocator is based on the reference implementation [7] published earlier. The key improvement has been the use of pointer arithmetics to determine the address of the CPU-local objects, which reduces dereferencing overhead.

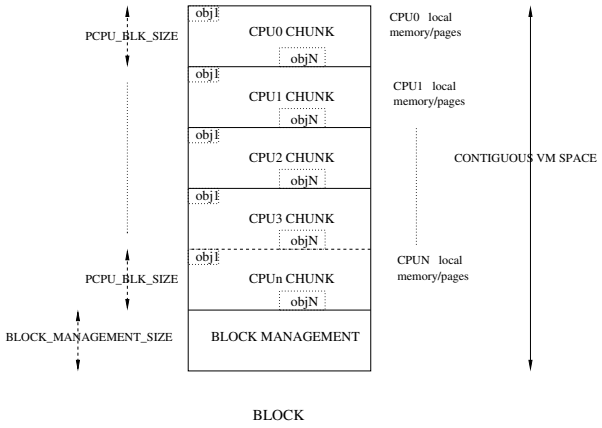

Figure 2: A block

3 Interleaved Dynamic Per-CPU Allocator

3.1 Design Goals

In order to address the inadequacies in the current per-CPU allocation schemes in the Linux kernel, a new allocator must do the following:

1. Fast pointer dereferencing to get to the per-CPU object
2. Allocate node local memory for all CPUs
3. Save on memory, minimize fragmentation, maximize cache line utilization
4. Work well with CPU hotplug and memory hotplug, sparse CPU numbers.
5. Get initialized early during boot
6. Independent of the slab allocator
7. Work well with its static sibling (static per-CPU areas)

A typical memory allocator returns a record (usually a pointer) that can be used to access the allocated object. A per-CPU allocator needs to return a record that can be used to access every copy of the object's private data to the corresponding CPU. In our allocator, this record is a cookie and the CPU-local versions of the allocated objects can be accessed using it. Dereferencing speeds are very important, since this is the fast path for all users of per-CPU data. The CPU-local versions of the object also need to be allocated from the memory nearest to the CPU on NUMA systems. Also, in order to avoid the overhead of an extra memory access in the current per-CPU data implementation, we needed to use pointer arithmetics to access the object corresponding to a given CPU. The pointer arithmetic should be simple and should use as few CPU cycles as possible.

3.2 Allocating a Block

The internal allocation unit of the interleaved per-CPU allocator is a `block`. Requests for per-CPU objects are served from a `block` of memory. The `block`s are allocated on demand for new per-CPU objects. A `block` is a contiguous virtual memory space (VA space) that is reserved to contain a chunk of objects corresponding to every CPU. It also contains additional space that is used to maintain internal structures for managing the blocks.

Figure 2 shows the layout of a `block`. The VA space within a `block` consists of two sections:

1. The top section consists of `NR_CPU` chunks of VA space each of size `PCPU_BLK_SIZE`. `PCPU_BLKSIZE` is a compile time constant. It represents the capacity of one `block`. Each CPU has one such per-CPU chunk within a `block`. Currently the size of each per-CPU chunk is two pages. `PCPU_BLKSIZE` is the size limit of a per-CPU object.

2. The bottom section of a `block` consists of memory used to maintain the per-CPU object buffer control information for this `block` and plus block descriptor size. This section is of size `BLOCK_MANAGEMENT_SIZE`.

While the VA for the entire `block` is allocated, the actual pages for each per-CPU chunk are allocated only if the corresponding CPU is present in the `cpu_possible_mask`. This has two benefits—it avoids unnecessary waste of memory and each chunk can be allocated so it is closest to the corresponding CPU. `alloc_page_node()` is used to get pages nearest to the CPU. The management pages at the bottom of a `block` are always allocated. Once the pages are allocated, VA space is then mapped with pages for the CPU-local chunks.

Also, as shown in Figure 3, there won't be mappings for any VA space corresponding to CPUs that are "not possible" on the system. The VA space is contiguous for `NR_CPUS` processors and this allows us to use pointer arithmetics to calculate the address of an object corresponding to a given CPU. We also save memory by not allocating for CPUs that are not in the `cpu_possible` mask.

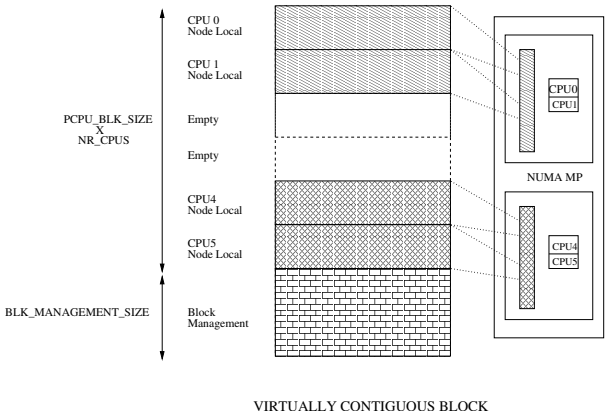

Figure 3: Page allocation for a block

3.3 Allocating Objects from a block

The per-CPU chunks inside a `block` are further divided into units of `currency`. A `currency` is the size of the smallest object that can be allocated in this scheme. The currency size is defined as `sizeof(void *)` in the current implementation. Any object in this allocator consists of one or more contiguous units of `currency`.

Each `block` in the system has a descriptor associated with it. The descriptor is defined as below:

```
struct pcpu_block {
    void *start_addr;
    struct page *pages[PCPUPAGES_PER_BLOCK * 2];
    struct list_head blklist;
    unsigned long bitmap[BITMAP_ARR_SIZE];
    int bufctl_fl[OBJS_PER_BLOCK];
    int bufctl_fl_head;
    unsigned int size_used;
};
```

This is embedded into the block management part of the block. In the current implementation, it is at the beginning of the block management section of a `block`. Each per-CPU object is allocated from one such `block` maintained by the interleaved allocator. The block descriptor records the base effective address of

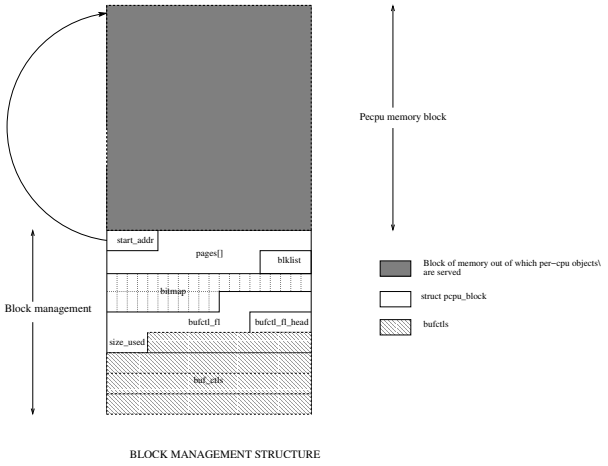

Figure 4: Managing blocks

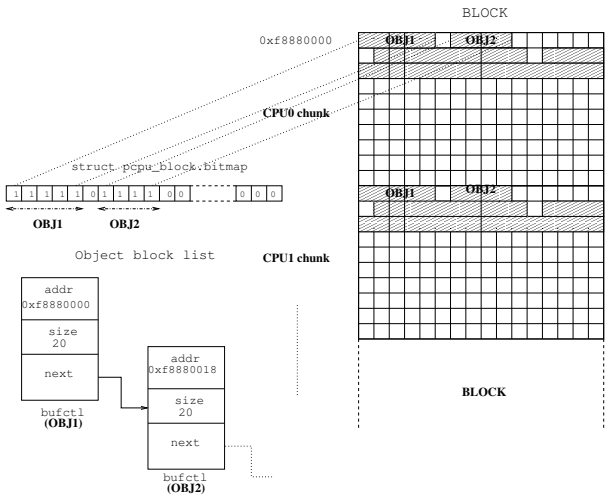

Figure 5: Object layout

the block (`start_addr`) as well as the allocation state of each `currency` within the `block`. The allocation state is recorded using a bitmap wherein each bit represents an isomorphic `currency` of every per-CPU chunk in that block. There are as many bits as the number of `currency` in one chunk of the block.

Figure 4 shows the organization of the block management area. Block descriptor has an array of pointers, each pointing to a CPU-local chunk of physical pages allocated for this block. Each object allocated from a `block` is represented by a `bufctl` data structure. These `bufctl` structures are embedded in the block management section of the `block` and they start right after the block descriptor. The block descriptor also has an array-based free list to allocate `bufctl` or object descriptors. `bufctl_fl` is the array-based free list and `bufctl_head` stores the head of this free list.

During allocation of a per-CPU object, the bitmap indicating `currency` allocation state is sorted and saved. This array is sorted in ascending order of available object sizes in that `block` due to contiguous `currency` regions. This array is traversed and the first element that fits the allocation requirement is used and the corresponding `currency` units are allocated.

Figure 5 shows the relation between a `block` in the allocator, per-CPU objects allocated from the `block`, the bitmap corresponding to these objects, `bufctl` structures and `bufctl` list for these objects. In this example, a per-CPU block starts at 0xf8880000. The currency size is assumed to be 4 (`sizeof (void *)` on x86). The squares in CPU chunks represent the allocator currency. The first five consecutive currencies make OBJ1 (shaded currencies in the block). Each currency is represented by a bit in the bitmap. Hence, bits 0–4 of the bitmap correspond to OBJ1. OBJ1 starts at address 0xf8880000. OBJ2 starts at 0xf8888018. The figure also depicts the bufctl structures and bufctl list for OBJ1 and OBJ2.

3.4 Managing blocks

The amount of per-CPU objects served by a single `block` is limited. So, our allocator allows allocation of new `block` on demand. Whenever a request for a per-CPU object cannot be met with any `block` currently in the system, a new `block` is added to the system that is isomorphic to existing ones.

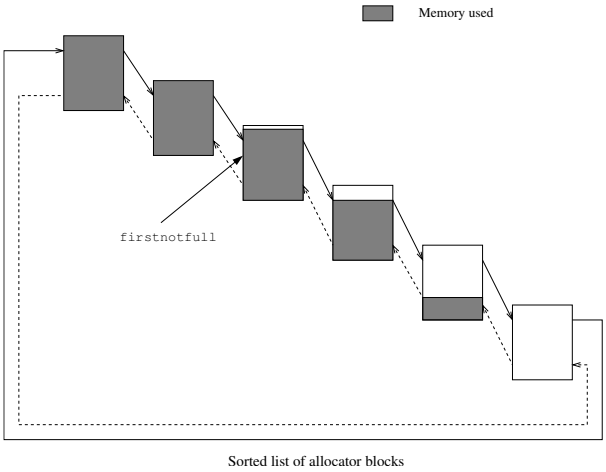

Figure 6: Managing block lists

These blocks are linked to one another in a circular doubly linked sorted list (Figure 6). `pcpu_block` counts the amount of memory used in the block (`size_used`). This list is sorted in descending order using `size_used`. The `firstnotfull` field contains the list position of the first `block` in the list that has available memory for allocation. On an allocation request, the list traversed from the `firstnotfull` position and the first available block with sufficient space is chosen for allocation. If no such block is found, a new block is created and added to this list. The blocks are repositioned in the list to preserve the sorted nature of the list upon every allocation and free request. During the course of freeing per-CPU objects, if the allocator notices that `blkp->size_used` goes to zero, the entire block—VA space, per-CPU pages, block management pages and the VA mapping are destroyed.

3.5 Accessing Per-CPU Data

A per-CPU allocation returns a pointer that is used as a cookie to access CPU-local version of the object for any given CPU. This pointer effectively points to address of the CPU-local object for CPU 0. To get to the CPU-local version of CPU N, the following arithmetics is used—`cpu_local_address = p + N * PCPU_BLKSIZE`, where `p` is the cookie returned by the interleaved allocator. Since `PCPU_BLKSIZE` is a carefully chosen compile time constant of a proper page order, the above arithmetics is optimized to a simple add and bit shift operations. The most expensive operation in accessing the CPU-local object is usually the determination of the current CPU number (`smp_processor_id()`). This is true for static per-CPU areas as well. To avoid the cost of `smp_processor_id()` during per-CPU data access, kernel developers like Rusty Russell have been contemplating using a dedicated processor register to get a handle to that processor's CPU-local data. The current static per-CPU area in the Linux kernel uses an array (`__per_cpu_offset[]`) to store a handle to each CPU's per-CPU data. With a dedicated processor register, `__per_cpu_offset[cpuN]` would be loaded into the register and users of per-CPU data would not need to make a call to `smp_processor_id()` to get to the CPU-local versions—simple arithmetic on the contents of the processor dedicated register will suffice. In fact, `smp_processor_id()` could be derived from the register based `__per_cpu_offset[]` table. This scheme can co-exist with our per-CPU allocator.

4 Using Dynamic Per-CPU Allocator

4.1 Per-CPU structures within the slab allocator

The Linux slab allocator uses arrays of object pointers to speed up object allocation and release. This avoids doing costly linked list or

```
struct kmem_cache_s {
    struct array_cache *array[NR_CPUS];
    /* ... additional members, only
     * touched from slow path ... */
};
struct array_cache {
    unsigned int avail;
    unsigned int limit;
    void * objects[];
};
```

Table 2: Main structures in the fast-path—before

```
struct kmem_cache_s { /* per-CPU variable */
    struct kmem_globalcache *global;
    unsigned int avail;
    unsigned int limit;
    void * objects[];
};
struct kmem_globalcache { /* one instance */
    /* ... additional members, only
     * touched from slow path ... */
};
```

Table 3: Main structures for fast-path—after

spinlock operations in each operation. Each object cache contains one array for each CPU. If an array is not empty, then an allocation little more than looking up the per-CPU array and returning one entry from that array. Therefore the time required for the pointer lookup is the most significant part of the execution time for `kmem_cache_alloc` and `kmem_cache_free`.

At present, the lookup code mimics the implementation of the dynamic per-CPU variables: `kmem_cache_create` returns a pointer to the structure that contains the array of pointers to the per-CPU variables. Each allocation or release looks up the correct per-CPU structure and returns an object from the array. Table 2 shows the (slightly simplified) structures.

While this is a simple implementation, it has several disadvantages:

- It is a code duplication and it would be better if slab could reuse the primitives provided by the dynamic per-CPU variables. This is not possible, because it would create a cyclic dependency: the dynamic per-CPU variable implementation relies on the slab allocator for its own allocations.
- It is a simple per-CPU allocator, therefore each access required a table lookup. Depending on the value of `NR_CPUS`, there might be even frequent write operations, and thus cache line transfers on the cache line that contains the table.
- The implementation is fixed within `slab.c`, it's not possible to override it with arch specific code, even if an architecture supports a fast per-CPU variable lookup.

Therefore the slab code was rearranged to use per-CPU variables natively for the object caches: `kmem_cache_create` returns the pointer to the per-CPU structure that contains the members that are needed in the fast-path of the allocator. The other members are stored in a new structure (`structkmem_globalcache`). The new structure layout is shown in Table 3.

The functions `kmem_cache_alloc()` and `kmem_cache_free()` only need to access `avail`, `limit`, and `objects`, thus there are no accesses to the global structure from the fast-path.

4.2 Statistics counters

As part of the scalable statistics counter work we carried out earlier, it has already been established that per-CPU data is useful for kernel statistics counters, and solves the problem of cache line bouncing on NUMA and multiprocessor systems [5]. During the development

of the 2.6 kernel, a number of kernel statistics were converted to use a dynamic per-CPU allocator. These include networking MIBs, disk statistics and the `percpu_counter` used in ext2 and ext3 filesystems. With our allocator, the per-CPU statistics counters become more efficient. In addition to faster dereferencing and node-local allocation, our allocator saves `NR_CPUS x sizeof(void *)` bytes of memory for each per-CPU counter by avoiding the array storing the CPU-local object pointers.

4.3 Distributed reference counters (bigrefs)

Rusty Russell has an experimental patch that makes use of the dynamic per-CPU memory allocator to avoid global atomic operations on reference counters[4].

A "bigref" reference counter would consist of two counters internally; one of type `atomic_t` which is the global counter, and another per-CPU counter of type `local_t`, the distributed counter. Per-CPU memory for the `local_t` is allocated when the bigref reference counter is initialized. The reference counter usually operates in the "fast" mode—it just increments or decrements the CPU-local `local_t` counter whenever the bigref reference counter needs to be incremented or decremented (`get()` and `put()` operations in Linux parlance). This operation on `local_t` per-CPU counters is much cheaper compared to operations on a global `atomic_t` type. In fact on x86, local increment is just an `incl` instruction.

The reference counter switches to a "slow" mode when the element being protected by the reference counter is no longer needed in the system and is being released or 'disowned'. This switch from fast mode to slow mode is done by using `synchronize_kernel()` to make sure all CPUs recognize slow mode operation before the 'disowning' completes. The reference counter is biased with a high value by setting the `atomic_t` counter with the high bias value before the switch to slow mode is initiated. In fact this biasing itself indicates beginning of the switch. This bias value is subtracted from the reference counter after the switch to slow mode.

Bigrefs save on space and dereference speeds when they use our per-CPU allocator. In addition to space saving, our allocator interlaces counters on cache lines too, which results in increased cache utilization.

5 Results

5.1 Slab enhancements

The new slab implementation discussed in Section 4.1 was tested with both micro-benchmarks and real-world test loads.

- Micro-benchmarks showed no change between the old and the new implementation; In a tight loop, `kmem_cache_alloc` needed around 35 CPU cycles on an 64-bit AMD Athlon™. The lack of improvement is not unexpected because a table lookup is only slow on a cache miss.
- Tests with tbench (version 3.03 with warm-up) on a 4-CPU HT Pentium 4 (2.8GHz Xeon) system showed an improvement of around 1%.

6 Future Work

The new allocator is not without its own limitations:

1. One major drawback of our allocator design is increased TLB footprint. Since our allocator uses the Linux vmalloc VM area to stitch all node local and block management pages into one contiguous block, hot per-CPU data may take too many TLB entries when there are too many allocator blocks within the system. Node-local page allocation and fast dereferencing are of utmost importance, so we have to use a virtually contiguous area for fast pointer arithmetic. But to limit the increased TLB usage, in the future, we may want to use large pages for blocks, and fit all per-CPU data in one block. This way, we can limit the number of TLB entries taken up by the dynamic per-CPU allocator.

2. The allocation operation is slow. It is not designed for allocation speed. It is designed for maximum utilization and minimum fragmentation. Given the current users of dynamic per-CPU allocator, it may not be valuable to improve allocation operation.

With the interleaved dynamic per-CPU allocator, it has also become possible to implement distributed locks [1] [2] and reference counters [4].

7 Conclusion

The interleaved per-CPU allocator we implemented is a step forward from where we are in the Linux kernel. The current allocator in the Linux kernel leads to false sharing and it is not optimized. Our allocator overcomes all of those problems. As the Linux kernel matures, this will allow use of more sophisticated primitives in the Linux kernel without adding any overhead. The design has evolved over a number of discussions and it is mature enough to handle all architectures.

8 Acknowledgments

Rusty Russell started the ball rolling by first implementing the static per-CPU areas and later advising us. We thank him for all of his contributions. We would also like to thank a number of Linux kernel hackers, including Dave Miller and Andrew Morton, all of whom advised us in many different situations. We are indebted to Gerrit Huizenga, Paul McKenney, Jim Wasko Jr., and Vijay Sukthankar for being so supportive of this effort.

9 Legal Statement

References

[1] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions of Computer Systems 9*, 1 (February 1991), 21–65.

[2] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third PPOPP* (Williamsburg, VA, April 1991), pp. 106–113.

[3] RUSSELL, R. Subject: [patch] 2.5.1-pre5: per-cpu areas. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=100759080903883&w=2`, December 2001.

[4] RUSSELL, R. Subject: Re: [patch] mm: Reimplementation of dynamic percpu memory allocator. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=110569951013489&w=2`, January 2005.

[5] SARMA, D. Scalable statistics counters. Available: `http://lse.sourceforge.net/counters/statctr.html`, May 2002.

[6] SARMA, D. Subject: [lse-tech] [rfc] dynamic percpu data allocator. Available: `http://marc.theaimsgroup.com/?l=lse-tech&m=102215919918354&w=2`, May 2002.

[7] SARMA, D. Subject: [rfc][patch] kmalloc_percpu. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=102761936921486&w=2`, July 2002.

Beagle: Free and Open Desktop Search

Jon Trowbridge
Novell
trow@novell.com

Abstract

I will be discussing Beagle, a desktop search system that is currently being developed by Novell. It acts as a search aggregator, providing a simple API for simultaneously querying multiple data sources. Pluggable backends do the actual searching while Beagle handles the details, such as consolidating and ranking the hits and passing them back to client applications. Beagle includes a core set of backends that build full-text indexes of your personal data, allowing you to efficiently search your files, e-mail, contacts, calendar, IM logs, notes and web history. These indexes are updated in real time to ensure that any search results will always reflect the current state of your data.

Check Online

This author did not provide the body of this paper by deadline. Please check online for any updates.

Glen or Glenda

Empowering Users and Applications with Private Namespaces

Eric Van Hensbergen
IBM Research
bergevan@us.ibm.com

Abstract

Private name spaces were first introduced into LINUX during the 2.5 kernel series. Their use has been limited due to name space manipulation being considered a privileged operation. Giving users and applications the ability to create private name spaces as well as the ability to mount and bind resources is the key to unlocking the full potential of this technology. There are serious performance, security and stability issues involved with user-controlled dynamic private name spaces in LINUX. This paper proposes mechanisms and policies for maintaining system integrity while unlocking the power of dynamic name spaces for normal users. It discusses relevant potential applications of this technology including its use with FILESYSTEM IN USERSPACE[24], V9FS[8] (the LINUX port of the PLAN 9 resource sharing protocol) and PLAN 9 FROM USER SPACE[4] (the PLAN 9 application suite including user space synthetic file servers ported to UNIX variants).

1 What's in a name?

Names are used in all aspects of computer science[21]. For example, they are used to reference variables in programing languages, index elements in a database, identify machines in a network, and reference resources within a file system. Within each of these categories names are evaluated in a specific context. Program variables have scope, database indexes are evaluated within tables, networks machine names are valid within a particular domain, and file names provide a mapping to underlying resources within a particular *name space*. This paper is primarily concerned with the evaluation and manipulation of names and name space contexts for file systems under the LINUX operating system.

File systems evolved from flat mappings of names to multi-level mappings where each catalog (or directory) provided a context for name resolution. This design was carried further by MULTICS[1] with deep hierarchies of directories including the concept of links between directories within the hierarchy[5]. Dennis Ritchie, Rudd Canaday and Ken Thompson built the first UNIX file system based on MULTICS, but with an emphasis on simplicity[22]. All these file systems had a single, global name space.

In the late 1980s, Thompson joined with Rob Pike and others in designing the PLAN 9 operating system. Their intent was to explore potential solutions to some of the shortcomings of UNIX in the face of the widespread use of high-speed networks[19]. It was designed from first principles as a seamless distributed system

with integrated secure network resource sharing.

The configuration of an environment to use remote application components or services in place of their local equivalent is achieved with a few simple command line instructions. For the most part, application implementations operate independent of the location of their actual resources. PLAN 9 achieves these goals through a simple well-defined interface to services, a simple protocol for accessing both local and remote resources, and through dynamic, stackable, per-process private name spaces which can be manipulated by any user or application.

On the other hand, the LINUX file system name space has traditionally been a global flat name space much like the original UNIX operating system. In November of 2000, Alexander Viro proposed implementing PLAN 9 style per-process name space bindings[28], and in late February 2001 released a patch[2] against the 2.4 kernel. This code was later adopted into the mainline kernel in 2.5. This support, which is described in more detail in section 4, established an infrastructure for private name spaces but restricted the creation and manipulation of name spaces as privileged.

This paper presents the case for making name space operations available to common users and applications while extending the existing LINUX dynamic name space support to have the power and flexibility of PLAN 9 name spaces. Section 2 describes the design, implementation and advantages of the PLAN 9 distributed system. Example applications of this technology are discussed in Section 3. The existing LINUX support is described in more detail in Section 4. Perceived barriers and solutions to extended LINUX name space support are covered in Section 5. Section 6 overviews related work and recent proposals as alternatives to our approach and Section 7 summarizes our conclusions and recommendations.

2 Background: Plan 9

In PLAN 9, all system resources and interfaces are represented as files. UNIX pioneered the concept of treating devices as files, providing a simple, clear interface to system hardware. In the 8th edition, this methodology was taken further through the introduction of the /proc synthetic file system to manage user processes[10]. Synthetic file systems are comprised of elements with no physical storage, that is to say the files represented are not present as files on any disk. Instead, operations on the file communicate directly with the sub-system or application providing the service. LINUX contains multiple examples of synthetic file systems representing devices (DEVFS), process control (PROCFS), and interfaces to system services and data structures (SYSFS).

PLAN 9 took the file system metaphor further, using file operations as the simple, well-defined interface to all system and application services. The design was based on the knowledge that any programmer knows how to interact with files. Interfaces to all kernel subsystems from the networking stack to the graphics frame buffer are represented within synthetic file systems. User-space applications and services export their own synthetic file systems in much the same way as the kernel interfaces. Common services such as domain name service (DNS), authentication databases, and window management are all provided as file systems. End-user applications such as editors and e-mail systems export file system interfaces as a means for data exchange and control. The benefits and details of this approach are covered in great detail in the existing PLAN 9 papers[18] and will be covered to a lesser extent by application examples in section 3.

9P[15] represents the abstract interface used to access resources under PLAN 9. It is somewhat analogous to the VFS layer in LINUX[11].

In PLAN 9, the same protocol operations are used to access both local and remote resources, making the transition from local resources to cluster resources to grid resources completely transparent from an implementation standpoint. Authentication is built into the protocol and was extended in its INFERNO[20] derivative Styx[14] to include various forms of encryption and digesting.

It is important to understand that all 9P operations can be associated with different active semantics in synthetic file systems. Traversal of a directory hierarchy can allocate resources or set locks. Reading or writing data to a file interface can initiate actions on the server. The dynamic nature of these semantics makes caching dangerous and in-order synchronous execution of file system operations a must.

The 9P protocol itself requires only a reliable, in-order transport mechanism to function. It is commonly used on top of TCP/IP[16], but has also been used over RUDP[13], PPP[23], and over raw reliable mechanisms such as the PCI bus, serial port connections, and shared memory. The IL protocol was designed specifically to provide 9P with a reliable, in order transport on top of an IP stack without the overhead of TCP[17].

The final key design point of PLAN 9 is the organization of all local and remote resources into a dynamic private name space. Manipulating an element's location within a name space can be used to configure which services to use, interpose stackable layers onto service interfaces, and create restricted "sandbox" environments. Under PLAN 9 and INFERNO, the name space of each process is unique and can be manipulated by ordinary users through mount and bind system calls.

Mount operations allow a client to attach new interfaces and resources which can be provided by the operating system, a synthetic file server, or from a remote server. Bind commands allow reorganization of the existing name space, allowing certain services to be bound to well-known locations. Bind operations can substitute one resource for another, for example, binding a remote device over a local one. Binding can also be used to create stackable layers by interposing one interface over another. Such interposer interfaces are particularly useful for debugging and statistics gathering.

The default mount and bind behavior is to replace the mount-point. However, PLAN 9 also allows multiple directories to be stacked at a single point in the name space, creating a *union* directory. Within such a directory, each component is searched to resolve name lookups. Flags to the mount and bind operations determine the position of a particular component in the stack. A special flag determines whether or not file creation is allowed within a particular component.

By default, processes inherit an initial name space from their parent, but changes made to the child's name space are not reflected in the parent's. This allows each process to have a context-specific name space. The PLAN 9 fork system call may be called with several flags allowing for the creation of processes with shared name spaces, blank name spaces, and restricted name spaces where no new file systems can be mounted. PLAN 9 also provides library functions (and associated system calls) for creating a new name space without creating a process and for constructing a name space based on a file describing mount sources, destinations, and options.

3 Applications

There are many areas where the pervasive use of private dynamic name spaces under PLAN 9

can be applied in similar ways under LINUX. Many of these are detailed in the foundational PLAN 9 papers [19, 18] as well as the PLAN 9 manual pages[15]. We will step through a subset of these applications and provide some additional potential applications in the LINUX environment.

Under PLAN 9, one of the more straightforward uses of dynamic name space is to bind resources into well-known locations. For example, instead of using a PATH environment variable, various executables are bound into a single `/bin` union directory. PLAN 9 clusters use a single file server providing resources for multiple architectures. Typical startup profiles bind the right set of binaries to the `/bin` directory. For example, if you logged in on an x86 host, the binaries from `/386/bin` would be bound to `/bin`, while on PPC `/power/bin` would be bound over bin. Then the user's private binary directory is bound on top of the system binaries. This has a side benefit of searching the various directories in a single lookup system call versus individually walking to elements in the path list from the shell.

Another use of stackable binds in PLAN 9 is within a development environment. You can bind directories (or even individual files) with your changes over read-only versions of a larger hierarchy. You can even recursively bind a blank hierarchy over the read-only hierarchy to deposit compiled object files and executables. The PLAN 9 development environment at Bell Labs has a single source tree which people bind their working directories and private object/executable trees over. Once they are satisfied with their changes, they can push them from the local versions to the core directories. Using similar techniques developers can also keep distinct groups of changes separated without having to maintain copies of the entire tree.

Crafting custom name spaces is also a good way to provide tighter security controls for services. Daemons exporting network services can be locked into a very restrictive name space, thus helping to protect system integrity even if the daemon itself becomes compromised. Similarly, users accessing data from other domains over mounted file systems don't run as much risk of other users gaining access if they mount the resources in a private name space. Users can craft custom sandboxes for untrusted applications to help protect against potential malicious software and applets.

As mentioned earlier, PLAN 9 combines dynamic name space with a remote resource sharing protocol to enable transparent distributed resource utilization. Remote resources are bound into the local name space as appropriate and applications run completely oblivious to what resources they are actually using. A straightforward example of this is mounting a networked stereo component's audio device from across the room instead of using your workstation's sound card before starting an audio jukebox application. A more practical example is mounting the external network protocol stack of a firewall device before running a web browser client. Since the external network protocol stack is only mounted for the particular browser client session, other services running in separate sessions with separate name spaces (and protocol stacks) are safe from external access. The PLAN 9 paradigm of mounting any distributed resource and transparently replacing local resources (or providing a network resource when a local resource isn't available) provides an interesting model for implementing grid and utility based computing.

Another example of this is the PLAN 9 cpu(1) command which is used to connect from a terminal to a cluster compute node. Note that this command doesn't operate like ssh or telnet. The cpu(1) command will export to the server the current name space of the process from which it was executed on the client. Server side

scripts take care of binding the correct architectural binaries for the cpu server over those of the client terminal. Interactive I/O between the cpu and the client is actually performed by the cpu server mounting the client's keyboard, mouse, and display devices into the session's private name space and binding those resources over its own. Custom profiles can be used to limit the resources exported to the cpu server, or to add resources such as local audio devices or protocol stacks. It represents a more elegant approach to the problems of grid, cluster, and utility-based computing providing a mechanism for the seamless integration and organization of local resources with those spread across the network.

Similar approaches can be provided to virtualization and para-virtualization environments. At the moment, the LINUX kernel is plagued by a plethora of "virtual" device drivers supporting various devices for various virtualized environments. Separate gateway devices are supported for Xen[7], VMware[30], IBM Hypervisors[3], User Moder Linux[9], and others. Additionally, each of these virtualization engines requires separate gateways for each class of device. The PLAN 9 paradigm provides a unified, simple, and secure method for supporting these various virtual architectures and their device, file system, and communication needs. Dynamic private name spaces enable a natural environment for sub-dividing and organizing resources for partitioned environments. Application file servers or generic plug-in kernel modules provide a variety of services including copy-on-write file systems, copy-on-write devices, multiplexed network connections, and command and control structures. IBM Research is currently investigating using V9FS together with private name spaces and application synthetic file servers to provide just such an approach for partitioned scale-out clusters executing high-performance computing applications.

4 Linux Name Spaces

The private name space support added in the 2.5 kernel revolved around the addition of a `CLONE_NEWNS` flag to the LINUX clone(2) system call. The clone(2) system call allows the creation of new threads which share a certain amount of context with the parent process. The flags to clone specify the degree of sharing which is desired and include the ability to share file descriptor tables, signal handlers, memory space, and file system name space. The current default behavior is for processes and threads to start with a shared copy of the global name space.

When the `CLONE_NEWNS` flag is specified, the child thread is started with a copy of the name space hierarchy. Within this thread context, modifications to either the parent or child's name space are not reflected in the other. In other words, when a new name space is requested during thread creation, file servers mounted by the child process will not be visible in the parent's name space. The converse is also true. In this way, a thread's name space operations can be isolated from the rest of the system. The use of the `CLONE_NEWNS` flag is protected by the `CAP_SYS_ADMIN` capability, making its use available only to privileged users such as root.

LINUX name spaces are currently manipulated by two system calls: `mount(2)` and `umount(2)`. The `mount(2)` system call attaches a file system to a mount-point within the current name space and the `umount(2)` system call detaches it. More recently in the 2.4 kernel series, the `MS_BIND` flag was added to allow an existing file or directory subtree to be visible at other mount-points in the current name space. Both system calls are only valid for users with `CAP_SYS_ADMIN` capability, and so are predominately used only by root. The table of mount points in a thread's

current name space can be viewed by looking at the `/proc/xxx/mounts` file.

Users may be granted the ability to mount and unmount file systems through the mount(1) application and certain flags in the fstab(5) configuration file. This support requires that the mount application be configured with set-uid privileges and that the exact mount source and destination be specified in the fstab(5). Certain network file systems (such as SMBFS, CIFS, and V9FS) which have a more user-centric paradigm circumvent this by having their own set-uid mount utilities: smbmnt(8), cifs.mount(8), and 9fs(1). More recently, there has been increased interest in user-space file servers such as FILESYSTEM IN USERSPACE (FUSE)[24] with its own set-uid mount application fusermount(8).

The proliferation of these various set-uid applications that circumvent the kernel protection mechanisms indicates the need to re-evaluate the existing restrictions so that a more practical set of policies can be put in place within the kernel. Users should be able to mount file systems when and where appropriate. Private name spaces seem to be a natural fit for preventing global name space pollution with individual user mount and bind activities. They also provide a certain degree of isolation from user mounted synthetic file systems, providing an additional degree of protection to system demons and other users who might otherwise unwittingly access a malicious user-level file server.

Private name space support in LINUX is under utilized primarily due to the classification of name space operations as privileged. It is further crippled by the lack of stackable name space semantics and application file servers. Unlocking applications and environments such as those described in Section 3 by removing some of the restrictions enforced by the LINUX kernel would create a much more elegant and powerful computing environment. Additionally, providing a more flexible, yet consistent set of kernel enforced policies would be far superior to the wide range of semantics currently enforced by file system specific set-uid mount applications.

5 Barriers and Solutions

PLAN 9 is not LINUX, and LINUX is not PLAN 9. There are significant security model and file system paradigm differences between the two systems. Concerns related to these differences have been broken down into four major categories: concerns with user name space manipulation, problems with users being able to mount arbitrary file systems, potential problems with user file systems, and problems with allowing users to create their own private name spaces.

5.1 Binding Concerns

The `mount(1)` command specified with the `-bind` option, hereafter referred to as a bind operation, is an incredibly useful tool even in a shared global name space. When combined with the notion of private name spaces, it allows users and applications to craft custom environments in which to work. However, the ability to dynamically bind directories and/or files over one another creates several security concerns that revolve around the ability to transparently replace system configuration and common data with potentially compromised versions.

PLAN 9 places no restrictions on bind operations. Users are free to bind over any system directory or file regardless of access permissions—binding writable layers over otherwise read-only directories can be one of the more useful operations. However, PLAN 9's

authentication and system configuration mechanisms are constructed in such a way as to not rely on accessing files when running under user contexts. In other words, authentication and configuration are system services which are started at boot (or reside on different servers), and so aren't affected by user manipulations of their private name spaces.

Under LINUX, system services are constructed differently and there is still heavy reliance on well-known files which are accessed throughout user sessions. Examples include such sensitive files as `/etc/passwd` and `/etc/fstab`.

Similar concerns apply to certain system directories which multiple users may have write access to, such as `/tmp` or `/usr/tmp`. If users are able to bind over these public directories under the global name space, they could potentially compromise the data of another user who inadvertently used a bound `/tmp` instead of the system `/tmp`.

These problems can be addressed with a simple policy of only allowing a user to bind over a directory they have explicit write access to. This solves the problem of system configuration files, but doesn't cover globally writable spaces such as `/usr/tmp`. A simple solution to protecting such shared spaces is to only allow user initiated binds within private name spaces. A slightly more complicated form of protection is based on the assumption that such public spaces have the *sticky bit* set in the directory permissions.

When used within directory permissions, the sticky bit specifies that files or subdirectories can only be renamed or deleted by their original owner, the owner of the directory, or a privileged process. This prevents users from deleting or otherwise interfering with each other's files in shared public spaces. A simple policy to extend this protection to user name space manipulation is to return a permissions error when a normal user attempts to bind over a directory in which the sticky bit is set.

While limiting binds to sticky-bit directories is reasonable enough, it is an unnecessary restriction. The use of private name spaces solves several security concerns with user-modifications to name space, and does so without overly limiting the user's ability to mount over these shared spaces. Another benefit of requiring user binds to be within a private name space is that it prevents such binds from polluting the global system name space.

5.2 Mounting Concerns

Another set of concerns has to do with allowing users to mount new file systems into a name space. As discussed previously, this is something currently accomplished through set-uid mount applications which check the user's permissions versus particular policies. A more global policy would give administrators more consistent control over users and help eliminate the potential problems caused by the use of set-uid applications

One of the primary problems with giving users the ability to mount arbitrary file systems is the concern that they may mount a file system with set-uid scripts allowing them to gain access to privileged accounts (i.e., root). It is relatively trivial for a user to construct a file system image, floppy, or CD-ROM on a personal machine with set-uid shells. If they were allowed to mount these on an otherwise secure system, they could instantly compromise it. The existing mount applications circumvent such a vulnerability by providing a NOSUID flag which disables interpretation of set-uid and set-gid permission flags. A similar

mechanism enforced as the default for all user-mounts would provide a certain level of protection against such an attack.

Another possible attack vector would be the image being mounted. Most file systems are written on the assumption that the backing store is somewhat secure and reputable. LINUX kernel community members have expressed concern that disk images could be constructed specifically to crash or corrupt certain file systems, so as to disable or disrupt system activity. This is particularly difficult to protect against, but not all file systems are vulnerable to such attacks. In particular, network file systems are written defensively to prevent such corruption from affecting the rest of the system. Such defensively written file systems could be marked with an additional file system flag marking them as safe for users to mount.

Each mounted file system uses a certain amount of system resources. Unlocking the ability to mount a new file system also unlocks the ability for the user to abuse the system resources by mounting new file systems until all system memory is expended. This sort of activity is easily controlled with per-user mount limits maintained using the kernel resource limit system with a policy set by the system administrator.

A slightly different form of resource abuse mentioned earlier is name space pollution. If users are granted the ability to mount and bind a large number of file systems, the resulting name space pollution could prove to be distracting, if not damaging to performance. Enforcing a policy in which users are only able to mount new file systems within a private name space easily contains such pollution to the user's session. Additionally, the current name space garbage collection will take care of conveniently unmounting file servers and recovering resources when the session associated with the private name space closes.

5.3 User File System Concerns

A driving motivation behind providing users the ability to mount and bind file systems is the increase in popularity of user-space file servers. These predominantly synthetic file systems are enabled through a number of different packages including V9FS and more predominantly FUSE. These packages export VFS interfaces or equivalent APIs to user space, allowing applications to act as file servers. Practical uses for such file servers include the exporting of archive file contents as mountable name spaces, adding cryptographic layers, and mapping of network transports such as ftp to synthetic file hierarchies.

Since they are implemented as user applications, these synthetic file servers pose an even greater danger to system integrity by allowing users to implement arbitrary semantics for operations. These implementations can easily provide corrupt data to system calls or block system call resolution indefinitely, bringing the entire system to a grinding halt. Because of this, application file servers have fallen under harsh criticism from the LINUX kernel community. However their many practical uses makes the engineering of a safe and reliable mechanism allowing their use in a LINUX environment highly desirable.

Many of the prior solutions mentioned can be used to limit the damage done by a malicious user-space file servers. Private name spaces can protect system daemons and other users from stumbling into a synthetic file system trap. Restrictions preventing set-uid and set-gid interpretation within user mounts can prevent malicious users from using application file servers to gain access to privileged accounts or information.

A different sort of permissions problem is also introduced by application file servers. Typi-

cally, the file servers are started by a certain user and information within the file system is accessed under that user's authority, potentially in a different authentication domain. For example, if a user mounts a ftpfs or an sshfs by logging into a remote server domain, they are potentially exposing the data from that domain to other users and administrators on the local domain. As this is undesirable, it is important that other users (besides the initiator) do not obtain direct access to mounted file systems. While there are several ways of approaching this (including overloaded permissions checks that deny access to anyone but the mounter), private name spaces seem to handle this nicely without changing other system semantics.

5.4 Private Name Space Concerns

While they are limited, several barriers do exist to user creation and use of private name spaces. One objection to allowing users to create their own private name spaces is the existence of a vulnerability in the `chroot(1)` infrastructure in the presence of such private name spaces. The `chroot(1)` command is used to establish a new root for a particular user's name space. However, if a private name space is created with the `CLONE_NS` flag, the new thread is allowed to traverse out of the chroot "jail" simply using the dot-dot traversal. This appears to be more of a bug than a feature and should be easy to defend against by never allowing a user to traverse out of the root of their current name space.

The same resource concerns that apply to user mounts also apply to private name spaces. However, since the user can have no more private name spaces than processes, there is a preexisting constraint. Additionally, due to the copy semantics present in the existing LINUX name space infrastructure, the user will be charged for every mount he inherits when creating a private name space. If these two limitations are deemed insufficient, an additional per-user limit can be established for private name spaces.

More prevalent among these perceived problems is the change in basic paradigm. No longer can the same file system environment be expected from every session on a particular system. In fact, depending on the extent to which private name spaces are used there may even be different file system views in different windows on the same session. The plurality of name spaces across processes and sessions provides a great deal of flexibility in construction of private environments, but is quite a departure from expected behavior.

The ability to maintain a certain degree of traditional semantics is desirable during a transition in paradigms. Further, having to mount core resources for each session is rather tedious and undesirable. To a certain extent this can be mitigated by more advanced inheritance techniques within the private name spaces—allowing changes in parents to be propagated to children but not vice versa. This is further discussed in the Related Work section regarding Alexander Viro's shared subtrees proposal.

Another possibility is a per-session name space created when a user logs into the system. This provides a single name space for that session separate from the global name space insulating user modifications from the unsuspecting. However, in simpler embodiments it doesn't provide the per-user name space semantics some desire (ie. the name space wouldn't actually bridge two different SSH sessions). One possibility here is to tightly bind creation and adoption of the per-user name space to the login process (potentially as part of the PAM infrastructure). Another possibility would be to use the name space description present in the

`/proc/xxxx/mounts` synthetic file to create a duplicate name space in different process groups. This would work well for network file systems, binds, and V9FS but may not work well for certain user file servers such as FUSE.

V9FS enables multi-session user file servers without problems as it separates mount-point from the file system semantics. In other words, when you run a V9FS application file server, it creates a mount point which could be used by several different clients to mount the resulting file system. Besides giving the ability to share the resulting file system between user sessions, this technique potentially allows other users to access the mount-point. User credentials are part of the V9FS mount protocol, so each user is authenticated on the file system based on their own credentials instead of the credentials of the user who initially started the file server application.

6 Related Work

There are several historical as well as ongoing attempts to provide more dynamic name space operations in LINUX and/or open up those operations to end-users and not just privileged administrators. There are also several outstanding request-for-comments on extensions to the existing name space support.

The original V9FS project had tried to integrate private name space support into the file system and remote-resource sharing [12]. While this worked in practice, Alexander Viro's release of private name space support within the LINUX kernel suspended work on the V9FS private name space implementation.

As a follow-up to Viro's initial name space support, he released a shared sub-tree request-for-comments[29] detailing specific policies for propagating name space changes from parent to children. This provides a more convenient form of inheritance allowing name space changes in parents to also take effect in children with private name spaces.

Miklos Szeredi, the project leader of FUSE has proposed several patches related to opening up and expanding name space support. Among these were an altered permission semantics[25] to prevent users other than the mounting user from accessing FUSE mounts. After this met from some resistance from the LINUX kernel community, Miklos proposed an invisible mount patch[26] which tries to protect other users from potentially malicious mounts by hiding them from other users without the use of private name spaces. A separate patch[27] attempted to unlock mount privileges by enforcing a static policy on user-mounts including some of the protections we have described previously (only writable directories can be mounted over, only safe file systems can be mounted, and set-uid/set-gid permissions are disabled). To date, none of these patches have been incorporated into the mainline, but most of these events are happening concurrently with the writing and revision of this paper.

One of the responses to the FUSE patches was the assertion that the job may have been better done in user-space by an extended form of the mount(1) application. The advantage to using a user-space policy solution is a much wider and dynamic set of policies than would be desirable to incorporate directly into the kernel. Such an application would have set-uid style permissions, which several in the community have criticized as undesirable. An alternative to this approach would be to use up-calls from the kernel to a user-space policy daemon.

Another outcome of the FUSE discussion was a patch[6] providing an unshare system call which could be used to create private name spaces in a pre-existing thread. In other words,

this would allow a thread to request a private name space without having been spawned with one, making the creation of private name spaces more accessible. The unshare patch also provides similar facilities for controlling other resources originally only available via flags during the clone system call.

The file system translator project (FiST)[33] takes a different approach, offering users the ability to add incremental features to existing file systems. It provides a set of templates and a toolkit which allow for relatively easy creation of kernel file system modules which sit atop pre-existing conventional file systems. The resulting modules have to be installed and mounted by a privileged user. Instead of relying on set-uid helper applications, FiST allows use of "private" instances of the file system through a special ioctl attach command and per-user sub-hierarchies. Several example file system layers are provided with the standard FiST distribution including cryptographic layers and access control list enforcement layers.

One of the more interesting FiST file system layers is UNIONFS[32][31]. It provides a fan-out file system which goes beyond the relatively simple semantics of PLAN 9's union directories by providing additional flexibility and granular control of specific components. There is also support for rudimentary sandboxing without the use of private name spaces.

Among the additional features of UNIONFS is recursive unification allowing deep binds of directories. In PLAN 9 and the existing LINUX name space implementations, a bind only affects a single file or directory. The recursive unification feature of UNIONFS allows entire hierarchies to be bound. This is particularly useful in the context of copy-on-write file system semantics. While such functionality can be provided with scripts under PLAN 9 and LINUX, the UNIONFS approach would seem to provide a more efficient and scalable solution.

7 Conclusions

Opening up name space operations to common users will enable better working environments and transparent cluster computing. Users should be granted the permission to establish private name spaces through flags provided to the clone(2) system call or using the newly proposed unshare system call. Once isolated in a private name space, normal users should be granted the ability to mount new resources and organize existing resources in ways they see fit. A simple set of system-wide restrictions on these activities will prevent malicious users from obtaining privileged access, disrupting system operation, or compromising protected data. Adding stackable file name spaces into the kernel file system interfaces would further extend these benefits.

8 Acknowledgements

Between the time this paper was proposed and published, much debate has occurred on the LINUX kernel mailing list and the LINUX file systems developers mailing list. I've incorporated a great deal of that discussion and commentary into this document and many of the ideas represented here come from that community.

I'd like to thank Alexander Viro for laying the ground work by adding the initial private name space support to LINUX. I'd also like to thank Miklos Szeredi and the FUSE team for pushing the ideas of unprivileged mounts and user application file servers.

Support for this paper was provided in part by the Defense Advance Research Projects Agency under Contract No. NBCH30390004.

References

[1] F. J. Corbato and V.A. Vyssotsky. Introduction and overview of the multics system. *Joint Computer Conference*, 1965.

[2] Jonathan Corbet. Kernel development. *LWN.NET Weekly News*, 0301, March 2001.

[3] IBM Corp. Virtualization engine. `http://www.ibm.com/`.

[4] Russ Cox. Plan 9 from user space. `http://swtch.com/plan9port`.

[5] R.C. Daley and P.G. Neumann. A general-purpose file system for secondary storage. *Fall Joint Computer Conference*, 1965.

[6] Janak Desai. new system call, unshare. EMAIL, May 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=111573064706562&w=2`.

[7] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[8] E. Van Hensbergen and R. Minnich. Grave robbers from outer space: Using 9p200 under linux. In *Proceedings of Freenix Annual Conference*, pages 83–94, 2005.

[9] Hans jorg H Oxer, Hans jorg Hoxer, Kerstin Buchacker, and Volkmar Sieh. Implementing a user mode linux with minimal changes from original kernel. *unknown*, 2002.

[10] T.J. Killian. Processes as files. In *USENIX Summer Conf. Proceedings*, Salt Lake City, UT, June 1984.

[11] Robert Love. *Linux Kernel Development*. Sam's Publishing, 800 E. 96th Street, Indianapolis, Indiana 46240, 2nd edition, August 2003.

[12] Ron Minnich. V9fs: A private name space system for unix and its uses for distributed and cluster computing. In *Conference Francaise sur les Systemes*, June 1999.

[13] C. Partridge and r. Hinden. Reliable data protocol. Internet RFC/STD/FYI/BCP Archives, April 1990.

[14] R. Pike and D. M. Ritchie. The styx architecture for distributed systems. *Bell Labs Technical journal*, 4(2):146–152, April-June 1999.

[15] Rob Pike et al. *Plan 9 Programmer's Manual - Manual Pages*. Vita Nuova Holdings Limited, 3rd edition, 2000.

[16] J. Postel. Transmission control protocol darpa internet program protocol specification. Internet RFC/STD/FYI/BCP Archives, September 1981.

[17] D. Presotto and P. Winterbottom. *The IL Protocol*, volume 2, pages 277–282. AT&T ell Laboratories, Murray Hill, NJ, 1995.

[18] D. Presotto R. Pike et al. The use of name spaces in plan 9. *Operating Systems Review*, 27(2):72–76, April 1993.

[19] K. Thompson R. Pike et al. Plan 9 from bell labs. *Computing Systems*, Vol 8(3):221–254, Summer 1995.

[20] R. Pike S. Dorward et al. The inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, Winter 1997.

[21] J. H. Saltzer. *Lecture Notes in computer Science, 60, Operating systems - An Advanced Course*, chapter 3.A.: Naming and Binding of Objects, pages 99–208. Springer-Verlag, 1978.

[22] Peter H. Salus. *The Daemon, the GNU, and the Penguin*, chapter 2 & 3: UNIX. Groklaw, 2005.

[23] W. Simpson. The point-to-point protocol (ppp) for the transmission of multi-protocol datagrams over point-to-point links. Internet RFC/STD/FYI/BCP Archives, May 1992.

[24] Miklos Szeredi. Filesystem in userspace. `http://fuse.sourceforge.net`.

[25] Miklos Szeredi. Fuse permission modell. EMAIL, April 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=111323066112311&w=2`.

[26] Miklos Szeredi. Private mounts. EMAIL, April 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=111437333932219&w=2`.

[27] Miklos Szeredi. Unpriviledged mount/umount. EMAIL, May 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=111513156417879&w=2`.

[28] Alexander Viro. Re: File system enhancement handled above the file system level. Email, November 2000.

[29] Alexander Viro. Shared subtrees. EMAIL, January 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=110565591630267&w=2`.

[30] VMware. Vmware home page. `http://www.vmware.com`.

[31] C. P. Wright et al. Versatility and unix semantics in a fan-out unification file system. Technical Report FSL-04-01B, Computer Science Department, Stony Brook University, October 2004. `http://www.fsl.cs.sunysb.edu/docs/unionfs-tr/unionfs.pdf`.

[32] C.P. Wright and E. Zadok. Unionfs: Bringing file systems together. *Linux Journal*, December 2004.

[33] E. Zadok. Writing stackable file systems. *Linux Journal*, pages 22–25, May 2003.

LINUX® Virtualization on Virtual Iron™ VFe

Alex Vasilevsky, David Lively, Steve Ofsthun
Virtual Iron Software, Inc.
{alex,dflively,sofsthun}@virtualiron.com

Abstract

After years of research, the goal of seamlessly migrating applications from shared memory multi-processors to a cluster-based computing environment continues to be a challenge. The main barrier to adoption of cluster-based computing has been the need to make applications cluster-aware. In trying to solve this problem two approaches have emerged. One consists of the use of middleware tools such as MPI, Globus and others. These are used to rework applications to run on a cluster. Another approach is to form a pseudo single system image environment by clustering multiple operating system kernels[Pfister-98]. Examples of these are Locus, Tandem NonStop Kernel, OpenSSI, and openMosix.

However, both approaches fall far short of their mark. Middleware level clustering tools require applications to be reworked to run a cluster. Due to this, only a handful of highly specialized applications sometimes referred to as `embarrassingly parallel`—have been made cluster-aware. Of the very few commercial cluster-aware applications, the best known is Oracle® Database Real Application Clustering. OS kernel clustering approaches present other difficulties. These arise from the sheer complexity of supporting a consistent, single system image to be seen on every system call made by every program running on the system: applications, tools, etc; to making existing applications that use SystemV shared-memory constructs to run transparently, without any modifications, on this pseudo single system.

In 2003, Virtual Iron Software began to investigate the potential of applying virtual machine monitors (VMM) to overcome difficulties in programming and using tightly-coupled clusters of servers. The VMM, pioneered by IBM in the 1960s, is a software-abstraction layer that partitions hardware into one or more virtual machines[Goldberg-74], and shares the underlying physical resource among multiple applications and operating systems.

The result of our efforts is Virtual Iron VFe, a purpose-built clustered virtual machine monitor technology, which makes it possible to transparently run any application, without modification, on a tightly-coupled cluster of computers. The Virtual Iron VFe software elegantly abstracts the underlying cluster of computers with a set of Clustered Virtual Machine Monitors (CVMM). Like other virtual machine monitors, the CVMM layer takes complete control of the underlying hardware and creates virtual machines, which behave like independent physical machines running their own operating systems in isolation. In contrast to other virtual machine monitors, the VFe software transparently creates a shared memory multi-processor out of a collection of tightly-coupled servers.

Within this system, each operating system has

the illusion of running on a single multi-processor machine with `N` CPUs on top of `M` physical servers interconnected by high throughput, low latency networks.

Using a cluster of VMMs as the abstraction layer greatly simplifies the utilization and programmability of distributed resources. We found that the VFe software can run any application without modification. Moreover, the software supports demanding workloads that require dynamic scaling, accomplishing this in a manner that is completely transparent to OSs and their applications.

In this paper we'll describe Linux virtualization on Virtual Iron VFe, the virtualization capabilities of the Virtual Iron Clustered VMM technology, as well as the changes made to the LINUX kernel to take advantage of this new virtualization technology.

1 Introduction

The CVMM creates virtual shared memory multi-processor servers (Virtual Servers) from networks of tightly-coupled independent physical servers (Nodes). Each of these virtual servers presents an architecture (the Virtual Iron Machine Architecture, or ViMA) that shares the user mode instruction set with the underlying hardware architecture, but replaces various kernel mode mechanisms with calls to the CVMM, necessitating a port of the guest operating system (aka guest OS) kernel intended to run on the virtual multi-processor.

The Virtual Iron Machine Architecture extends existing hardware architectures, virtualizing access to various low-level processor, memory and I/O resources. The software incorporates a type of Hybrid Virtual Machine Monitor [Robin-00], executing non-privileged instructions (a subset of the hardware platform's Instruction Set Architecture) natively in hardware, but replacing the ISA's privileged instructions with a set of (sys)calls that provide the missing functionality on the virtual server. Because the virtual server does not support the full hardware ISA, it's not a virtual instance of the underlying hardware architecture, but rather a virtual instance of the Virtual Iron Machine Architecture (aka Virtual Hardware), having the following crucial properties:

- The virtual hardware acts like a multi-processor with shared memory.
- Applications can run natively "as is," transparently using resources (memory, CPU and I/O) from all physical servers comprising the virtual multi-processor as needed.
- Virtual servers are isolated from one another, even when sharing underlying hardware. At a minimum, this means a software failure in one virtual server does *not* affect[1] the operation of other virtual servers. We also prevent one virtual server from seeing the internal state (including deallocated memory contents) of another. This property is preserved even in the presence of a maliciously exploitive (or randomly corrupted) OS kernel.

Guaranteeing the last two properties simultaneously requires a hardware platform with the following key architectural features[Goldberg-72]:

- At least two modes of operation (aka privilege levels, or rings) (but three is better for performance reasons)

[1] Unreasonably, that is. Some performance degradation can be expected for virtual servers sharing a CPU, for example. But there should be no way for a misbehaving virtual server to starve other virtual servers of a shared resource.

- A method for non-privileged programs to call privileged system routines
- A memory relocation or protection mechanism
- Asynchronous interrupts to allow the I/O system to communicate with the CPU

Like most modern processor architectures, the Intel IA-32 architecture has all of these features. Only the Virtual Iron CVMM is allowed to run in kernel mode (privilege level 0) on the real hardware. Virtual server isolation implies the guest OS cannot have uncontrolled access to any hardware features (such as the CPU control registers) nor to certain low-level data structures (such as the paging directories/tables and interrupt vectors).

Since the IA-32 has four privilege levels, the guest OS kernel can run at a level more highly privileged than user mode (privilege level 3), though it may not run in kernel mode (privilege level 0, reserved for the CVMM). So the LINUX kernel runs in supervisor mode (privilege level 1) in order to take advantage of the IA-32's memory protection hardware to keep applications from accessing pages meant only for the kernel.

2 System Design

In the next few sections we describe the basic design of our system. First, we mention the features of the virtualization that our CVMM provides. Next, we introduce the architecture of our system and how virtual resources are mapped to physical resources. And lastly we describe the LINUX port to this new virtual machine architecture.

2.1 Virtual Machine Features

The CVMM creates an illusion of a shared memory virtual multi-processor. Key features of our virtualization are summarized below:

- The CVMM supports an Intel® ISA architecture of modern Intel processors (such as Intel XEON™).
- Individual VMMs within the CVMM are not implemented as a traditional virtual machine monitor, where a complete processor ISA is exposed to the guest operating system; instead a set of data structures and APIs abstract the underlying physical resources and expose a "virtual processor" architecture with a conceptual ISA to the guest operating system. The instruction set used by a guest OS is similar, but not identical to that of the underlying hardware. This results in a greatly improved performance, however it does require modifications to the guest operating system. This approach to processor virtualization is known in the industry as hybrid virtualization or as paravirtualization[Whitaker-00].
- The CVMM supports multiple virtual machines running concurrently in complete isolation. In the Virtual Iron architecture, the CVMM provides a distributed hardware sharing layer via the virtual multiprocessor machine. This virtual multiprocessor machine provides access to the basic I/O, memory and processor abstractions. A request to access or manipulate these items is handled via the ViMA APIs presented by the CVMM.
- Being a clustered system Virtual Iron VFe provides *dynamic resource management*, such as node eviction or addition visible

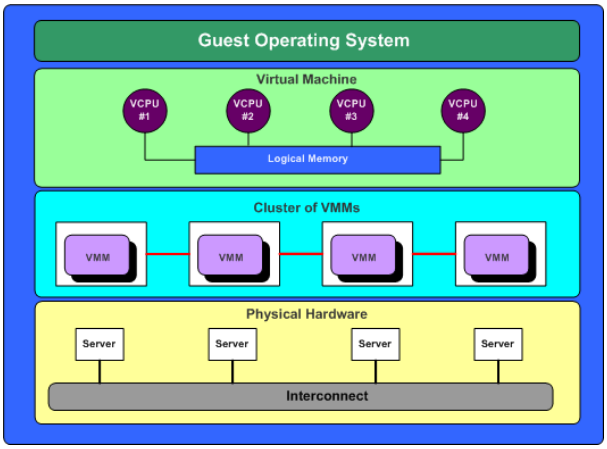

Figure 1: A Cluster of VMMs supporting a four processor VM.

to the Virtual Machine as hot-plug processor(s), memory and device removal or addition respectively.

We currently support LINUX as the guest OS; however the underlying architecture of the Virtual Iron VFe is applicable to other operating systems.

2.2 Architecture

This section outlines the architecture of Virtual Iron VFe systems. We start with differing views of the system to introduce and reinforce the basic concepts and building blocks. The Virtual Iron VFe system is an aggregation of component systems that provide scalable capabilities as well as unified system management and reconfiguration. Virtual Iron VFe software creates a shared memory multi-processor system out of component systems which then runs a single OS image.

2.2.1 Building Blocks

Each system is comprised of elementary building blocks: processors, memory, interconnect, high-speed I/O and software. The basic hardware components providing processors and memory are called nodes. Nodes are likely to be packages of several components such as processors, memory, and I/O controllers. All I/O in the system is performed over interconnect fabric, fibre channel, or networking controllers. All elements of the system present a shared memory multiprocessor to the end applications. This means a unified view of memory, processors and I/O. This level of abstraction is provided by the CVMM managing the processors, memory and the interconnect fabric.

2.2.2 Server View

Starting from the top, there is a virtual server running guest operating system, such as RHAS, SUSE, etc. The guest operating system presents the expected multi-threaded, POSIX server instance running multiple processes and threads. Each of these threads utilizes resources such as processor time, memory and I/O. The virtual server is configured as a shared memory multi-processor. This results in a number of processors on which the guest operating system may schedule processes and threads. There is a unified view of devices, memory, file systems, buffer caches and other operating system items and abstractions.

2.2.3 System View

The Building Blocks View differs from the previously discussed Server View in significant ways. One is a collection of unshared components. The other is a more typical, unified, shared memory multi-processor system. This needs to be reconciled. The approach that we use is to have the CVMM that presents Virtual Processors (VPs) with unified logical memory to the guest OS, and maps these VPs onto the

physical processors and logical memory onto distributed physical memory. A large portion of the instruction set is executed by the machine's physical processor without CVMM intervention, the resource control and management is done via ViMA API calls into the CVMM. This is sufficient to create a new machine model/architecture upon which we run the virtual server. A virtual server is a collection of virtual processors, memory and virtual I/O devices. The guest OS runs on the virtual server and the CVMM manages the mapping of VPs onto the physical processor set, which can change as the CVMM modifies the available resources.

Nodes are bound into sets known as Virtual Computers (VC). Each virtual computer must contain at least one node. The virtual computers are dynamic in that resources may join and leave a virtual computer without any system interruption. Over a longer time frame, virtual computers may be created, destroyed and reconfigured as needed. Each virtual computer may support multiple virtual servers, each running a single instance of an operating system. There are several restrictions on virtual servers, virtual computers, and nodes. Each virtual server runs on a single virtual computer, and may not cross virtual computers. An individual node is mapped into only a single virtual computer.

The virtual server guest operating system, LINUX for instance, is ported to run on a new virtual hardware architecture (more details on this further in the document). This new virtual hardware architecture is presented by the CVMM. From the operating system point of view, it is running on a shared memory multi-processor system. The virtual hardware still performs the computational jobs that it always has, including context switching between threads.

In summary, the guest operating system runs on a shared memory multi-processor system of new design. The hardware is managed by the CVMM that maps physical resources to virtual resources.

3 Implementation of the CVMM

In this section we describe how the CVMM virtualizes processors, memory and I/O devices.

3.1 Cluster of VMMs (CVMM)

The CVMM is the software that handles all of the mapping of resources from the physical to virtual. Each node within a CVMM cluster runs an instance of the VMM, and these instances form a shared-resource cluster that provides the services and architecture to support the virtual computers and appear as a single shared memory multi-processor system. The resources managed by the CVMM include:

- Nodes
- Processors
- Memory, local and remote
- I/O (devices, buses, interconnects, etc)
- Interrupts, Exceptions and Traps
- Inter-node communication

Each collection of communicating and cooperating VMMs forms a virtual computer. There is a one-to-one mapping of virtual computer to the cluster of VMMs. The CVMM is re-entrant and responsible for the scheduling and management of all physical resources. It is as thin a layer as possible, with a small budget for the overhead as compared to a bare LINUX system.

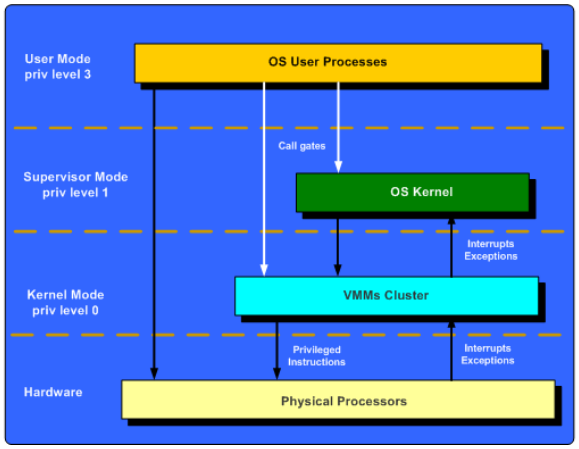

Figure 2: Virtual Iron paravirtualization (Intel IA-32).

3.2 Virtualizing Processors

Each of the physical processors is directly managed by the CVMM and only by the CVMM. A physical processor is assigned to a single virtual computer, and may be used for any of the virtual servers that run on that virtual computer. As we stated before, the method of virtualizing the processors that is used in the Virtual Iron VFe is *paravirtualization*. The diagram in *Figure 2* illustrates our implementation of the paravirtualization concept on the IA-32 platform:

In this scheme, the vast majority of the virtual processor's instructions are executed by the real processor without any intervention from the CVMM, and certain privileged instructions used by the guest OS are rewritten to use the ViMA APIs. As with other VMMs, we take advantage of underlying memory protection in the Intel architecture. The CVMM runs in the privilege ring-0, the guest OS runs in ring-1 and the user applications run in ring-3. The CVMM is the only entity that runs in ring-0 of the processor and it is responsible for managing all operations of the processor, such as booting, initialization, memory, exceptions, and so forth.

Virtual processors are mapped to physical processors by the CVMM. There are a number of rules that are followed in performing this mapping:

- Virtual processors are scheduled concurrently, and there are never more virtual processors than physical processors within a single virtual server.
- The CVMM maintains the mapping of virtual processors to physical processors.
- Physical processors may belong to a virtual computer, but are not required to be used or be active for any particular virtual server.

These rules lead to a number of conclusions. First, any number of physical processors may be assigned to a virtual computer. However, at any given moment, the number of active physical processors is the same as the number of virtual processors in the current virtual server. Moreover, the number of active virtual processors in a virtual server is less than or equal to the number of physical processors available. For instance, if a node is removed from a virtual computer, it may be necessary for a virtual server on that virtual computer to reduce the number of active virtual processors.

3.3 Interrupts, Traps and Exceptions

The CVMM is set-up to handle all interrupts, traps and exceptions. Synchronous traps and exceptions are mapped directly back into the running virtual processor. Asynchronous events, such as interrupts, have additional logic such that they can be mapped back to the appropriate virtual server and virtual processor.

3.4 Virtualizing Memory

Memory, like processors, is a resource that is shared across a virtual computer and used by the virtual servers. Shared memory implemented within a distributed system naturally results in non-uniform memory access times. The CVMM is responsible for memory management, initialization, allocation and sharing. Virtual servers are not allowed direct access to the page tables. However, these tables need to be managed to accommodate a number of goals:

- First, they need to be able to specifically locate an individual page which may reside in any one of the physical nodes
- They must be able to allow several levels of cost. That is, the virtual server should be able to manipulate page tables at the lowest possible cost in most instances to avoid round-trips through the CVMM.
- Isolation is a requirement. No virtual server should be able to affect any other virtual server. If several virtual servers are running on the same virtual computer, then any failures, either deliberate or accidental, should not impact the other virtual servers.

The illusion of a shared memory multiprocessor system is maintained in the Virtual Iron architecture by the sharing of the memory resident in all of the nodes in a virtual computer. As various processors need access to pages of memory, that memory needs to be resident and available for local access. As processes, or the kernel, require access to pages, the CVMM is responsible for insuring the relevant pages are accessible. This may involve moving pages or differences between the modified pages.

3.5 Virtualizing I/O Devices

Just as with the other resources, the CVMM manages all I/O devices. No direct access by a virtual server is allowed to any I/O device, control register or interrupt. The ViMA provides APIs to access the I/O devices. The virtual server uses these APIs to access and control all I/O devices.

All I/O for the virtual computer is done via interfaces and mechanisms that can be shared across all the nodes. This requires a set of drivers within the CVMM that accommodate this, as well as a proper abstraction at the level of a virtual server to access the Fibre Channel and Ethernet.

With the previous comments in mind, the job of the CVMM is to present a virtualized I/O interface between the virtual computer physical I/O devices and the virtual servers. This interface provides for both sharing and isolation between virtual servers. It follows the same style and paradigm of the other resources managed by the CVMM.

4 LINUX Kernel Port

This section describes our port of the LINUX 2.6 kernel to the IA-32 implementation of the Virtual Iron Machine Architecture. It doesn't specify the architecture in detail, but rather describes our general approach and important pitfalls and optimizations, many of which can apply to other architectures real and virtual. First we'll look at the essential porting work required to make the kernel run correctly, then at the more substantial work to make it run well, and finally at the (substantial) work required to support dynamic reconfiguration changes.

4.1 Basic Port

We started with an early 2.6 LINUX kernel, deriving our architecture port from the i386 code base. The current release as of this writing is based on the 2.6.9 LINUX kernel. As the burden of maintaining a derived architecture is substantial, we are naturally interested in cooperating with various recent efforts to refactor and generalize support for derived (e.g., x86_64) and virtualized (e.g., Xen) architectures.

The ViMA interface is mostly implemented via soft interrupts (like syscalls), though memory-mapped interfaces are used in some special cases where performance is crucial. The data structures used to communicate with the CVMM (e.g., descriptor tables, page tables) are close, if not identical, to their IA-32 equivalents.

The basic port required only a single modification to common code, to allow architectures to override *alloc_pgd()* and *free_pgd()*. Though as we'll see later, optimizing performance and adding more dynamic reconfiguration support required more common code modifications.

The virtual server configuration is always available to the LINUX kernel. As mentioned earlier, it exposes the topology of the underlying hardware: a cluster of *nodes*, each providing memory and (optionally) CPUs. The configuration also describes the virtual devices available to the server. Reading virtual server configuration replaces the usual boot-time BIOS and ACPI table parsing and PCI bus scanning.

4.2 Memory Management

The following terms are used throughout this section to describe interactions between the LINUX guest operating system and the Virtual Iron CVMM.

Page Type	Meaning
Physical page	A local memory instance (copy) of a ViMA logical page. The page contents are of interest to the owning virtual server.
Physical page frame	A local memory container, denoted by a specific physical address, managed by the CVMM.
Logical page	A virtual server page, the contents of which are managed by the guest operating system. *The physical location of a logical page is not fixed, nor even exposed to the guest operating system.*
Logical page frame	A logical memory container, denoted by a specific logical address, managed by the guest operating system.
Replicated page	A logical page may be replicated on multiple nodes as long as the contents are quaranteed to be identical. Writing to a replicated logical page will invalidate all other copies of the page.

Table 1: Linux Memory Management in Virtual Iron VFe

Isolating a virtual server from the CVMM and other virtual servers sharing the same hardware requires that memory management be carefully controlled by the CVMM. Virtual servers cannot see or modify each other's memory under any circumstances. Even the contents of freed or "borrowed" physical pages are never visible to any other virtual server.

Accomplishing this isolation requires explicit mechanisms within the CVMM. For example, CPU control register cr3 points to the top-level page directory used by the CPU's paging unit. A malicious guest OS kernel could try to point this to a `fake` page directory structure mapping pages belonging to other virtual servers into its own virtual address space. To prevent this, ***only*** *the CVMM can create and modify the page directories / tables used by the hardware, and it must ensure that cr3 is set only to a top-level page directory that it created for the appropriate virtual server.*

On the other hand, the performance of memory management is also of crucial importance. Taking a performance hit on every memory access is not acceptable; the *common case* (in which the desired logical page is in local memory, mapped, and accessible) *suffers no virtualization overhead.*

The MMU interface under ViMA 32-bit architecture is mostly the same as that of the IA-32 in PAE mode, with three-level page tables of 64-bit entries. A few differences exist, mostly that ours map 32-bit virtual addresses to 40-bit logical addresses, and that we use software dirty and access bits since these aren't set by the CVMM.

The page tables themselves live in logical memory, which can be distributed around the system. To reduce possibly-remote page table accesses during page faults, the CVMM implements fairly aggressive software TLB. Unlike the x86 TLB, the CVMM supports Address Space Number tags, used to differentiate and allow selective flushing of translations from different page tables. The CVMM TLB is naturally kept coherent within a node, so a lazy flushing scheme is particularly useful since (as we'll see later) we try to minimize cross-node process migration.

4.3 Virtual Address Space

The standard 32-bit LINUX kernel reserves the last quarter (gigabyte) of virtual addresses for its own purposes. The bottom three quarters of virtual addresses makes up the standard process (user-mode) address space.

Much efficiency is to be gained by having the CVMM share its virtual address space with the guest OS. So the LINUX kernel is mapped into the top of the CVMM's user-space, somewhat reducing the virtual address space available to LINUX users. The amount of virtual address space required by the CVMM depends on a variety of factors, including requirements of drivers for the real hardware underneath. This overhead becomes negligible on 64-bit architectures.

4.4 Booting

As discussed earlier, a guest OS kernel runs at privilege level 1 in the IA-32 ViMA. We first replaced the privileged instructions in the arch code by syscalls or other communication with the CVMM. Kernel execution starts when the CVMM is told to start a virtual server and pointed at the kernel. The boot virtual processor then starts executing the boot code. VPs are always running in protected mode with paging enabled, initially using a `null` page table signifying direct (logical = virtual) mapping. So the early boot code is fairly trivial, just establishing a stack pointer and setting up the initial kernel page tables before dispatching to C code. Boot code for secondary CPUs is even more trivial since there are no page tables to build.

4.5 Interrupts and Exceptions

The LINUX kernel registers handlers for interrupts with the CVMM via a virtualized Inter-

rupt Descriptor Table. Likewise, the CVMM provides a virtualized mechanism for masking and unmasking interrupts. Any information (e.g., cr2, etc.) necessary for processing an interrupt or exception that is normally readable only at privilege level 0 is made available to the handler running at level 1. Interrupts actually originating in hardware are delivered to the CVMM, which processes them and routes them when necessary to the appropriate virtual server interrupt handlers. Particular care is taken to provide a "fast path" for exceptions (like page faults) and interrupts generated and handled locally.

Particularly when sharing a physical processor among several virtual servers, interrupts can arrive when a virtual server is not currently running. In this case, the interrupt(s) are `pended`, possibly coalescing several for the same device into a single interrupt. Because the CVMMs handle all actual device communication, LINUX is not subject to the usual hardware constraints requiring immediate processing of device interrupts, so such coalescing is not dangerous, provided that the interrupt handlers realize the coalescing can happen and act accordingly.

4.6 I/O

The ViMA I/O interface is designed to be flexible and extensible enough to support new classes of devices as they come along. The interface is not trying to present something that looks like `real hardware`, but rather higher-level generic conduits between the guest OS and the CVMM. That is, the ViMA itself has no understanding of I/O operation semantics; it merely passes data and control signals between the guest operating system and the CVMM. It supports the following general capabilities:

- device discovery
- device configuration
- initiation of (typically asynchronous) I/O operations
- completion of asynchronous I/O operations

Because I/O performance is extremely important, data is presented in large chunks to mitigate the overhead of going through an extra layer. The only currently supported I/O devices are Console (VCON), Ethernet (VNIC), and Fibre Channel storage (VHBA). We have implemented the `bottom layer` of three new device drivers to talk to the ViMA, while the interface from above remains the same for drivers in the same class. Sometimes the interface from above is used directly by applications, and sometimes it is used by higher-level drivers. In either case, the upper levels work "as is."

In almost all cases, completion interrupts are delivered on the CPU that initiated the operation. But since CPUs (and whole nodes) may be dynamically removed, LINUX can steer outstanding completion interrupts elsewhere when necessary.

4.7 Process and Thread Scheduling

The CVMM runs one task per virtual processor, corresponding to its main thread of control. The LINUX kernel further divides these tasks to run LINUX processes and threads, starting with the vanilla SMP scheduler. This approach is more like the one taken by CoVirt[King-03] and VMWare Workstation[Sugerman-01], as opposed to having the underlying CVMM schedule individual LINUX processes as done in L4[Liedtke-95] . This is consistent with our

general approach of exposing as much information and control as possible (without compromising virtual server isolation) to the guest OS, which we assume can make better decisions because it knows the high-level context. So, other than porting the architecture-specific context-switching code, no modifications were necessary to use the LINUX scheduler.

4.8 Timekeeping

Timekeeping is somewhat tricky on such a loosely coupled system. Because the *jiffies* variable is used all over the place, updating the global value on every clock interrupt generates prohibitively expensive cross-node memory traffic. On the other hand, LINUX expects jiffies to progress uniformly. Normally *jiffies* is aliased to *jiffies_32*, the lower 32 bits of the full 64-bit *jiffies_64* counter. Through some linker magic, we make *jiffies_32* point into a special per-node page (a page whose logical address maps to a different physical page on each node), so each node maintains its own *jiffies_32*. The global *jiffies_64* is still updated every tick, which is no longer a problem since most readers are looking at *jiffies_32*. The local *jiffies_32* values are adjusted (incrementally, without going backwards) periodically to keep them in sync with the global value.

4.9 Crucial Optimizations

The work described in the previous sections is adequate to boot and run LINUX, but the resulting performance is hardly adequate for all but the most contrived benchmarks. The toughest challenges lie in minimizing remote memory access (and communication in general).

Because the design space of potentially useful optimizations is huge, we strive to focus our optimization efforts by guiding them with performance data. One of the advantages of a virtual machine is ease of instrumentation. To this end, our CVMM has a large amount of (optional) code devoted to gathering and reporting performance data, and in particular for gathering information about cross-node memory activity. Almost all of the optimizations described here were driven by observations from this performance data gathered while running our initial target applications.

4.9.1 Logical Frame Management and NUMA

When LINUX runs on non-virtualized hardware, page frames are identified by physical address, but when it runs on the ViMA, page frames are described by `logical` address. Though logical page frames are analogous to physical page frames, logical page frames have somewhat different properties:

- Logical page frames are dynamically mapped to physical page frames by the CVMM in response to page faults generated while the guest OS runs
- Logical page frames consume physical page frames only when mapped and referenced by the guest OS.
- The logical page frames reserved by the CVMM are independent of the physical page frames reserved for PC-compatible hardware and BIOS.

Suppose we have a system consisting of four dual-processor SMP nodes. Such a system can be viewed either as a “flat” eight-processor SMP machine or (via `CONFIG_NUMA`) as a two-level hierarchy of four two-processor nodes (i.e., the same as the underlying hardware). While the former view works correctly,

hiding the real topology has serious performance consequences. The NUMA kernel assumes each node manages its own range of physical pages. Though pages can be used anywhere in the system, the NUMA kernel tries to avoid frequent accesses to remote data.

In some sense, the ViMA can be treated is a virtual cache coherent NUMA (ccNUMA) machine, in that access to memory is certainly non-uniform.

By artificially associating contiguous logical page ranges with nodes, we can make our virtual server look like a ccNUMA machine. We realized much better performance by treating the virtual machine as a ccNUMA machine reflecting the underlying physical hardware. In particular the distribution of memory into more zones alleviates contention for the zone lock and lru_lock. Furthermore, the optimizations that benefit most ccNUMA machines benefit ours. And the converse is true as well. We're currently cooperating with other NUMA LINUX developers on some new optimizations that should benefit all large ccNUMA machines.

For various reasons, the most important being some limitations of memory removal support, we currently have a fictitious CPU-less `node 0` that manages all of low memory (the DMA and NORMAL zones). So HIGHMEM is divvied up between the actual nodes in proportion to their relative logical memory size.

4.9.2 Page Cache Replication

To avoid the sharing of page metadata by nodes using replicas of read-only page cache pages, we have implemented a NUMA optimization to replicate such pages on-demand from node-local memory. This improves benchmarks that do a lot of exec'ing substantially.

4.9.3 Node-Aware Batch Page Updates

Both *fork()* and *exit()* update page metadata for large numbers of pages. As currently coded, they update the metadata in the order the pages are walked. We see measurable improvements by splitting this into multiple passes, each updating the metadata only for pages on a specific node.

4.9.4 Spinlock Implementation

The i386 spinlock implementation also turned out to be problematic, as we expected. The atomic operation used to try to acquire a spinlock requires write access to the page. This works fine if the lock isn't under contention, particularly if the page is local. But if someone else is also vying for the lock, spinning as fast as possible trying to access remote data and causing poor use of resources. We continue to experiment with different spinlock implementations (which often change in response to changes in the memory access characteristics of the underlying CVMM). Currently we always try to get the lock in the usual way first. If that fails, we fall into our own `spinlock_wait()` that does a combination of "remote" reads and yielding to the CVMM before trying the atomic operation again. This avoids overloading the CVMM to the point of restricting useful work from being done.

4.9.5 Cross-Node Scheduling

The multi-level scheduling domains introduced in 2.6 LINUX kernel match very nicely with a hierarchical system like Virtual Iron VFe. However, we found that the cross-node scheduling decisions in an environment like this are based on much different factors than the schedulers for more tightly-coupled domains.

Moreover, because cross-node migration of a running program is relatively expensive, we want to keep such migrations to a minimum. So the cross-node scheduler can run *much* less often than the other domain schedulers, so it becomes permissible to take a little longer making the scheduling decision and take more factors into account. In particular, task and node memory usage are crucial—much more important than CPU load. So we have implemented a different algorithm for the cross-node scheduling domain.

The cross-node scheduling algorithm represents node load (and a task's contribution to it) with a 3-d vector whose components represent CPU, memory, and I/O usage. Loads are compared by taking the vector norm[Bubendorfer-96]. While we're still experimenting heavily with this scheduler, a few conclusions are clear. First, memory matters far more than CPU or I/O loads in a system like ours. Hence we weight the memory component of the load vector more heavily than the other two. It's also important to be smart about how tasks share memory.

Scheduling and logical memory management is tightly intertwined. Using the default NUMA memory allocation, processes try to get memory from the node on which they're running when they allocate. We'd prefer that processes use such local pages so they don't fight with other processes or the node's swap daemon when memory pressure rises. This implies that we would rather avoid moving a process after it allocates its memory. Redistributing a process to another node at *exec()* time makes a lot of sense, since the process will have its smallest footprint at that point. Processes often share data with other processes in the same process group. So we've modified *sched_exec()* to consider migrating an exec'ing process to another node only if it's a process group leader (and with even more incentive—via a lower imbalance threshold—for session group leaders). Furthermore, when *sched_exec()* does consider migrating to another node, it looks at the 3-d load vectors described earlier. This policy has been particularly good for distributing the memory load around the nodes.

4.10 Dynamic Reconfiguration Support (Hotplug Everything)

When resources (CPU or memory) are added or removed from a the cluster, the CVMM notifies the guest OS via a special "message" interrupt also used for a few other messages ("shutdown," "reboot," etc.). LINUX processes the interrupt by waking a message handler thread, which then reads the new virtual server configuration and starts taking the steps necessary to realize it. Configuration changes occur in two phases. During the first phase, all resources being removed are going away. LINUX acknowledges the change when it has reduced its resource usage accordingly. The resources are, of course, not removed until LINUX acknowledges. During the second phase, all resources being added are added (this time before LINUX acknowledges), so LINUX simply adds the new resources and acknowledges that phase. This implies certain constraints on configuration changes. For example, there must be at least one VP shared between the old and new configurations.

4.10.1 CPU and Device Hotplug

Adding and removing CPUs and devices required some porting to our methods of starting and stopping CPUs and devices, but for the most part this is much easier with idealized virtual hardware than with the real thing.

4.10.2 Node Hotplug

Adding and removing whole nodes was a little more problematic as most iterations over nodes in the system assumes online nodes are contiguous going from 0 to *numnodes-1*. Node removal can leave a "hole" which invalidates this assumption. The CPUs associated with a node are made "physically present" or absent as the node is added or removed.

4.10.3 Memory Hotplug

Memory also comes with a node (though nodes' logical memory can be increased or decreased without adding or removing nodes), and must be made hotpluggable. Unfortunately our efforts in this area proceeded independently for quite a while until we encountered the memory hotplug effort being pursued by other members of LINUX development community. We've decided to combine our efforts and plan on integrating with the new code once we move forward from 2.6.9 code base. Adding memory isn't terribly hard, though some more synchronization is needed. At the global level, a memory hotplug semaphore, analogous to the CPU hotplug semaphore, was introduced. Careful ordering of the updates to the zones allows most of the existing references to zone memory info to continue to work without locking.

Removing memory is much more difficult. Our existing approach removes only high memory, and does so by harnessing the swap daemon. A new page bit, *PG_capture*, is introduced (name borrowed from the other memory hotplug effort) to mark pages that are destined for removal. Such pages are swapped out more aggressively so that they may be reclaimed as soon as possible. Freed pages marked for capture are taken off the free lists (and out of the per-cpu pagesets), zeroed (so the CVMM can forget them), then counted as removed. During memory removal, the swap daemons on nodes losing memory are woken often to attempt to reclaim pages marked for capture. In addition, we try reclaiming targeted pages from the shrinking zones' active lists.

This approach works well on a mostly idle (or at least suspended) machine, but has a number of weaknesses, particularly when the memory in question is being actively used. Direct page migration (bypassing swap) would be an obvious performance improvement. There are pages that can't be removed for various reasons. For example, pages locked into memory via *mlock()* can't be written to disk for security reasons.

But because our logical pages aren't actually tied to nodes (but just artificially assigned to them for management purposes), we can tolerate a substantial number of "unremovable" pages. A node that has been removed, but still has some "unremovable" pages is known as a "zombie" node. No new pages are allocated from the node, but existing pages and zone data are still valid. We'll continue to try and reclaim the outstanding pages via the node's swap daemon (now running on a different node, of course). If another node is added in its place before all pages are removed, the new node can subsume the "unremovable" pages and it becomes a normal page again. In addition, it is also possible to exchange existing free pages for "unremovable" pages to reclaim space for replicas. While this scheme is currently far from perfect or universal, it works predictably in enough circumstances to be useful.

5 Conclusion

In this paper we have presented a Clustered Virtual Machine Monitor that virtualizes a set

of distributed resources into a shared memory multi-processor machine. We have ported LINUX Operating System onto this platform and it has shown to be an excellent platform for deploying a wide variety of general purpose applications.

6 Acknoweledgement

We would like to thank all the members of the Virtual Iron Software team without whom Virtual Iron VFe and this paper would not be possible. Their contribution is gratefully acknowledged.

References

[Pfister-98] Gregory F. Pfister. *In Search of Clusters, Second Edition, Prentice Hall PTR*, pp. 358–369, 1998.

[Goldberg-74] R.P. Goldberg. Survey of Virtual Machines Research. *Computer*, pp. 34–45, June 1974.

[Robin-00] J.S. Robin and C.E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, pp. 3–4, August 20, 2000.

[Goldberg-72] R.P. Goldberg. *Architectural Principles for Virtual Computer Systems.* Ph.D. Thesis, Harvard University, Cambridge, MA, 1972.

[Whitaker-00] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *ACM SIGOPS Operating System Rev.*, vol. 36, no SI, pp. 195–209, Winter 2000.

[King-03] S. King, G. Dunlap, and P. Chen Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Technical Conference*, 2003.

[Sugerman-01] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMWare Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, June, 2001.

[Liedtke-95] Dr. Jochen Liedtke. On Micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December, 1995.

[Bubendorfer-96] K.P. Bubendorfer. *Resource Based Policies for Load Distribution.* Masters Thesis, Victoria University of Wellington, Wellington, New Zealand, 1996.

Clusterproc: Linux Kernel Support for Clusterwide Process Management

Bruce J. Walker
Hewlett-Packard
bruce.walker@hp.com

Laura Ramirez
Hewlett-Packard
laura.ramirez@hp.com

John L. Byrne
Hewlett-Packard
john.l.byrne@hp.com

Abstract

There are several kernel-based clusterwide process management implementations available today, providing different semantics and capabilities (OpenSSI, openMosix, bproc, Kerrighed, etc.). We present a set of hooks to allow various installable kernel module implementations, with a high degree of flexibility and virtually no performance impact. Optional capabilities that can be implemented via the hooks include: clusterwide unique pids, single init, heterogeneity, transparent visibility and access to any process from any node, ability to distribute processes at exec or fork or thru migration, file inheritance and full controlling terminal semantics, node failure cleanup, clusterwide `/proc/<pid>`, checkpoint/restart and scale to thousands of nodes. In addition, we describe an OpenSSI-inspired implementation using the hooks and providing all the features described above.

1 Background

Kernel based cluster process management (CPM) has been around for more than 20 years, with versions on Unix by Locus[1] and Mosix[2]. The Locus system was a general purpose Single System Image (SSI) cluster, with a single root filesystem and a single namespace for processes, files, networking and interprocess communication objects. It provided high availability as well as a simple management paradigm and load balancing of processes. Mosix focused on process load balancing. The concepts of Locus have moved to Linux via the OpenSSI[3] open source project. Mosix has moved to Linux via the openmosix[4] project.

OpenSSI and Mosix were not initially targeted at large scale parallel programming clusters (eg. those using MPI). The BProc[5] CPM project has targeted that environment to speed up job launch and simplify process management and cluster management. More recent efforts by Kerrighed[6] and USI[7] (now Cassat[8]) were also targeted at HPC environments, although Cassat is now interested in commercial computing.

These 5 CPM implementations have somewhat different cluster models (different forms of SSI) and thus fairly different implementations, in part driven by the environment they were originally developed for. The “Introduction to SSI” paper[10] details some of the differences. Here we outline some of the characteristics relevant to CPM. Mosix started as a workstation technology that allowed a user on one workstation to utilize cpu and memory from another workstation by moving running processes (process migration) to the other workstation. The mi-

grated processes had to see the OS view of the original workstation (home node) since there was no enforced common view of resources such as processes, filesystem, ipc objects, binaries, etc. To accomplish the home-node view, most system calls had to be executed back on the home node—the process was effectively split, with the kernel part on the home node and the application part elsewhere. What this means to process ids is that home nodes generate ids which are not clusterwide unique. Migrated processes retain their home node pid in a private data structure but are assigned a local pid by the current host node (to avoid pid conflicts). The BProc model is similar except there is a single home node (master node) that all processes are created on. These ids are thus clusterwide unique and a local id is not needed on the host node.

The model in OpenSSI, Kerrighed and Cassat is different. Processes can be created on any node and are given a single clusterwide pid when they are created. They retain that pid no matter where they execute. Also, the node the process was created on does not retain part of the process. What this means to the CPM implementation is that actions to be done against processes are done where the process is currently executing and not on the creation or home node.

There are many other differences among the CPM implementations. For example, OpenSSI has a single, highly available init process while most/all other implementations do not. Additionally, BProc does not retain a controlling terminal (or any other open files) when processes move, while other implementations do. Some implementations, like OpenSSI, support clusterwide ptrace, while others do not.

With some of these differences in mind, we next look at the goals for a set of CPM hooks that would satisfy most of the CPM implementations.

2 Goals and Requirements for the Clusterproc Hooks

The general goals for the hooks are to enable a variety of CPM implementations while being non-invasive enough to be accepted in the base kernel. First we look at the base kernel requirements and then some of the functional requirements.

Changes to the base kernel should retain the architectural cleanliness and not affect the performance. Base locking should be used and copies of base routines should be avoided. The clusterproc implementations should be installable modules. It should be possible to build the kernel with the hooks disabled and that version should have no impact on performance. If the hooks are enabled, the module should be optional. Without the module loaded, performance impact should be negligible. With the module loaded, one would have a one node cluster and performance will depend on the CPM implementation.

The hooks should enable at least the following functionality:

- optionally have a per process data structure maintained by the CPM module;
- allowing for the CPM module to allocate clusterwide process ids;
- support for distributed process relationships including parent/child, process group and session; optional support for distributed thread groups and ptrace parent;
- optional ability to move running processes from one node to another either at exec/fork time or at somewhat arbitrary points in their execution;

- optional ability to transparently checkpoint/restart processes, process groups and thread groups;
- optional ability to have process continue to execute even if the node they were created on leaves the cluster;
- optional ability to retain relationships of remaining processes, no matter which nodes may have crashed;
- optional ability to have full controlling terminal semantics for processes running remotely from their controlling terminal device;
- full, but optional `/proc/<pid>` capability for all processes from all nodes;
- capability to support either an "init" process per node or a single init for the entire cluster;
- capability to function within a shared root environment or in an environment with a root filesystem per node;
- capability to be an installable module that can be installed either from the ramdisk/initramfs or shortly thereafter;
- support for clusters of up to 64000 nodes, with optional code to support larger;

In the next section we detail a set of hooks designed to meet the above set of goals and requirements. Following that is the design of the OpenSSI 3.0, as adapted to the proposed hooks.

3 Proposed Hook Architecture, Hook Categories and Hooks

To enable the optional inclusion of clusterwide process management (referred also as "clusterproc" or CPM) capability, very small data structure additions and a set of entry points are proposed. The data structure additions are a pointer in the task structure (CPM implementations could then allocate a per process structure that this pointer points to), and 2 flag bits. The infrastructure for the hooks is patterned after the security hooks, although not exactly the same. If `CONFIG_CLUSTERPROC` is not set, the hooks are turned into inline functions that are either empty or return the default value. With `CONFIG_CLUSTERPROC` defined, the hook functions call clusterproc ops if they are defined, otherwise returning the default value. The ops can be replaced, and the clusterproc install-able module will replace the ops with routines to provide the particular CPM implementation. The clusterproc module would be loaded early in boot. All the code to support the clusterwide process model would be under GPL. To enable the module some additional symbols will have to exported to GPL modules.

The proposed hooks are grouped into categories below. Each CPM implementation can provide op functions for all or some of the hooks in each category. For each category we list the relevant hook functions in pseudo-C. The names would actually be `clusterproc_xxx` but to fit here we leave out the `clustproc_` part. The parameters are abreviated. For each category, we describe the general purpose of the hooks in that category and how the hooks could be used in different CPM implementations. The categories are: Init and Reaper; Allocation/ Free; Update Parent; Process lock/unlock; Exit/Wait/Reap; Signalling; Priority and Capability; Setpgid/Setsid; Ptrace; Controlling Terminal; and Process movement;

3.1 Init and Reaper

```
void single_init();
```

```
void child_reaper(*pid);
```

One of the goals was to allow the cluster to run with a single init process for the cluster. The `single_init` hook in `init/main.c`, `init()` can be used in a couple of ways. First, if there is to be a single init, this routine can spawn a "reaper" process that will locally reap the orphan processes that init normally reaps. On the node that is going to have the init, the routine returns to allow init to be exec'd. On other nodes it can exit so there is no process 1 on those nodes. The other hook in this category is `child_reaper`, which is in timer.c, `sys_getppid()`. It returns 1 if the process's parent was the `child_reaper` process. Neither of these hooks would be used in those CPM implementations that have an init process per node.

3.2 Allocation/ Free

```
int fork_alloc(*tsk);
void exit_dealloc(*tsk);
int pid_alloc(pid);
int local_pid(pid);
void strip_pid(*pid);
```

There are 5 hooks functions in this category. First is `fork_alloc`, which is called in `copy_process()` in `fork.c`. This routine is called to allow the CPM to allocate a private data structure pointed to by `clusterproc` pointer which is added to the task structure. Freeing that structure is done via the hook `exit_dealloc()` which is called in `release_task()` in `exit.c` and under error conditions in `copy_process()` in `fork.c` The `exit_dealloc` routine can also be used to do remote notifications necessary for pgrp and session management. All CPM implementations will probably use these hooks. The other 3 hooks deal with pid allocation and freeing and are only used in those implementations presenting a clusterwide single pid space. The `pid_alloc` hook is called in `alloc_pidmap()` in `pid.c`. It takes a locally unique pid and returns a clusterwide unique pid, possibly by encoding a node number in some of the high order bits. The `local_pid` and `strip_pid` hooks are in `free_pidmap()`, also in `pid.c` The `local_pid` hook returns 1 if this pid was generated on this node and the process id is no longer needed clusterwide. Otherwise return 0. The `strip_pid` hook is called to undo the effects of `pid_alloc` so the base routines on each node can manage their part of the clusterwide pid space.

3.3 Update parent

```
int update_parent(*ptsk,*ctsk,
            flag,sig,siginfo);
```

`Update_parent` is a very general hook called in several places. It is used by a child process to notify a parent process if the parent process is executing remotely. In `ptrace.c`, it is called in `__ptrace_unlink()` and `__ptrace_link()`. In the arch version of `ptrace.c` it is called in `sys_ptrace()`. In `exit.c` it is called in `reparent_to_init()` and in `fork.c`, `copy_process()`, it is called in the `CLONE_PARENT` case. Although not all CPM implementations will support distributed ptrace or `CLONE_PARENT`, support for some of the instances of this hook will probably be in each CPM implementation.

3.4 Process lock/unlock

```
void proc_lock(*tsk,base_lock);
void proc_unlock(*tsk,base_lock);
```

The proc_lock and proc_unlock hooks allow the CPM implementation to either use the base tsk->proc_lock (default) or to introduce a sleep lock in their private process data structure. In some implementations, a sleep lock is needed because remote operations may be executed while this lock is held. In addition, calls to proc_lock and proc_unlock are added in exit_notify(), in exit.c, because exit_notify() may not be atomic and may need to be interlocked with setpgid() and with process movement (the locking calls for setpgid and process movement would be in the CPM implementation.

3.5 Exit/Wait/Reap

```
int rmt_reparent_children(*tsk);
int is_orphan_pgrp(pgrp);
void rmt_orphan_pgrp(newpgrp,*ret);
void detach_pid(*tsk,nr,type);
int rmt_thread_group_empty(*tsk,pid,opt);
int rmt_wait_task_zombie(*tsk,noreap,
      *siginfo,*stat_addr, *rusage);
int rmt_wait_stopped(*tsk,int,noreap,
       *siginfo,*stat_addr,*rusage);
int rmt_wait_continued(*tsk,noreap,
      *siginfo,*stat_addr, *rusage);
```

There are several hooks proposed to accomplish all the actions around the exit of a process and wait/reap of that process. One early action in exit is to reparent any children to the child_reaper. This is done in forget_original_parent() in exit.c. The rmt_reparent_children hook provides an entry to reparent those children not executing with the parent. Accurate orphan process group processing can be difficult with other pgrp members, children and parents all potentially executing on different nodes. The "home-node" model implementations will have all the necessary information at the home node. For non–home-node implementations like OpenSSI, two hooks are proposed—is_orphan_pgrp and rmt_orphan_pgrp. is_orphan_pgrp is called in is_orphan_pgrp(), in exit.c. It returns 1 if orphaned and 0 if non-orphaned. If not provided, the existing base algorithm is used. rmt_orphan_pgrp is called in will_become_orphaned_pgrp() in exit.c. It is called if there are no local processes remaining that make the process group non-orphan. In that case it determines if the pgrp will become orphan and if so it effects the standard action on the pgrp members. A detach_pid hook in detach_pid() is proposed to allow CPM implementations to update any data structures maintained for process groups and sessions.

There are 4 proposed hooks in wait. The first, in eligible_child(), exit.c, is rmt_thread_group_empty. This is used to determine if the thread group is empty, for thread groups in which the thread group leader is executing remotely. If it is empty, the thread group leader can be reaped; otherwise it cannot. The other 3 hooks are rmt_wait_task_zombie, rmt_wait_stopped and rmt_wait_continued which are called in wait_task_zombie(), wait_task_stopped() and wait_task_continued() respectively. These hooks allow the CPM implementation to move the respective functions to the node where the process is and then execute the base call there, returning an indication if the child was reaped or if there was an error.

3.6 Signalling

```
void rmt_sigproc(pid,sig,*siginfo,*error);
int pgrp_list_local(pgrp,*flag);
int rmt_sigpgrp(pgrp,sig,*signfo);
void sigpgrp_rmt_members(pgrp,sig,*siginfo,
                                *reg,flag);
int kill_all(sig,*siginfo,*count,*ret,tgid);
void rmt_sig_tgkill(tgid,*siginfo,pid,flag,
                             *tsk*,*error);
void rmt_send_sigio(*tsk,*fown,fd,band);
int rmt_pgrp_send_sigio(pgid,*fown,fd,band);
void rmt_send_sigurg(*tsk,*fown);
int rmt_pgrp_send_sigurg(pgid,*fown);
void timedwait(timeout,*timespec,*ret);
```

There are many places in the kernel that may signal a process, for a variety of of reasons. Hooks in `kill_proc_info()` and `kill_pg_info()` handle many cases. One could define a general hook function that handles many of the cases (process, pgrp, killall, sigio, sigurg, etc.). Doing so would reduce the number of different hook functions but would require a superset of parameters and the op would have to relearn the reason it was called. For now we have proposed them as separate hooks. `rmt_sigproc` is the hook in `kill_proc_info()`, called only if the process is not found locally. It tries to find the process on another node and deliver the signal. For the process group case we currently have 3 hooks in `kill_pg_info()`. Based on the assumption that some node knows the list of pgrp members (or at least the nodes they are on), the first hook (`pgrp_list_local`) determines if such a list is local. If not, it calls `rmt_sigpgrp` which will transfer control to such a node, so the base `kill_pg_info()` can be called. Given we are now executing on the node where the list is, the `pgrp_list_local` hook can lock the list so that no members will be missed during the signal operation. After that, the base code to signal locally executing pgrp members is executed, followed by the code to signal remote members (`sigpgrp_rmt_members`). That hook also does the list unlock. Support for clusterwide kill –1 is provided by a hook in `kill_something_info()`. A CPM implementation could loop thru all the nodes in the cluster, calling `kill_something_info()` on each one. Linux has 2 system calls for thread signalling—`sys_tkill()` and `sys_tgkill()`. The proposed hook `rmt_sig_tgkill` is inserted in each of these system calls to find the thread(s) and deliver the signal, if the threads were not already found locally. The `send_sigio()` function in `fcntl.c` can send process or pgrp signals. The `rmt_send_sigio` or `rmt_pgrp_send_sigio` hook is called if the process or process group is remote. Similar hooks are needed in `send_sigurg()` in `fcntl.c` (with different parameters). The final signal related hook is `timedwait`, which is called from `sys_rt_sigtimedwait()`, in `signal.c`. It is called only if a process was in a scheduled timeout and was woken up to do a migrate. It restarts the `sys_rt_sigtimedwait()` after the migrate.

3.7 Priority and Capability

```
void priority(cmd,who,niceval,*tsk,*err);
int pgrp_priority(cmd,who,niceval,*ret);
int prio_user(cmd,who,niceval,*err);
int capability(cmd,pid,header,data,*reg);
int pgrp_capset(pgrp,*effective,*inherit,
                        *permitted,*ret);
int capset_all(*effective,*inherit,
                        *permitted,*ret);
```

In `sys_setpriority()` (`sys.c`), scheduling priority can be set on processes, process groups or "all processes owned by a given user." A get-priority can be done for a process or a pgrp. The `priority`, `pgrp_priority` and `prio_user` hooks are proposed to deal with distributions issues for these functions. Capability setting/getting (`sys_capset()` and `sys_capget()` in `capability.c`) are quite similar and `capability`, `pgrp_capset` and `capset_all` hooks are proposed for those functions.

3.8 Setpgid/Setsid

```
int is_process_local(pid,pgid);
int rmt_setpgid(pid,pgid,caller,sid);
int verify_pgid_session(pgid,sid);
void pgrp_update(*tsk);
void setpgid_done(*tsk,pid);
void rmt_proc_getattr(pid,*pgid,*siod);
void setsid(void);
```

Setpgid (`sys.c`) may be quite straightforward to handle in the home/master node implementations because all the process, process group and

session information will be at the home/master node. For the more peer-oriented implementations, in the most general case there could be several nodes involved. First, while the setpgid operation is most often done against oneself, it doesn't have to be, so there is a hook set early in `sys_setpgid` to move execution to the node on which the setpgid is to be done (`is_process_local` and `rmt_setpgid`). `is_process_local` can also acquire a sleep lock on the process since setpgid may not be atomic to the `tasklist_lock`. One of the tests in setpgid is to make sure there is someone in the proposed process group in the same session as the caller. If that check isn't satisfied locally, `verify_pgid_session` is called to check the rest of the process group. Given the operation is approved, the `pgrp_update` hook is called to allow the CPM implementation to adjust orphan pgrp information, to create or update any central pgrp member list and to update any cached information that might be at the process's parent's execution site (to allow him to easily do waits). A final hook in `sys_setpgid` (`setpgid_done`) is called to allow the CPM implementation to release the process lock acquired in `is_process_local`.

The `rmt_proc_getattr` hook in `sys_getpgid()` and `sys_getsid()` supplies the pgid and/or sid for processes not executing locally.

The `setsid` hook in `sys_setsid()` can be used by the CPM implementation to update cached information at the parent's execution node, at children execution nodes and at any session or pgrp list management nodes.

3.9 Ptrace

```
void rmt_ptrace(request,pid,addr,data,*ret);
*tsk rmt_find_pid(pid);
int ptrace_lock(*tsk,*tsk)
void ptrace_unlock(*tsk,*tsk);
```

Clusterwide ptrace support is not provided in all CPM implementations (eg. BProc) but can be supported with the help of a few hooks. Unfortunately `sys_ptrace()` is in the arch tree, in `ptrace.c`. The `rmt_ptrace` hook is needed if the process to be ptraced is not local. It reissues the call on the node where the process is running. In `ptrace_attach()`, in the non-arch version of `ptrace.c`, the `rmt_find_pid` hook is used in the scenario that the request was generated remotely. This hook helps ensure that the the process being traced is attached to the process debugging and not to a server daemon acting on behalf of that process. The `ptrace_lock` and `ptrace_unlock` hooks are used in `do_ptrace_unlink()` (`ptrace.c`) and `de_thread()` (`exec.c`). They can be used to provide atomicity across operations that require remote messages.

3.10 Controlling Terminal

```
void clear_my_tty(*tsk);
void update_ctty_pgrp(pgrp,npgrp);
void rmt_vhangup(*tsk);
void get_tty(*tsk,*tty_nr,*tty_pgrp);
void clear_tty(sid,*tty,flag);
void release_rmt_tty(*tsk,flag);
int has_rmt_ctty();
int rmt_tty_open(*inode,*file);
void rmt_tty_write_message(*msg,*tty);
int rmt_is_ignored(pid,sig);
```

Some CPM implementation do not support controlling terminal for processes after they move (eg. BProc). In the home-node style CPM, the task structure on the home node will have `tty` pointer. On the node where the process migrated, the task structure has no `tty` pointer. As long as any interrogation or updating using that pointer is done on the home node, this strategy works. For CPM implementations where system calls are done locally, some hooks are needed to deal with a potentially remote controlling terminal. The proposed strategy is that if the controlling terminal

is remote, the `tty` pointer would be null but there would be information in the CPM private data structure.

Daemonize(), in `exit.c`, normally clears the `tty` pointer in the task structure. Additionally it calls the hook `clear_my_tty` to do any other bookkeeping in the case where the controlling terminal is remote. In `drivers/char/tty_io.c`, the routines `do_tty_hangup()`, `disassociate_dev()`, `release_dev()` and `tiosctty()` all call the hook `clear_tty`, which clears the tty information for all members of the session on all nodes. `release_rmt_tty` is called by `disassociate_ctty()` if the tty is not local; the hook calls `disassociate_ctty()` on the node where the tty is. `get_tty` is called in `proc_pid_stat()` in `fs/proc/array.c` to gather the foreground pgrp and device id for the task's controlling terminal. The hook `update_ctty_pgrp` is called by `tiocspgrp()`, in `drivers/char/tty_io.c` and can be used by the CPM to inform all members of the old pgrp that they are no longer in the controlling terminal foreground pgrp and to inform the new pgrp members as well. Distributed knowledge of which pgrp is the foreground pgrp is important for correct behavior in the situation when the controlling terminal node crashes. `Sys_vhangup()`, in `fs/open.c`, has a call to `rmt_vhangup()` if `tty` is not set (if there is a remote tty, the CPM can call `sys_vhangup()` on that node). In `drivers/char/tty_io.c`, `tiosctty()` and `tty_open()` call the hook `has_rmt_ctty` to determine if the process already has a controlling terminal that is remote. Also in `drivers/char/tty_io.c`, the `tty_open()` function calls `rmt_tty_open` for opens of `/dev/tty` if the controlling terminal is remote. The `is_ignored()` function in `drivers/char/n_tty.c` calls `rmt_is_ignored` if it is called by an agent for a process that is actually running remotely. Finally, `rmt_tty_write_message` is called in `kernel/printk.c`, `tty_write_message()` if the tty it wants to write to is remote.

3.11 Process movement

```
int do_rexec(*char,*argv,*envp,
                    *regs,*reg);
void rexec_done();
int do_rfork(flags,stk,*regs,size,
             *ptid,*ctid,pid,*ret);
int do_migrate(*regs,signal,flags);
```

As mentioned in the goals, the hooks should allow for process movement at exec() time, at fork() time and during execution. Earlier versions of OpenSSI accomplished this via new system calls. The proposal here does not require any system calls although that is an option. For fork() and exec(), a hook is put in `do_fork()` and `do_execve()` respectively. Ops behind the hooks can determine if the operation should be done on another node. A load balancing algorithm can be consulted or the process could have been marked (eg. via a procfs file like `/proc/<pid>/goto`) for remote movement. An additional hook, `rexec_done` is provided so the CPM implementation can get control after the exec on the new node has completed but before returning to user mode, so that process setup can be completed and the original node can be informed that the remote execve() was successful.

A single hook is needed for process migration. The proposed mechanism is that via `/proc/<pid>/goto` or a load balancing subsystem, processes have `TIF_MIGPENDING` flag (added flag in flags field of `thread_info` structure) set if they should move. That flag is checked just before going back to user space, in `do_notify_resume()`, in `arch/xxx/kernel/signal.c` and calls the `do_migrate` hook. Checkpoint and restart can be invoked via the same hook (migrate to/from disk).

Determining if these hooks are sufficient to allow an implementation that satisfies the goals and requirements outlined earlier is best done by implementing a CPM using the hooks. The OpenSSI 3.0 CPM, which provides almost all the requirements, including optional ones, has been adapted to work via the hooks described above. Work to ensure that other CPM implementations can also be adapted needs to be done. The OpenSSI 3.0 CPM design is described in the next section.

4 Clusterproc Design for OpenSSI

In this section we describe a Cluster Process Management (CPM) implementation adapted from OpenSSI 2.0. It is part of a functional cluster which is a subset of OpenSSI. The subset does not have a cluster filesystem, a single root or single init. It does not have clusterwide device naming, a clusterwide IPC space or a cluster virtual ip. It does not have connection or process load balancing. All those capabilities will be subsequently added to this CPM implementation to produce OpenSSI 3.0.

To allow the CPM implementation to be part of a functional cluster, several other cluster components are needed. A loadable membership service is needed, together with an intra-node communication service layered on tcp sockets. To enable the full ptrace and remote controlling terminal support, a remote copy–to/from–user capability is needed. Also, a set of remote file ops is needed to allow access to remote controlling terminals. Finally, a couple of files are added to `/proc/<pid>` to provide and get information for CPM. Implementations of all needed capability are available and none require significant hooks. Like the clusterproc hooks, however, these hooks must be studied and included if they are general and allow for different implementations.

In this section we describe the process id and process tracking design, the module initialization, and per process private data. Then we describe how all the process relationships are managed clusterwide, followed by sections on `/proc` and process movement.

4.1 Process Ids and Process Tracking

As in OpenSSI, process ids are created by first having the local base kernel generate a locally unique id and then, using the hooks, adding the local node number in the higher order bits of the pid. This is the only pid the process will have and when the pid is no longer in use, the locally unique part is returned to the pool on the node it was generated on. The node who generated the process id (creation or origin node) is responsible for tracking if the process still exists and where it is currently running so operations on the process can be routed to the correct node and so ultimately the pid can be reused. If the origin node leaves the cluster, tracking is taken over by a designated node in the cluster (surrogate origin node) so processes are always findable without polling.

4.2 Clusterproc Module Initialization and Per Process Clusterproc Data Structure

The clusterproc module is loaded during the ramdisk processing although it could be done later. It assumes the membership, intra-node communication remote copy-in/copy-out and remote file ops modules are already loaded and registers with them. It sets up its data structures and installs the function pointers in the clusterproc op table. It also allocates and initializes clusterproc data structures for all existing processes, linking the structures into the task structure. After this initialization, each new process created will get a private clusterproc data structure via the `fork_alloc` hook.

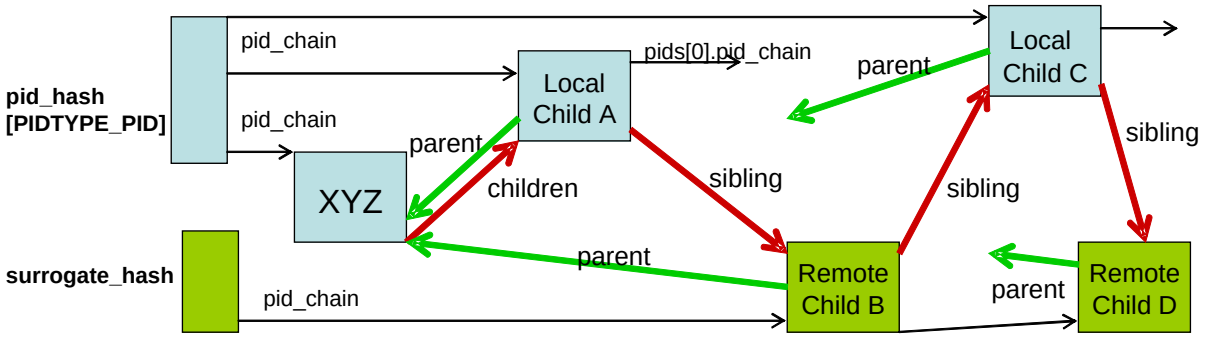

Figure 1: Parent xyz's execution node

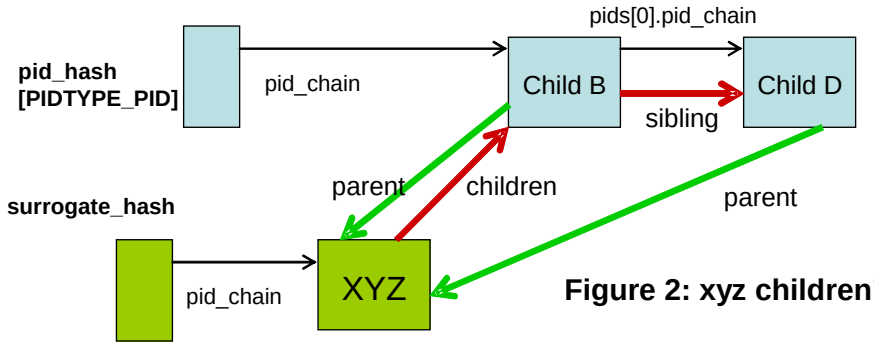

Figure 2: xyz children's execution node

4.3 Parent/Child/Ptrace Relationships

To minimize hooks and changes to the base Linux, the complete parent/child/sibling relationship is maintained at the current execution node of the parent, using surrogate task structures for any children that are not currently executing on that node. Surrogate task structures are just struct `task_struct` but are not hashed into the any of the base pid hashes and thus only visible to the base in the context of the parent/child relationship. Surrogate task structures have cached copies of the fields the parent will need to execute `sys_wait()` without having to poll remote children. The reap operation does involve an operation to the child execution node. The `update_parent` hook is used to maintain the caches of child information. For each node that has children but no parent, there is a surrogate task structure for the parent and a partial parent/child/sibling list. Surrogate task structures are hashed off a hash header private to the CPM module. Figure 1 shows how parent process XYZ is linked to his children on his execution node and Figure 2 shows the structures on a child node where XYZ is not executing.

Ptrace parent adds some complexity because a process's parent changes over time and `real_parent` can be different from `parent`. The `update_parent` hook is used to maintain all the proper links on all the nodes.

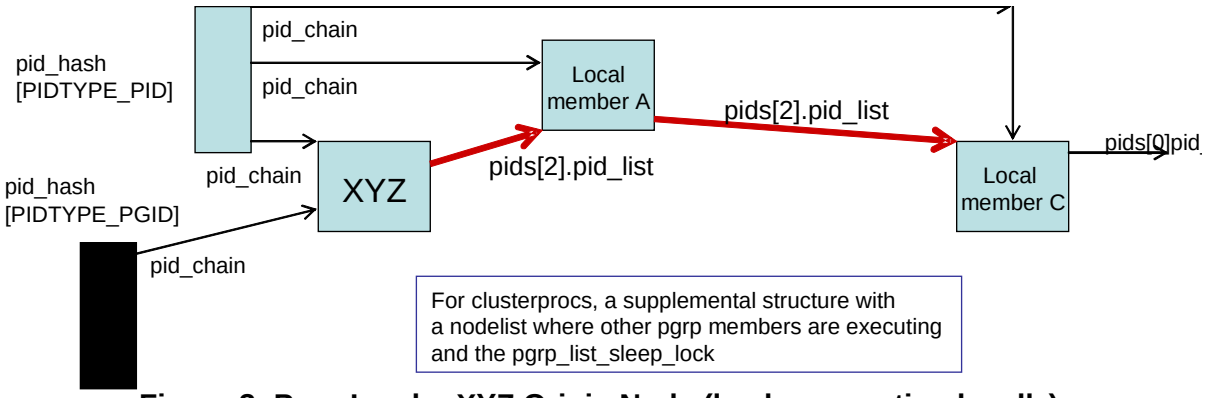

Figure 3: Pgrp Leader XYZ Origin Node (leader executing locally)

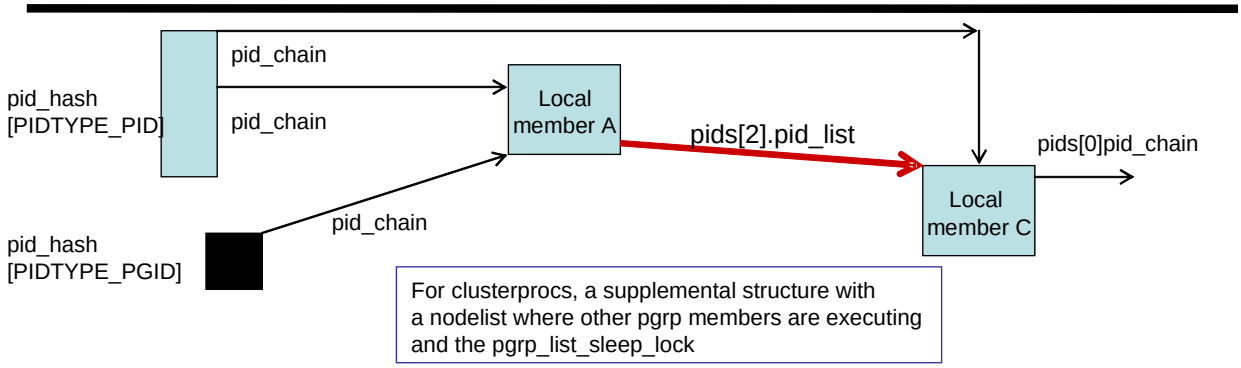

Figure 4: Pgrp Leader XYZ Origin Node (leader not executing locally)

4.4 Process Group and Session Relationships

With the process tracking described above, actions on an individual process is pretty straightforward. Actions on process groups and sessions are more complicated because the members may be scattered. For this CPM implementation, we keep a list of nodes where members are executing on the origin/surrogate origin node for the pid that is the name of the pgrp or session. On that origin node any local members are linked together as in the base but an additional structure is maintained that records the other nodes where members are on. This structure also has a sleep lock in it to make certain pgrp or session operations are atomic. Figures 3 and 4 shows the data structure layout on the origin node with and without the pgrp leader executing on that node. As with process tracking, this origin node role is assumed by the surrogate origin if the origin node fails and is thus not a single point of failure.

Operations on process groups are directed to the origin node (aka the list node). On that node the operation first gets the sleep lock. Then the operation can be done on any locally executing members by invoking the standard base code. Then the operation is sent to each node in the node list and the standard base operation is done for any members on that node.

A process group is orphan if no member has a parent in a different pgrp but with the same session id (sid). Linux needs to know if a process group is orphan to determine if processes

can stop (SIGTSTP, SIGTTIN, SIGTTOU). If a process group is orphan, they cannot. Linux also needs to know when a process group becomes orphan, because at that point any members that are stopped get SIGHUP and SIGCONT signals. A process exit might effect its own pgrp and the pgrp on all its children, which could involve looking at all the pgrp members (and their parents) of all the pgrps of all the exiting process's children. When all the pgrps and processes are distributed, this could be very expensive. The OpenSSI CPM, through the described hooks, has each pgrp list cache whether it is orphan or not, and if not, which nodes have processes contributing to its non-orphaness. Process exit can locally determine if it necessary to update the current process's pgrp list. Each child must be informed of the parents exit, but they can locally determine if they have to update the pgrp list orphan information. With a little additional information this mechanism can survive arbitrary node failures.

4.5 Controlling Terminal Management

In the OpenSSI cluster, the controlling terminal may be managed on a node other than that of the session leader or any of the processes using it. There is a relationship in that processes need to know who their controlling terminal is (and where it is) and the controlling terminal needs to know which session it is associated with and which process group is the foreground process group.

In the base linux, processes have a `tty` pointer to their controlling terminal. The `tty_struct` has a `pgrp` and a `session` field. In clusterproc, the base structures are maintained as is, with the pgrp and session fields in the tty structure and the tty pointer in the task's signal structure. The tty pointer will be maintained if the tty is local to the process. If the tty is not local, the clusterproc structure will have cttynode and cttydev fields to allow CPM code to determine if and where the controlling terminal is. To avoid hooks in some of the routines being executed at the controlling terminal node, svrprocs (agent kernel threads akin to nfsd's) doing opens, ioctls, reads and writes of devices at the controlling node will masquerade as the process doing the request (pid, pgrp, session, and tty). To avoid possible problems their masquerading might cause, svrprocs will not be hashed on the `pid_hash[PIDTYPE_PID]`.

4.6 Clusterwide `/proc`

Clusterwide `/proc` is accomplished by stacking a new pseudo filesystem (`cprocfs`) over an unmodified `/proc`. Hooks may be needed to do the stacking but will be modest. In addition, a couple of new files are added to `/proc/<pid>`—a `goto` file to facilitate process movement and a `where` file to display where the process is currently executing. The proposed semantics for `cprocfs` would be that:

- readdir presents all processes from all nodes and other proc files would either be an aggregation (sysvipc, uptime, net/unix, etc.) or would pass thru to the local `/proc`
- `cprocfs` function ships all ops on processes to the nodes where they are executing and then calls the `procfs` on those nodes;
- `cprocfs` inodes don't point at task structures but at small structures which have hints as to where the process is executing.
- `/proc/node/#` directories are redirected to the `/proc` on that node so one can access all the hardware information for any node.

- readdir of /proc/node/# only shows the processes executing on that node.

4.7 Process Movement

Via the hooks described earlier, the OpenSSI CPM system provides several forms of process movement, including a couple of forms of remote exec, an rfork and somewhat arbitrary process migration. In addition, these interfaces allow for transparent and programmatic checkpoint/restart.

The external interfaces to invoke process movement are library routines which in turn use the /proc/<pid>/goto interface to affect how standard system calls function. Writes to this file would take a buffer and length. To allow considerable flexibility in specifying the form of the movement and characteristics/functions to be performed as part of the movement, the buffer consists of a set of stanzas, each made up of a command and arguments. The initial set of commands is: rexec, rfork, migrate, checkpoint, restart, and context, but additional commands can be added. The arguments to rexec, rfork and migrate() are a node number. The argument to checkpoint and restart are a pathname for the checkpoint file. The context command indicates whether the process is to have the context of the node it is moving to or remain the way it was. Do_execve() and do_fork() have hooks which, if clusterproc is configured, will check the goto information that was stored off the clusterproc structure, and if appropriate, turn an exec into an rexec or a fork into an rfork.

The goto is also used to enable migrations. Besides saving the goto value, the write to the goto sets a new bit in the thread_info structure (TIF_MIGPENDING). Each time the process leaves the kernel to return to user space (did a system call or serviced an interrupt), the do_notify_resume() function is called if any of the flags in thread_info.flags are set (normally there are none set). The function do_notify_resume() now has a hook which will check for the TIF_MIGPENDING flag and if it is set, the process migrates itself. This hook only adds pathlength when any of the flags are set (TIF_SIGPENDING, etc.), which is very rarely.

OpenSSI currently has checkpoint/restart capability and this can be adapted to use the goto file and migration hook. Two forms of kernel-based checkpoint/restart have been done in OpenSSI. The first is transparent to the process, where the action is initiated by another process. The other is when the process is checkpoint/restart aware and is doing the checkpoint on itself. In that case, the process may wish to "know" when it is being restarted. To do that, we propose that the process open the /proc/self/goto file and attach a signal and signal handler to it. Then, when the process is restarted, the signal handler will be called. Checkpoint/restart are variants of migrate. The argument field to the goto file is a pathname. In the case of checkpoint, the TIF_MIGPENDING will be set and at the end of the next system call, the process will save its state in the filename specified. Another argument can determine whether the process is to continue or destroy at that point. Restart is done by first creating a new process and then doing the "restart" goto command to populate the new process with the saved image in the file which is specified as an argument.

A more extensive design document is available on the OpenSSI web site[9].

5 Summary

Process management in Linux is a complicated subsystem. There are several differ-

ent relationships—parent/child, process group, session, thread group, ptrace parent and controlling terminal (session and foreground pgrp). There are some intricate rules, like orphan process groups and `de_thread` with ptrace on the thread group leader. Making all this function in a completely single system way requires quite a few different hook functions, as defined above (some could be combined to reduce this number), but there is no performance impact and the footprint impact on the base kernel is very small (patch file touches 23 files with less than 500 lines of total changes, excluding the new `clusterproc.h` file).

References

[1] Popek, G., Walker, B. *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[2] Barak, A., Guday, S., Wheeler, R. *The MOSIX Distributed Operating System, Load Balancing for UNIX* volume 672 of Lecture Notes in Computer Science, Spinger-Verlag, 1993

[3] `http://www.openssi.org`

[4] `http://www.openmosix.org`

[5] `http://bproc.sourceforge.net`

[6] Valle'e, G., Morin, C., et.al., *Process migration based on gobelins distributed shared memory*, in proceedings of the workshop on Distributed Shared Memory (DSM'02) in CCGRID 2002, pg. 325–330, IEEE Computer Society, May 2002.

[7] Private communication

[8] `http://www.cassatt.com`

[9] `http://openssi.org/proc-hooks/proc-hooks.pdf`

[10] `http://openssi.org/ssi-intro.pdf`

Flow-based network accounting with Linux

Harald Welte
netfilter core team / hmw-consulting.de / Astaro AG
laforge@netfilter.org

Abstract

Many networking scenarios require some form of network accounting that goes beyond some simple packet and byte counters as available from the 'ifconfig' output.

When people want to do network accouting, the past and current Linux kernel didn't provide them with any reasonable mechanism for doing so.

Network accounting can generally be done in a number of different ways. The traditional way is to capture all packets by some userspace program. Capturing can be done via a number of mechanisms such as `PF_PACKET` sockets, `mmap()`ed `PF_PACKET`, `ipt_ULOG`, or `ip_queue`. This userspace program then analyzes the packets and aggregates the result into per-flow data structures.

Whatever mechanism used, this scheme has a fundamental performance limitation, since all packets need to be copied and analyzed by a userspace process.

The author has implemented a different approach, by which the accounting information is stored in the in-kernel connection tracking table of the `ip_conntrack` stateful firewall state machine. On all firewalls, that state table has to be kept anyways—the additional overhead introduced by accounting is minimal.

Once a connection is evicted from the state table, its accounting relevant data is transferred to userspace to a special accounting daemon for further processing, aggregation and finally storage in the accounting log/database.

1 Network accounting

Network accounting generally describes the process of counting and potentially summarizing metadata of network traffic. The kind of metadata is largely dependant on the particular application, but usually includes data such as numbers of packets, numbers of bytes, source and destination ip address.

There are many reasons for doing accounting of networking traffic, among them

- transfer volume or bandwisth based billing
- monitoring of network utilization, bandwidth distribution and link usage
- research, such as distribution of traffic among protocols, average packet size, ...

2 Existing accounting solutions for Linux

There are a number of existing packages to do network accounting with Linux. The follow-

ing subsections intend to give a short overview about the most commonly used ones.

2.1 nacctd

`nacctd` also known as `net-acct` is probably the oldest known tool for network accounting under Linux (also works on other Unix-like operating systems). The author of this paper has used `nacctd` as an accounting tool as early as 1995. It was originally developed by Ulrich Callmeier, but apparently abandoned later on. The development seems to have continued in multiple branches, one of them being the netacct-mysql[1] branch, currently at version 0.79rc2.

Its principle of operation is to use an `AF_PACKET` socket via `libpcap` in order to capture copies of all packets on configurable network interfaces. It then does TCP/IP header parsing on each packet. Summary information such as port numbers, IP addresses, number of bytes are then stored in an internal table for aggregation of successive packets of the same flow. The table entries are evicted and stored in a human-readable ASCII file. Patches exist for sending information directly into SQL databases, or saving data in machine-readable data format.

As a pcap-based solution, it suffers from the performance penalty of copying every full packet to userspace. As a packet-based solution, it suffers from the penalty of having to interpret every single packet.

2.2 ipt_LOG based

The Linux packet filtering subsystem iptables offers a way to log policy violations via the kernel message ring buffer. This mechanism is called `ipt_LOG` (or `LOG target`). Such messages are then further processed by `klogd` and `syslogd`, which put them into one or multiple system log files.

As `ipt_LOG` was designed for logging policy violations and not for accounting, its overhead is significant. Every packet needs to be interpreted in-kernel, then printed in ASCII format to the kernel message ring buffer, then copied from klogd to syslogd, and again copied into a text file. Even worse, most syslog installations are configured to write kernel log messages synchronously to disk, avoiding the usual write buffering of the block I/O layer and disk subsystem.

To sum up and anlyze the data, often custom perl scripts are used. Those perl scripts have to parse the LOG lines, build up a table of flows, add the packet size fields and finally export the data in the desired format. Due to the inefficient storage format, performance is again wasted at analyzation time.

2.3 ipt_ULOG based (ulogd, ulog-acctd)

The iptables `ULOG target` is a more efficient version of the `LOG target` described above. Instead of copying ascii messages via the kernel ring buffer, it can be configured to only copies the header of each packet, and send those copies in large batches. A special userspace process, normally ulogd, receives those partial packet copies and does further interpretation.

`ulogd`[2] is intended for logging of security violations and thus resembles the functionality of LOG. it creates one logfile entry per packet. It

[1] `http://netacct-mysql.gabrovo.com`

[2] `http://gnumonks.org/projects/ulogd`

supports logging in many formats, such as SQL databases or PCAP format.

`ulog-acctd`[3] is a hybrid between `ulogd` and `nacctd`. It replaces the `nacctd` libpcap/PF_PACKET based capture with the more efficient ULOG mechanism.

Compared to `ipt_LOG`, `ipt_ULOG` reduces the amount of copied data and required kernel/userspace context switches and thus improves performance. However, the whole mechanism is still intended for logging of security violations. Use for accounting is out of its design.

2.4 iptables based (ipac-ng)

Every packet filtering rule in the Linux packet filter (`iptables`, or even its predecessor `ipchains`) has two counters: number of packets and number of bytes matching this particular rule.

By carefully placing rules with no target (so-called *fallthrough*) rules in the packetfilter ruleset, one can implement an accounting setup, i.e., one rule per customer.

A number of tools exist to parse the iptables command output and summarized the counters. The most commonly used package is `ipac-ng`[4]. It supports advanced features such as storing accounting data in SQL databases.

The approach works quite efficiently for small installations (i.e., small number of accounting rules). Therefore, the accounting granularity can only be very low. One counter for each single port number at any given ip address is certainly not applicable.

[3] `http://alioth.debian.org/projects/pkg-ulog-acctd/`

[4] `http://sourceforge.net/projects/ipac-ng/`

2.5 ipt_ACCOUNT (iptaccount)

`ipt_ACCOUNT`[5] is a special-purpose iptables target developed by Intra2net AG and available from the netfilter project patch-o-matic-ng repository. It requires kernel patching and is not included in the mainline kernel.

`ipt_ACCOUNT` keeps byte counters per IP address in a given subnet, up to a '/8' network. Those counters can be read via a special `iptaccount` commandline tool.

Being limited to local network segments up to '/8' size, and only having per-ip granularity are two limiteations that defeat `ipt_ACCOUNT` as a generich accounting mechainism. It's highly-optimized, but also special-purpose.

2.6 ntop (including PF_RING)

`ntop`[6] is a network traffic probe to show network usage. It uses `libpcap` to capture the packets, and then aggregates flows in userspace. On a fundamental level it's therefore similar to what `nacctd` does.

From the ntop project, there's also `nProbe`, a network traffic probe that exports flow based information in Cisco NETFLOW v5/v9 format. It also contains support for the upcoming IETF IPFIX[7] format.

To increase performance of the probe, the author (Luca Deri) has implemented `PF_RING`[8], a new zero-copy mmap()ed implementation for

[5] `http://www.intra2net.com/opensource/ipt_account/`

[6] `http://www.ntop.org/ntop.html`

[7] IP Flow Information Export `http://www.ietf.org/html.charters/ipfix-charter.html`

[8] `http://www.ntop.org/PF_RING.html`

packet capture. There is a libpcap compatibility layer on top, so any pcap-using application can benefit from `PF_RING`.

`PF_RING` is a major performance improvement, please look at the documentation and the paper published by Luca Deri.

However, `ntop` / `nProbe` / `PF_RING` are all packet-based accounting solutions. Every packet needs to be analyzed by some userspace process—even if there is no copying involved. Due to `PF_RING` optimiziation, it is probably as efficient as this approach can get.

3 New ip_conntrack based accounting

The fundamental idea is to (ab)use the connection tracking subsystem of the Linux 2.4.x / 2.6.x kernel for accounting purposes. There are several reasons why this is a good fit:

- It already keeps per-connection state information. Extending this information to contain a set of counters is easy.
- Lots of routers/firewalls are already running it, and therefore paying its performance penalty for security reasons. Bumping a couple of counters will introduce very little additional penalty.
- There was already an (out-of-tree) system to dump connection tracking information to userspace, called ctnetlink.

So given that a particular machine was already running `ip_conntrack`, adding flow based acconting to it comes almost for free. I do not advocate the use of `ip_conntrack` merely for accounting, since that would be again a waste of performance.

3.1 ip_conntrack_acct

`ip_conntrack_acct` is how the in-kernel `ip_conntrack` counters are called. There is a set of four counters: numbers of packets and bytes for original and reply direction of a given connection.

If you configure a recent (>= 2.6.9) kernel, it will prompt you for `CONFIG_IP_NF_CT_ACCT`. By enabling this configuration option, the per-connection counters will be added, and the accounting code will be compiled in.

However, there is still no efficient means of reading out those counters. They can be accessed via *cat /proc/net/ip_conntrack*, but that's not a real solution. The kernel iterates over all connections and ASCII-formats the data. Also, it is a polling-based mechanism. If the polling interval is too short, connections might get evicted from the state table before their final counters are being read. If the interval is too small, performance will suffer.

To counter this problem, a combination of conntrack notifiers and ctnetlink is being used.

3.2 conntrack notifiers

Conntrack notifiers use the core kernel notifier infrastructure (`struct notifier_block`) to notify other parts of the kernel about connection tracking events. Such events include creation, deletion and modification of connection tracking entries.

The `conntrack notifiers` can help us overcome the polling architecture. If we'd only listen to *conntrack delete* events, we would always get the byte and packet counters at the end of a connection.

However, the events are in-kernel events and therefore not directly suitable for an accounting application to be run in userspace.

3.3 ctnetlink

`ctnetlink` (short form for conntrack netlink) is a mechanism for passing connection tracking state information between kernel and userspace, originally developed by Jay Schulist and Harald Welte. As the name implies, it uses Linux `AF_NETLINK` sockets as its underlying communication facility.

The focus of `ctnetlink` is to selectively read or dump entries from the connection tracking table to userspace. It also allows userspace processes to delete and create conntrack entries as well as *conntrack expectations*.

The initial nature of `ctnetlink` is therefore again polling-based. An userspace process sends a request for certain information, the kernel responds with the requested information.

By combining `conntrack notifiers` with `ctnetlink`, it is possible to register a notifier handler that in turn sends `ctnetlink` event messages down the `AF_NETLINK` socket.

A userspace process can now listen for such *DELETE* event messages at the socket, and put the counters into its accounting storage.

There are still some shortcomings inherent to that *DELETE* event scheme: We only know the amount of traffic after the connection is over. If a connection lasts for a long time (let's say days, weeks), then it is impossible to use this form of accounting for any kind of quota-based billing, where the user would be informed (or disconnected, traffic shaped, whatever) when he exceeds his quota. Also, the conntrack entry does not contain information about when the connection started—only the timestamp of the end-of-connection is known.

To overcome limitation number one, the accounting process can use a combined event and polling scheme. The granularity of accounting can therefore be configured by the polling interval, and a compromise between performance and accuracy can be made.

To overcome the second limitation, the accounting process can also listen for *NEW* event messages. By correlating the *NEW* and *DELETE* messages of a connection, accounting datasets containign start and end of connection can be built.

3.4 ulogd2

As described earlier in this paper, `ulogd` is a userspace packet filter logging daemon that is already used for packet-based accounting, even if it isn't the best fit.

`ulogd2`, also developed by the author of this paper, takes logging beyond per-packet based information, but also includes support for per-connection or per-flow based data.

Instead of supporting only `ipt_ULOG` input, a number of interpreter and output plugins, `ulogd2` supports a concept called *plugin stacks*. Multiple stacks can exist within one deamon. Any such stack consists out of plugins. A plugin can be a source, sink or filter.

Sources acquire per-packet or per-connection data from `ipt_ULOG` or `ip_contnrack_acct`.

Filters allow the user to filter or aggregate information. Filtering is requird, since there is no way to filter the ctnetlink event messages within the kernel. Either the functionality is enabled or not. Multiple connections can be aggregated to a larger, encompassing flow. Packets could be aggregated to flows (like `nacctd`), and flows can be aggregated to even larger flows.

Sink plugins store the resulting data to some form of non-volatile storage, such as SQL databases, binary or ascii files. Another sink is a NETFLOW or IPFIX sink, exporting information in industy-standard format for flow based accounting.

3.5 Status of implementation

`ip_conntrack_acct` is already in the kernel since 2.6.9.

`ctnetlink` and the `conntrack event notifiers` are considered stable and will be submitted for mainline inclusion soon. Both are available from the patch-o-matic-ng repository of the netfilter project.

At the time of writing of this paper, `ulogd2` development was not yet finished. However, the ctnetlink event messages can already be dumped by the use of the "conntrack" userspace program, available from the netfilter project.

The "conntrack" prorgram can listen to the netlink event socket and dump the information in human-readable form (one ASCII line per ctnetlink message) to stdout. Custom accounting solutions can read this information from stdin, parse and process it according to their needs.

4 Summary

Despite the large number of available accounting tools, the author is confident that inventing yet another one is worthwhile.

Many existing implementations suffer from performance issues by design. Most of them are very special-purpose. nProbe/ntop together with `PF_RING` are probably the most universal and efficient solution for any accounting problem.

Still, the new `ip_conntrack_acct`, `ctnetlink` based mechanism described in this paper has a clear performance advantage if you want to do acconting on your Linux-based stateful packetfilter—which is a common case. The firewall is suposed to be at the edge of your network, exactly where you usually do accounting of ingress and/or egress traffic.

Introduction to the InfiniBand Core Software

Bob Woodruff
Intel Corporation
robert.j.woodruff@intel.com

Sean Hefty
Intel Corporation
sean.hefty@intel.com

Roland Dreier
Topspin Communications
roland@topspin.com

Hal Rosenstock
Voltaire Corporation
halr@coltaire.com

Abstract

InfiniBand support was added to the kernel in 2.6.11. In this paper, we describe the various modules and interfaces of the InfiniBand core software and provide examples of how and when to use them. The core software consists of the host channel adapter (HCA) driver and a mid-layer that abstracts the InfiniBand device implementation specifics and presents a consistent interface to upper level protocols, such as IP over IB, sockets direct protocol, and the InfiniBand storage protocols. The InfiniBand core software is logically grouped into 5 major areas: HCA resource management, memory management, connection management, work request and completion event processing, and subnet administration. Physically, the core software is currently contained within 6 kernel modules. These include the Mellanox HCA driver, ib_mthca.ko, the core verbs module, ib_core.ko, the connection manager, ib_cm.ko, and the subnet administration support modules, ib_sa.ko, ib_mad.ko, ib_umad.ko. We will also discuss the additional modules that are under development to export the core software interfaces to userspace and allow safe direct access to InfiniBand hardware from userspace.

1 Introduction

This paper describes the core software components of the InfiniBand software that was included in the linux 2.6.11 kernel. The reader is referred to the architectural diagram and foils in the slide set that was provided as part of the paper's presentation at the Ottawa Linux Symposium. It is also assumed that the reader has read at least chapters 3, 10, and 11 of InfiniBand Architecture Specification [IBTA] and is familiar with the concepts and terminology of the InfiniBand Architecture. The goal of the paper is not to educate people on the InfiniBand Architecture, but rather to introduce the reader to the APIs and code that implements the InfiniBand Architecture support in Linux. Note that the InfiniBand code that is in the kernel has been written to comply with the InfiniBand 1.1 specification with some 1.2 extensions, but it is important to note that the code is not yet completely 1.2 compliant.

The InfiniBand code is located in the kernel tree under `linux-2.6.11/drivers/infiniband`. The reader is encouraged to read the code and header files in the kernel tree. Several pieces of the InfiniBand stack that are in the kernel contain good examples of how to use

the routines of the core software described in this paper. Another good source of information can be found at the www.openib.org website. This is where the code is developed prior to being submitted to the linux kernel mailing list (lkml) for kernel inclusion. There are several frequently asked question documents plus email lists `<openib-general@openib.org>`. where people can ask questions or submit patches to the InfinBand code.

The remainder of the paper provides a high level overview of the mid-layer routines and provides some examples of their usage. It is targeted at someone that might want to write a kernel module that uses the mid-layer or someone interested in how it is used. The paper is divided into several sections that cover driver initialization and exit, resource management, memory management, subnet administration from the viewpoint of an upper level protocol developer, connection management, and work request and completion event processing. Finally, the paper will present a section on the user-mode infrastructure and how one can safely use the InfiniBand resource directly from userspace applications.

2 Driver initialization and exit

Before using InfiniBand resources, kernel clients must register with the mid-layer. This also provides the way, via callbacks, for the client to discover the available InfiniBand devices that are present in the system. To register with the InfiniBand mid-layer, a client calls the `ib_register_client` routine. The routine takes as a parameter a pointer to a `ib_client` structure, as defined in `linux-2.6.11/drivers/infiniband/include/ib_verbs.h`. The structure takes a pointer to the client's name, plus two function pointers to callback routines that are invoked when an InfiniBand device is added or removed from the system. Below is some sample code that shows how this routine is called:

```
static void my_add_device(
   struct ib_device *device);

static void my_remove_device(
   struct ib_device *device);

static struct ib_client my_client = {
 .name   = "my_name",
 .add    = my_add_device,
 .remove = my_remove_device
};
static int __init my_init(void)
{
 int ret;

 ret = ib_register_client(
       &my_client);
    if (ret)
       printk(KERN_ERR
"my ib_register_client failed\n");
 return ret;
}
static void __exit my_cleanup(void)
{
   ib_unregister_client(
             &my_client);
}
module_init(my_init);
module_exit(my_cleanup);
```

3 InfiniBand resource management

3.1 Miscellaneous Query functions

The mid-layer provides routines that allow a client to query or modify information about the various InfiniBand resources.

```
ib_query_device
ib_query_port
ib_query_gid
ib_query_pkey
```

```
ib_modify_device
ib_modify_port
```

The `ib_query_device` routine allows a client to retrieve attributes for a given hardware device. The returned `device_attr` structure contains device specific capabilities and limitations, such as the maximum sizes for queue pairs, completion queues, scatter gather entries, etc., and is used when configuring queue pairs and establishing connections.

The `ib_query_port` routine returns information that is needed by the client, such as the state of the port (Active or not), the local identifier (LID) assigned to the port by the subnet manager, the Maximum Transfer Unit (MTU), the LID of the subnet manager, needed for sending SA queries, the partition table length, and the maximum message size.

The `ib_query_pkey` routine allows the client to retrieve the partition keys for a port. Typically, the subnet manager only sets one pkey for the entire subnet, which is the default pkey.

The `ib_modify_device` and `ib_modify_port` routines allow some of the device or port attributes to be modified. Most ULPs do not need to modify any of the port or device attributes. One exception to this would be the communication manager, which sets a bit in the port capabilities mask to indicate the presence of a CM.

Additional query and modify routines are discussed in later sections when a particular resource, such as queue pairs or completion queues, are discussed.

3.2 Protection Domains

Protection domains are a first level of access control provided by InfiniBand. Protection domains are allocated by the client and associated with subsequent InfiniBand resources, such as queue pairs, or memory regions.

Protection domains allow a client to associate multiple resources, such as queue pairs and memory regions, within a domain of trust. The client can then grant access rights for sending/receiving data within the protection domain to others that are on the Infinband fabric.

To allocate a protection domain, clients call the `ib_alloc_pd` routine. The routine takes and pointer to the device structure that was returned when the driver was called back after registering with the mid-layer. For example:

```
my_pd = ib_alloc_pd(device);
```

Once a PD has been allocated, it is used in subsequent calls to allocate other resources, such as creating address handles or queue pairs.

To free a protection domain, the client calls `ib_dealloc_pd`, which is normally only done at driver unload time after all of the other resources associated with the PD have been freed.

```
ib_dealloc_pd(my_pd);
```

3.3 Types of communication in InfiniBand

Several types of communication between end-points are defined by the InfiniBand architecture specification [IBTA]. These include reliable-connected, unreliable-connected, reliable-datagram, and unreliable datagrams. Most clients today only use either unreliable datagrams or reliable connected communications. An analogy in the IP network stack would be that unreliable datagrams are analogous to UDP packets, while a reliable-connected queue pairs provide a

connection-oriented type of communication, similar to TCP. But InfiniBand communication is packet-based, rather than stream oriented.

3.4 Address handles

When a client wants to communicate via unreliable datagrams, the client needs to create an address handle that contains the information needed to send packets.

To create an address handle the client calls the routine `ib_create_ah()`. An example code fragment is shown below:

```
struct ib_ah_attr   ah_attr;
struct ib_ah        *remote_ah;

memset(&ah_attr, 0, sizeof ah_attr);
ah_attr.dlid     = remote_lid;
ah_attr.sl       = service_level;
ah_attr.port_num = port->port_num;

remote_ah = ib_create_ah(pd, &ah_attr);
```

In the above example, the pd is the protection domain, the `remote_lid` and `service_level` are obtained from an SA path record query, and the `port_num` was returned in the device structure through the `ib_register_client` callback. Another way to get the `remote_lid` and `service_level` information is from a packet that was received from a remote node.

There are also core verb APIs for destroying the address handles and for retrieving and modifying the address handle attributes.

```
ib_destroy_ah
ib_query_ah
ib_modify_ah
```

Some example code that calls `ib_create_ah` to create an address handle for a multicast group can be found in the IPoIB network driver for InfiniBand, and is located in `linux-2.6.11/drivers/infiniband/ulp/ipoib`.

3.5 Queue Pairs and Completion Queue Allocation

All data communicated over InfiniBand is done via queue pairs. Queue pairs (QPs) contain a send queue, for sending outbound messages and requesting RDMA and atomic operations, and a receive queue for receiving incoming messages or immediate data. Furthermore, completion queues (CQs) must be allocated and associated with a queue pair, and are used to receive completion notifications and events.

Queue pairs and completion queues are allocated by calling the `ib_create_qp` and `ib_create_cq` routines, respectively.

The following sample code allocates separate completion queues to handle send and receive completions, and then allocates a queue pair associated with the two CQs.

```
send_cq = ib_create_cq(device,
          my_cq_event_handler,
          NULL,
          my_context,
          my_send_cq_size);
recv_cq = ib_create_cq(device,
          my_cq_event_handler,
          NULL,
          my_context,
          my_recv_cq_size);

init_attr->cap.max_send_wr  = send_cq_size;
init_attr->cap.max_recv_wr  = recv_cq_size;
init_attr->cap.max_send_sge = LIMIT_SG_SEND;
init_attr->cap.max_recv_sge = LIMIT_SG_RECV;

init_attr->send_cq          = send_cq;
init_attr->recv_cq          = recv_cq;
init_attr->sq_sig_type      = IB_SIGNAL_REQ_WR;
init_attr->qp_type          = IB_QPT_RC;
init_attr->event_handler = my_qp_event_handler;

my_qp = ib_create_qp(pd, init_attr);
```

After a queue pair is created, it can be connected to a remote QP to establish a connection. This is done using the QP modify routine and the communication manager helper functions described in a later section.

There are also mid-layer routines that allow destruction and release of QPs and CQs, along

with the routines to query and modify the queue pair attributes and states. These additional core QP and CQ support routines are as follows:

```
ib_modify_qp
ib_query_qp
ib_destroy_qp
ib_destroy_cq
ib_resize_cq
```

Note that `ib_resize_cq` is not currently implemented in the mthca driver.

An example of kernel code that allocates QPs and CQs for reliable-connected style of communication is the SDP driver [SDP]. It can be found in the subversion tree at openib.org, and will be submitted for kernel inclusion at some point in the near future.

4 InfiniBand memory management

Before a client can transfer data across InfiniBand, it needs to register the corresponding memory buffer with the InfiniBand HCA. The InfiniBand mid-layer assumes that the kernel or ULP has already pinned the pages and has translated the virtual address to a Linux DMA address, i.e., a bus address that can be used by the HCA. For example, the driver could call `get_user_pages` and then `dma_map_sg` to get the DMA address.

Memory registration can be done in a couple of different ways. For operations that do not have a scatter/gather list of pages, there is a memory region that can be used that has all of physical memory pre-registered. This can be thought of as getting access to the "Reserved L_key" that is defined in the InfiniBand verbs extensions [IBTA].

To get the memory region structure that has the keys that are needed for data transfers, the client calls the `ib_get_dma_mr` routine, for example:

```
mr = ib_get_dma_mr(my_pd,
     IB_ACCESS_LOCAL_WRITE);
```

If the client has a list of pages that are not physically contiguous but want to be virtually contiguous with respect to the DMA operation, i.e., scatter/gather, the client can call the `ib_reg_phys_mr` routine. For example,

```
*iova = &my_buffer1;

buffer_list[0].addr = dma_addr_buffer1;
buffer_list[0].size = buffer1_size;
buffer_list[1].addr = dma_addr_buffer2;
buffer_list[1].size = buffer2_size;

mr = ib_reg_phys_mr(my_pd,
     buffer_list,
     2,
     IB_ACCESS_LOCAL_WRITE |
     IB_ACCESS_REMOTE_READ |
     IB_ACCESS_REMOTE_WRITE,
     iova);
```

The `mr` structure that is returned contains the necessary local and remote keys, lkey and rkey, needed for sending/receiving messages and performing RDMA operations. For example, the combination of the returned iova and the rkey are used by a remote node for RDMA operations.

Once a client has completed all data transfers to a memory region, e.g., the DMA is completed, the client can release to the resources back to the HCA using the `ib_dereg_mr` routine, for example:

```
ib_dereg_mr(mr);
```

There is also a verb, `ib_rereg_phys_mr` that allows the client to modify the attributes of

a given memory region. This is similar to doing a de-register followed by a re-register but where possible the HCA reuses the same resources rather than deallocating and then reallocating new ones.

```
status = ib_rereg_phys_mr(mr,
         mr_rereg_mask,
         my_pd,
         buffer_list,
         num_phys_buf,
         mr_access_flags,
        iova_start);
```

There is also a set of routines that allow a technique called fast memory registration. Fast Memory Registration, or FMR, was implemented to allow the re-use of memory regions and to reduce the overhead involved in registration and deregistration with the HCAs. Using the technique of FMR, the client typically allocates a pool of FMRs during initialization. Then when it needs to register memory with the HCA, the client calls a routine that maps the pages using one of the pre-allocated FMRs. Once the DMA is complete, the client can unmap the pages from the FMR and recycle the memory region and use it for another DMA operation. The following routines are used to allocate, map, unmap, and deallocate FMRs.

```
ib_alloc_fmr
ib_unmap_fmr
ib_map_phys_fmr
ib_dealloc_fmr
```

An example of coding using FMRs can be found in the SDP [SDP] driver available at `openib.org`.

NOTE: These FMRs are a Mellanox specific implementation and are NOT the same as the FMRs as defined by the 1.2 InfiniBand verbs extensions [IBTA]. The FMRs that are implemented are based on the Mellanox FMRs that predate the 1.2 specification and so the developers deviated slightly from the InfiniBand specification in this area.

InfiniBand also has the concept of memory windows [IBTA]. Memory windows are a way to bind a set of virtual addresses and attributes to a memory regions by posting an operation to a send queue. It was thought that people might want this dynamic binding/unbinding intermixed with their work request flow. However, it is currently not used, primarily because of poor H/W performance in the existing HCA, and thus is not implemented in the mthca driver in Linux.

However, there are APIs defined in the midlayer for memory windows for when it is implemented in mthca or some future HCA driver. These are as follows:

```
ib_alloc_mw
ib_dealloc_mw
```

5 InfiniBand subnet administration

Communication with subnet administration(SA) is often needed to obtain information for establishing communication or setting up multicast groups. This is accomplished by sending management datagram (MAD) packets to the SA through InfiniBand special QP 1 [IBTA]. The low level routines that are needed to send/receive MADs along with the critical data structures are defined in `linux-2.6.11/drivers/infiniband/include/ib_mad.h`.

Several helper functions have been implemented for obtaining path record information or joining multicast groups. These relieve

most clients from having to understand the low level MAD routines. Subnet administration APIs and data structures are located in `linux-2.6.11/drivers/infiniband/include/ib_sa.h` and the following sections discuss their usage.

5.1 Path Record Queries

To establish connections, certain information is needed, such as the source/destination LIDs, service level, MTU, etc. This information is found in a data structure known as a path record, which contains all relevant information of a path between a source and destination. Path records are managed by the InfiniBand subnet administrator(SA). To obtain a path record, the client can use the helper function:

```
ib_sa_path_rec_get
```

This function takes the device structure, returned by the register routine callback, the local InfiniBand port to use for the query, a timeout value, which is the time to wait before giving up on the query, and two masks, `comp_mask` and `gfp_mask`. The `comp_mask` specifies the components of the `ib_sa_path_rec` to perform the query with. The `gfp_mask` is the mask used for internal memory allocations, e.g., the ones passed to kmalloc, `GFP_KERNEL`, `GFP_USER`, `GFP_ATOMIC`, `GFP_USER`. The `**query` parameter is a returned identifier of the query that can be used to cancel it, if needed. For example, given a source and destination InfiniBand global identifier (sgid/dgid) and the partition key, here is an example query call taken from the SDP [SDP] code.

```
query_id = ib_sa_path_rec_get(
     info->ca,
     info->port,
     &info->path,
     (IB_SA_PATH_REC_DGID |
      IB_SA_PATH_REC_SGID |
      IB_SA_PATH_REC_PKEY |
    IB_SA_PATH_REC_NUMB_PATH),
      info->sa_time,
      GFP_KERNEL,
      sdp_link_path_rec_done,
      info,
      &info->query);

if (result < 0) {
    sdp_dbg_warn(NULL,
"Error <%d> restarting path query",
               result);
}
```

In the above example, when the query completes, or times-out, the client is called back at the provided callback routine, `sdp_link_path_rec_done`. If the query succeeds, the path record(s) information requested is returned along with the context value that was provided with the query.

If the query times out, the client can retry the request by calling the routine again.

Note that in the above example, the caller must provide the DGID, SGID, and PKEY in the `info->path` structure, In the SDP example, the `info->path.dgid`, `info->path.sgid`, and `info->path.pkey` are set in the SDP routine `do_link_path_lookup`.

5.2 Cancelling SA Queries

If the client wishes to cancel an SA query, the client uses the returned `**query` parameter and query function return value (query id), e.g.,

```
ib_sa_cancel_query(
     query_id,
     query);
```

5.3 Multicast Groups

Multicast groups are administered by the subnet administrator/subnet manager, which configure InfiniBand switches for the multicast group. To participate in a multicast group, a client sends a message to the subnet administrator to join the group. The APIs used to do this are shown below:

```
ib_sa_mcmember_rec_set
ib_sa_mcmember_rec_delete
ib_sa_mcmember_rec_query
```

The `ib_sa_mcmember_rec_set` routine is used to create and/or join the multicast group and the `ib_sa_mcmember_rec_delete` routine is used to leave a multicast group. The `ib_sa_mcmember_rec_query` routine can be called get information on available multicast groups. After joining the multicast group, the client must attach a queue pair to the group to allow sending and receiving multicast messages. Attaching/detaching queue pairs from multicast groups can be done using the API shown below:

```
ib_attach_mcast
ib_detach_mcast
```

The gid and lid in these routines are the multicast gid(mgid) and multicast lid (mlid) of the group. An example of using the multicast routines can be found in the IP over IB code located in `linux-2.6.11/drivers/infiniband/ulp/ipoib`.

5.4 MAD routines

Most upper level protocols do not need to send and receive InfiniBand management datagrams (MADs) directly. For the few operations that require communication with the subnet manager/subnet administrator, such as path record queries or joining multicast groups, helper functions are provided, as discussed in an earlier section.

However, for some modules of the mid-layer itself, such as the communications manager, or for developers wanting to implement management agents using the InfiniBand special queue pairs, MADs may need to be sent and received directly. An example might be someone that wanted to tunnel IPMI [IPMI] or SNMP [SNMP] over InfiniBand for remote server management. Another example is handling some vendor-specific MADs that are implemented by a specific InfiniBand vendor. The MAD routines are defined in `linux-2.6.11/drivers/infiniband/include/ib_mad.h`.

Before being allowed to send or receive MADs, MAD layer clients must register an agent with the MAD layer using the following routines. The `ib_register_mad_snoop` routine can be used to snoop MADs, which is useful for debugging.

```
ib_register_mad_agent
ib_unregister_mad_agent
ib_register_mad_snoop
```

After registering with the MAD layer, the MAD client sends and receives MADs using the following routines.

```
ib_post_send_mad
ib_coalesce_recv_mad
ib_free_recv_mad
ib_cancel_mad
ib_redirect_mad_qp
ib_process_mad_wc
```

The ib_post_send_mad routine allows the client to queue a MAD to be sent. After a MAD is received, it is given to a client through their receive handler specified when registering. When a client is done processing an incoming MAD, it frees the MAD buffer by calling ib_free_recv_mad. As one would expect, the ib_cancel_mad routine is used to cancel an outstanding MAD request.

ib_coalesce_recv_mad is a place-holder routine related to the handling of MAD segmentation and reassembly. It will copy received MAD segments into a single data buffer, and will be implemented once the InfiniBand reliable-multi-packet-protocol (RMPP) support is added.

Similarly, the routine ib_redirect_mad_qp and the routine ib_process_mad_wc are place holders for supporting QP redirection, but are not currently implemented. QP redirection permits a management agent to send and receive MADs on a QP other than the GSI QP (QP 1). As an example, a protocol which was data intensive could use QP redirection to send and receive management datagrams on their own QP, avoiding contention with other users of the GSI QP, such as connection management or SA queries. In this case, the client can re-redirect a particular InfiniBand management class to a dedicated QP using the ib_redirect_mad_qp routine. The ib_process_mad_wc routine would then be used to complete or continue processing a previously started MAD request on the redirected QP.

6 InfiniBand connection management

The mid-layer provides several helper functions to assist with establishing connections. These are defined in the header file, linux-2.6.11/drivers/infiniband/include/ib_cm.h Before initiating a connection request, the client must first register a callback function with the mid-layer for connection events.

```
ib_create_cm_id
ib_destroy_cm_id
```

The ib_create_id routine creates a communication id and registers a callback handler for connection events. The ib_destroy_cm_id routine can be used to free the communication id and de-register the communication callback routine after the client is finished using their connections.

The communication manager implements a client/server style of connection establishment, using a three-way handshake between the client and server. To establish a connection, the server side listens for incoming connection requests. Clients connect to this server by sending a connection request. After receiving the connection request, the server will send a connection response or reject message back to the client. A client completes the connection setup by sending a ready to use (RTU) message back to the server. The following routines are used to accomplish this:

```
ib_cm_listen
ib_send_cm_req
ib_send_cm_rep
ib_send_cm_rtu
ib_send_cm_rej
ib_send_cm_mra
ib_cm_establish
```

The communication manager is responsible for retrying and timing out connection requests. Clients receiving a connection request may require more time to respond to a request than the

timeout used by the sending client. For example, a client tries to connect to a server that provides access to disk storage array. The server may require several seconds to ready the drives before responding to the client. To prevent the client from timing out its connection request, the server would use the `ib_send_cm_mra` routine to send a message received acknowledged (MRA) to notify the client that the request was received and that a longer timeout is necessary.

After a client sends the RTU message, it can begin transferring data on the connection. However, since CM messages are unreliable, the RTU may be delayed or lost. In such cases, receiving a message on the connection notifies the server that the connection has been established. In order for the CM to properly track the connection state, the server calls `ib_cm_establish` to notify the CM that the connection is now active.

Once a client is finished with a connection, it can disconnect using the disconnect request routine (`ib_send_cm_dreq`) shown below. The recipient of a disconnect request sends a disconnect reply.

```
ib_send_cm_dreq
ib_send_cm_drep
```

There are two routines that support path migration to an alternate path. These are:

```
ib_send_cm_lap
ib_send_cm_apr
```

The `ib_send_cm_lap` routine is used to request that an alternate path be loaded. The `ib_send_cm_apr` routine sends a response to the alternative path request, indicating if the alternate path was accepted.

6.1 Service ID Queries

InfiniBand provides a mechanism to allow services to register their existence with the subnet administrator. Other nodes can then query the subnet administrator to locate other nodes that have this service and get information needed to communicate with the other nodes. For example, clients can discover if a node contains a specific UD service. Given the service ID, the client can discover the QP number and QKey of the service on the remote node. This can then be used to send datagrams to the remote service. The communication manager provides the following routines to assist in service ID resolution.

```
ib_send_cm_sidr_req
ib_send_cm_sidr_rep
```

7 InfiniBand work request and completion event processing

Once a client has created QPs and CQs, registered memory, and established a connection or set up the QP for receiving datagrams, it can transfer data using the work request APIs. To send messages, perform RDMA reads or writes, or perform atomic operations, a client posts send work request elements (WQE) to the send queue of the queue pair. The format of the WQEs along with other critical data structures are located in `linux-2.6.11/drivers/infiniband/include/ib_verbs.h`. To allow data to be received, the client must first post receive WQEs to the receive queue of the QP.

```
ib_post_send
ib_post_recv
```

The post routines allow the client to post a list of WQEs that are linked via a linked list. If the format of WQE is bad and the post routine detects the error at post time, the post routines return a pointer to the bad WQE.

To process completions, a client typically sets up a completion callback handler when the completion queue (CQ) is created. The client can then call `ib_req_notify_cq` to request a notification callback on a given CQ. The `ib_req_ncomp_notif` routine allows the completion to be delivered after n WQEs have completed, rather than receiving a callback after a single one.

```
ib_req_notify_cq
ib_req_ncomp_notif
```

The mid-layer also provides routines for polling for completions and peeking to see how many completions are currently pending on the completion queue. These are:

```
ib_poll_cq
ib_peek_cq
```

Finally, there is the possibility that the client might receive an asynchronous event from the InfiniBand device. This happens for certain types of errors or ports coming online or going offline. Readers should refer to section 11.6.3 of the InfiniBand specification [IBTA] for a list of possible asynchronous event types. The mid-layer provides the following routines to register for asynchronous events.

```
ib_register_event_handler
ib_unregister_event_handler
```

8 Userspace InfiniBand Access

The InfiniBand architecture is designed so that multiple userspace processes can share a single InfiniBand adapter at the same time, with each the process using a private context so that fast path operation can access the adapter hardware directly without requiring the overhead of a system call or a copy between kernel space and userspace.

Work is currently underway to add this support to the Linux InfiniBand stack. A kernel module `ib_uverbs.ko` implements character special devices that are used for control path operations, such as allocating userspace contexts and pinning userspace memory as well as creating InfiniBand resources such as queue pairs and completion queues. On the userspace side, a library called libibverbs will provide an API in userspace similar to the kernel API described above.

In addition to adding support for accessing the verbs from userspace, a kernel module (`ib_umad.ko`) allows access to MAD services from userspace.

There also now exists a kernel module to proxy CM services into userspace. The kernel module is called `ib_ucm.ko`.

As the userspace infrastructure is still under construction, it has not yet been incorporated into the main kernel tree, but it is expected to be submitted to lkml in the near future. People that want get early access to the code can download it from the InfiniBand subversion development tree available from `openib.org`.

9 Acknowledgments

We would like to acknowledge the United States Department of Energy for their fund-

ing of InfiniBand open source work and for their computing resources used to host the `openib.org` web site and subversion data base. We would also like to acknowledge the DOE for their work in scale-up testing of the InfiniBand code using their large clusters.

We would also like to acknowledge all of the companies of the `openib.org` alliance that have applied resources to the `openib.org` InfiniBand open source project.

Finally we would like the acknowledge the help of all of the individuals in the Linux community that have submitted patches, provided code reviews, and helped with testing to ensure the InfiniBand code is stable.

10 Availability

The latest stable release of the InfiniBand code is available in the Linux releases (starting in 2.6.11) available from `kernel.org`.

```
ftp://kernel.org/pub
```

For those that want to track the latest InfiniBand development tree, it is located in a subversion database at openib.org.

```
svn checkout
https://openib.org/svn/gen2
```

References

[IBTA] The InfiniBand Architecture Specification, Vol. 1, Release 1.2
`http://www.ibta.org`

[SDP] The SDP driver, developed by Libor Michalek
`http://www.openib.org`

[IPMI] Intelligent Platform Management Interface
`http://developer.intel.com`

[SNMP] Simple Network Management Protocol
`http://www.ietf.org`

[LJ] May 2005 Linux Journal—InfiniBand and Linux
`http://www.linuxjournal.com`

Linux Is Now IPv6 Ready

Hideaki Yoshifuji
Keio University / USAGI/WIDE Project
yoshfuji@linux-ipv6.org

Abstract

Linux has included an IPv6 protocol stack for a long time. Its quality, however, was not quite good at the early stage. The USAGI Project was founded to promote this situation and provide high quality IPv6 stack for Linux systems. As a result of 5 years of our intensive activity, our stack is now certified as IPv6 Ready. It has been merged into main-line kernel so that the Linux IPv6 stack has enough quality to get the IPv6 Ready Logo now. To maintain the stack stable, we developed an automatic testing system, which greatly helps us saving our time. In this paper and the relevant presentation, we will show our efforts and technology to get the Logo and to maintain the quality of kernel. In addition, we will discuss our future plan.

1 Introduction

The current Internet has been running with the Internet Protocol Version 4, so called as IPv4, since the end of 1960s. At the end of 1980, the internet experts working at the IETF (Internet Engineering Task Force) recognized that we needed a new version of Internet Protocol to cope with rapid growth of the Internet. In 1992, this new version of protocol was named as IPng (IP next generation).

The first full-scale technical discussion on the IPng started in 1992 at the IETF. IPng, i.e., Internet Protocol Version 6 (IPv6), was intended to solve the various problems on the traditional IPv4, such as performance of packet forwarding, protocol extensibility, security and privacy.

According to the above principles, the basic specification of IPv6 was defined in 1994. After a series of experimental implementation and network operation (e.g., 6bone), the IPv6 technology is now getting into professional phase and applied to production. Commercial IPv6 services by Internet Service Providers and the applications running with IPv6 have been already available around us. This means that IPv6 stack implemented in any devices must be of production quality.

Linux system has also supported the IPv6 protocol as well as other operating systems such as FreeBSD, Sun Solaris and Microsoft Windows XP. Linux has included IPv6 stack since 1996 when early Linux 2.1.x version released. However, Linux IPv6 stack was not actively developed nor maintained for some time.

Considering above circumstances, USAGI Project was lunched in October, 2000. To deploy IPv6, it aims at providing improved IPv6 stack on Linux, which is one of the most popular open-source operating system in the world.

With a number of developments and treatments on problems, the quality has been remarkably

improved. It is now good enough to be certified to the IPv6 Ready Logo Phase-1.

2 Quality Assessment of IPv6

There are some basic concepts about the quality assessment of IPv6. Among them, TAHI Conformance Test Suite, IPv6 Ready Logo Program Phase-1 and Phase-2 are most important ones.

2.1 TAHI Conformance Test Suite

TAHI Conformance Teat Suite is designed to examine the conformity to the IPv6 specifications. The details of the test are described in the test-scripts including the following fields; e.g.

- IPv6 Core
- ICMPv6
- Neighbor Discovery
- Stateless Address Autoconfiguration
- Path MTU Discovery
- Tunneling
- Robustness
- IPsec

This test suite is considered one of the de-facto standard tools for judgment of conformance of IPv6 stack. Linux IPv6 stack can also be examined by this suite.

2.2 IPv6 Ready Logo Program

IPv6 Ready Logo Program is a worldwide authorization activity for the interoperability on the IPv6. To obtain the certification, applicants should submit corresponding results of self test and pass the examination of the interoperability for the test scenario prior to the judgment. Some test sets, such as TAHI Conformance Test Tool that is a core part of TAHI Conformance Test Suite, are admitted as a tool for the Self Test.

2.2.1 Phase-1

The IPv6 Ready Logo Phase-1 indicates that the product includes IPv6 mandatory core protocols and can interoperate with any other IPv6 equipments. Self Test covers mandatory features of IPv6 core, ICMPv6, Neighbor Discovery, and Stateless Address Autoconfiguration. On the other hand, simple trial for the interoperability is carried out.

2.2.2 Phase-2

Phase-2 logo indicates that a product has successfully satisfied strong requirements stated by the IPv6 Logo Committee (v6LC). The v6LC defines the test profiles with associated requirements for specific functionalities.

The Core Protocols Logo covers the fields of IPv6 core, NDP, Addrconf, PMTU, ICMPv6, is designed to examine the MUST- and SHOULD- items in specifications, and its tests for interoperability are much more complicated than those of Phase-1. Other discussions are underway on the tests for IPsec, MLDv2, Mobile IPv6.

3 Quality Improvement Activities on Linux

As mentioned above, when the USAGI Project started, the quality of IPv6 stack is far beyond satisfaction though it was available on Linux. Linux IPv6 stack could not get good scores in the fields of Neighbor Discovery and Stateless Address Autoconfiguration in TAHI IPv6 Conformance Test Suite.

The members of USAGI Project and other contributers analyzed the problems of the stack, which are categorized as follows:

Improper State Transition. In Neighbor Discovery, improper state transition to the specification had been carried out. To solve the problem, the mutual dependency was sorted out in state machine to make the maintenance easier.

Inadequate time management. Time management at Neighbor Discovery and Stateless Address Auto-configuration did not have enough time accuracy. We conducted the structural reform mentioned above which enable the simplification of the management and more accurate time control.

Inadequate use of routing. In the previous method, there were some occasions where invalid route was used in an improper way.

Improper treatment against wrong input. Checks of input from outer sources were not adequate, improper treatments were going on for the wrong or malicious input.

The project results dealing with the above problems have been applied in main-line kernel step by step, until the version 2.6.11-rc2.

4 TAHI Automatic Running System

USAGI Project has been seeking for more featured and higher functional code, improving IPv6 stack for Linux, its libraries and applications. The results have been gradually accepted in the Linux community. For example, many improvements on IPv6 stack have been added in main-line kernels.

These good results are obtained by introducing TAHI Automatic Running System, which is improved system of TAHI Conformance Test Suite.

4.1 Background

The main objective of USAGI Project is to provide a better environment of IPv6 for Linux, and all the member of this project have been working hard for this purpose. One of those activities is to merge USAGI kernel patches to the main-line kernel.

Active improvement and amendment of the main-line kernel are under way on daily basis. As for the codes around the network, other patches as well as those by USAGI are tried to be taken in. Many maintainers and contributors are always wary of not being involved in mixing up bugs, but it is very difficult to avoid completely the possibility of regression after alternation.

The improved code is accepted widely in Linux community. While it is important to continue developing new functionality further more, to maintain the quality of present code is definitely necessary.

In order to solve these problems, a system was developed, which enables us to test the

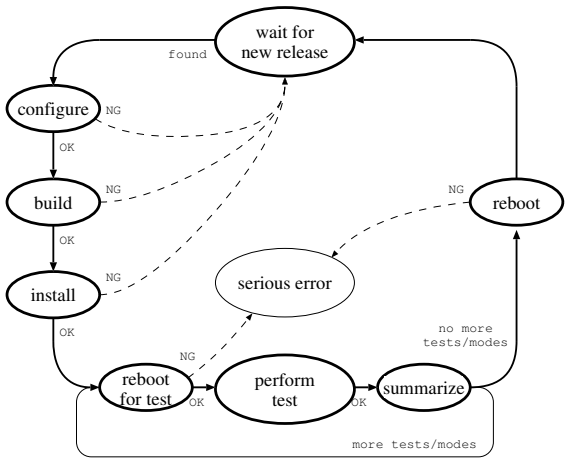

Figure 1: The Flowchart of the system

functions of IPv6 stack at each release of the snapshots of main-line kernel that is published every day. TAHI IPv6 Conformance Test Tool (`http://www.tahi.org`) provided by TAHI Project is used in order to test the functions of IPv6. Using this system that runs the test automatically, immediate amendment and tackling of the problems are possible even if some regression may be observed on the functions of IPv6. The system is open to the general public, and you can see it at `http://testlab.linux-ipv6.org` through the connect via IPv6.

4.2 Procedure of the System

The system is a bunch of some procedures, each of which consists of waiting for new release of the kernel, building-up, and testing. Those procedures are repeated, and the results are observed. The state transition of each procedure is shown in Figure 1.

The system waits for new kernel release when the test is not performed. The release objectives are not only stable version, but also rc version that is a preparing stage for stable version, together with bk version[1] that is released every night.

When the system finds a new release of a version, it begins to build up the kernel with automatic procedures of configuration, building-up, and installment. The logs in each procedure are preserved in the system, so that building errors can be analyzed. When the treatment of each procedure fails, the system assumes that the source contained the cause of problem, waiting for next release of version.

Once the building-up of the kernel has finished, the system carries out the test. This system is designed to run multiple tests with several settings for one kernel. For this purpose, NUT, the test target, is rebooted with proper mode such as router or host, and with proper settings such as IP address, prior to the launch of each test, and then the test is carried out. Each time when the test is finished, the result is shown in a table, which is compared with the previous records. That enable us to make sure if the regression might have occurred after the introduction of a new patch. The logs of each rebooting process and test result are preserved. If the system failed to reboot the target, it will stop its automatic operation and wait for manual resume after checking.

After the test of kernel, the system will be back to the stage of waiting for a new release of kernel. Before this transition to the waiting stage, the test target will be rebooted in order to get it back to the stable stage, with putting it back to a stable kernel verified. The logs at the stage of rebooting are also preserved.

4.3 Collected Data and Access to the Information

The system collects many kinds of data, such as the results of tests, the differences between

[1] as of April 2005

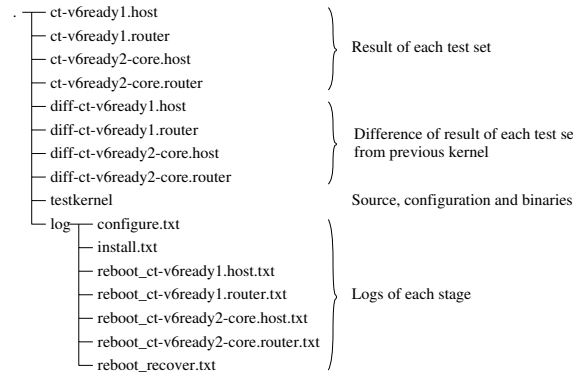

Figure 2: Data Collected by the Autorun System

each test result and its previous equivalent test, the source and the compiled binaries, or various logs in each process of tests like kernel-build. Figure 2 shows what kinds of data should be collected at each release of kernel.

Each datum is exported by HTTP daemon and people can browse it using web browsers. The browser window is designed to seek the objective data open to the public as quick as possible. Figure 3 shows an access example through the web browser.

4.4 Development in Future

As of April in 2005, according to this system, IPv6 Ready Logo Phase 1 Self Test and Phase 2 Core Protocols Self Test are conducted only on the version 2.6 in main-line kernel. The expecting developments in future are as follows;

1. parallel proceedings to different kernel release, such as USAGI kernel and main-line kernel
2. supporting other tests, such as those for IPsec, MLDv2 and Mobile IPv6
3. General definition to the test target and test sets

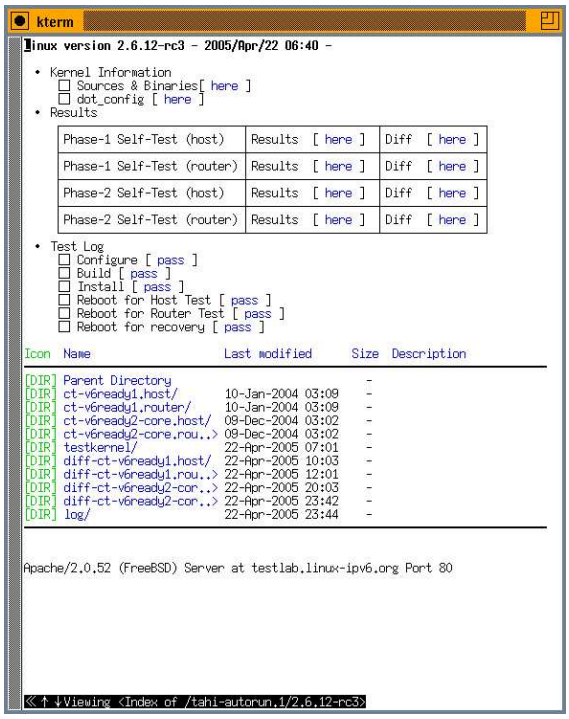

Figure 3: Access Example through Web Browser

These items are now being discussed under development.

5 Linux Is IPv6 Ready

As described in Section 2.2, IPv6 Ready Logo Program is an international activity for proof of interoperability. As of December 2004, more than 120 products have been approved with gaining the Phase-1 certification, which means those products have the basic interoperability in IPv6.

IPv6 used to be classified as EXPERIMENTAL in Linux, so that people are worrying for long time that IPv6 in the Linux would not be useful. Getting this logo, however, the anxiety would be expelled completely.

5.1 USAGI Project

USAGI Project decided to join this program to gain the international certificate, which will show what credible results we are offering.

In 2004, USAGI Project took part in IPv6 Ready Logo Program with its patched kernel and tool. In February, the Project obtained IPv6 Ready Logo Phase-1 on both functions of host and router with its product based on 2.6 kernel and with its enhanced tool. In September 2004, and in April 2005, it gained IPv6 Ready Logo Phase-1 on 2.4-based kernel, in addition to 2.6-based one, on the functions of host and router.

5.2 Main-line

Many improvements by USAGI Project members and other developers were unified into the main-line kernel in the 2.6.11 timeframe, and the version 2.6.11-rc2, with patched radvd (router advertisement daemon), is finally approved with IPv6 Ready Logo Phase-1.

In addition to this, the version 2.6.12 will include the kernel function that is needed to get IPv6 Ready Logo Phase-2 certificate.

5.3 KNOPPIX/IPv6

USAGI Project collaborated with KNOPPIX/IPv6 (`http://www.alpha.co.jp/knoppix/ipv6/`), which uses the provided code by USAGI to their products, and helped them to take part in IPv6 Ready Logo Program. KNOPPIX (`http://www.knopper.net/knoppix/`) is one of the major Linux distributions which makes it possible to boot with single CD without any special installing operation. USAGI Project collaborated with AIST (National Institute of Advanced Industrial Science and Technology), Alpha Systemss Inc. to develop special IPv6-aware KNOPPIX based on KNOPPIX Japanese Edition (`http://unit.aist.go.jp/itri/knoppix/`). The code provided by USAGI Project was integrated, and the resulting product was named “KNOPPIX/IPv6” (`http://www.alpha.co.jp/knoppix/ipv6/`). With KNOPPIX/IPv6, together with the merit of KNOPPIX that “only starting of CD-ROM is needed without installment and setting-up” and the high quality IPv6 protocol stack by USAGI Project, a new technical 1 CD OS world has been achieved where beginners can easily experience the IPv6 world.

To summarize the dealing with situation of KNOPPIX/IPv6, major desktop applications such as web browsers (Mozilla, Konqueror), mail clients (Sylpheed, Kmail) support IPv6. On the other hand, as for coping with fundamental IPv6 networking, 6to4 is adopted. This function is supported, because users do not always connect their machines to the global IPv6 Internet. Even if a user connects to the IPv4-only network, KNOPPIX/IPv6 automatically detects it and configures 6to4 tunnel to the outside local network. Therefore, from now on, it is possible for users to enjoy more sophisticated network without realizing whether it is IPv4 or IPv6.

In September 2004, KNOPPIX/IPv6 could obtain IPv6 Ready Logo Phase-1 on both kernel versions based on 2.4 and 2.6. The necessary tests to confirm interoperability were conducted at laboratory of USAGI Project in Keio University as a cooperative work of the Project and developers of the KNOPPIX Japanese Edition.

5.4 Development in Future

USAGI Project is going to participate in IPv6 Ready Logo Phase-1 Program to improve the quality of interoperability.

On the other hand, in the new IPv6 Ready Logo Phase-2 Program, the aim of which is verification of the system whether it is available in the real network environment, not only basic IPv6 functions but also IPsec, MIPv6, and MLD are subject to verification. USAGI Project is going to participate actively in the Phase-2 Program, and will play an initiative role in the quality improvement.

The Self Test for IPv6 Ready Logo contains a lot of its functions, playing a great role for the quality improvement and maintenance of the Linux, contributing to the less personal burden of labor. However, it does not warrant liability against the stress and attack from the outside, nor the stability in SMP (symmetric multiprocessing). Now, IPv6 is enabled by default in Linux, it is more important to maintain the stability of the system on the more complicated in higher stage. The members of this project will continue to achieve this objective through fulfillment of the experiments and the practical environment.

Acknowledgements

This work was performed based on the USAGI Project and the author is grateful to valuable comments and helpful discussion with Prof. Jun Murai at Keio University and Prof. Hiroshi Esaki at The University of Tokyo.

The usbmon: USB monitoring framework

Pete Zaitcev
Red Hat, Inc.
zaitcev@redhat.com

Abstract

For years, Linux developers used `printk()` to debug the USB stack, but this approach has serious limitations. In this paper we discuss "usbmon," a recently developed facility to snoop USB traffic in a more efficient way than can be done with `printk()`.

From far away, usbmon is a very straightforward piece of code. It consists of circular buffers which are filled with records by hooks into the USB stack and a thin glue to the user code which fetches these records. The devil, however, is in details. Also the user mode tools play a role.

1 Introduction

This paper largely deals with the kernel part of the USB monitoring infrastructure, which is properly called "usbmon" (all in lower case). We describe usbmon's origins, overall design, internals, and how it is used, both by C code in kernel and by human users. To conclude, we consider if experience with usbmon is applicable to subsystems other than USB.

2 Origins

Although the need to have a robust, simple, and unobtrusive method of snooping appears to be self-evident, Linux USB developers were getting by with layers of macros on top of `printk()` for years. Current debugging facilities are represented by a patchwork of build-time configuration settings, such as `CONFIG_USB_STORAGE_DEBUG`. To make the use of systems with tracing included more palatable, `usbserial` and several other modules offer "debug" parameter.

Limitations of the this approach became pronounced as more users running preconfigured kernels appeared. For a developer, it is often undesirable to make users to rebuild their kernels with `CONFIG_USB_STORAGE_DEBUG` enabled. These difficulties could be overcome by making tracing configurable at runtime, by a module parameter. Nonetheless, this style of tracing is still not ideal, for several reasons. Output to the system console and/or log file is lossy when a large amount of data is piped through though the syslog subsystem. Timing variations introduced by formatted printouts skew results, which makes it harder to pinpoint problems when peripherals require delays in the access pattern. And finally, `printk()` calls have to be added all the time to capture what is important. Often it seems as if the one key `printk()` is missing, but once added, it

stays in the code forever and serves to obscure printouts needed at that time.

A facility similar to tcpdump(8) would be a great help for USB developers. The usbmon aims to provide one.

David Harding proposed a patch to address this need back in 2002, but for various reasons that effort stalled without producing anything suitable to be accepted into the Linus' kernel. The usbmon picks up where the previous work left and is available in the mainline kernel starting with version 2.6.11.

3 Architecture

The USB monitoring or snooping facilities for Linux consist of the kernel part, or usbmon, and the user mode part, or user mode tools. To jump-start the development, usbmon took the lead while tools lagged.

At highest level of architecture, usbmon is uncomplicated. It consists of circular buffers, fed by hooks in the USB stack. Every call puts an event into a buffer. From there, user processes fetch these events for further processing or presentation to humans. Events for all devices on a particular bus are delivered to users together, separately from other buses. There is no filtering facility of any sort.

At the lower level, a couple of interesting decisions were made regarding the placement of hooks and the formatting of events when presented to user programs.

An I/O request in the USB stack is represented by so-called "URB." A peripheral-specific driver, such as usb-storage, initializes and submits URB with a call to the USB stack core. The core dispatches URB to a Host Controller Driver, or HCD. When I/O is done, HCD invokes a specified callback to notify the core and the requesting driver. The usbmon hooks reside in the core of USB stack, in the submission and callback paths. Thus, usbmon relies on HCD to function properly and is only marginally useful in debugging of HCDs. Such an arrangement is accepted for two reasons. First, it allows usbmon to be unobtrusive and significantly less buggy itself. Second, the vast majority of bugs occur outside of HCDs, in in upper level drivers or peripherals.

The user interface to the usbmon answers to diverse sets of requirements with priorities changing over time. Initially, a premium is placed on ease of implementation and the possibility to access the information with simple tools. But in perspective, performance starts to play a larger role. The first group of requirements favors an interface typified by `/proc` filesystem, the one of pseudo text files. The second group pulls toward a binary and versioned API.

Instead of forcing a choice between text and binary interfaces, usbmon adopts a neutral solution. Its data structures are set up to facilitate several distinct types of consumers of events (called "readers"). Various reader classes can provide text and binary interfaces. At this time, only text-based interface class is implemented.

Every instance of a reader has its own circular buffer. When hooks are called, they broadcast events to readers. Readers replicate events into all buffers which are active for a given bus. To be sure, this entails an extra overhead of data copying. However, the complication of having all aliasing properly tracked and resolved turned out to be insurmountable in the time frame desired, and the performance impact was found manageable.

```
struct mon_bus {
    struct list_head bus_link;
    spinlock_t lock;
    struct dentry *dent_s;      /* Debugging file */
    struct dentry *dent_t;      /* Text interface file */
    struct usb_bus *u_bus;
    /* Ref */
    int nreaders;               /* Under mon_lock AND mbus->lock */
    struct list_head r_list;    /* Chain of readers (usually one) */
    struct kref ref;            /* Under mon_lock */
    /* Stats */
    unsigned int cnt_text_lost;
};

struct mon_reader {    /* An instance of a process which opened a file */
    struct list_head r_link;
    struct mon_bus *m_bus;
    void *r_data;
    void (*rnf_submit)(void *data, struct urb *urb);
    void (*rnf_complete)(void *data, struct urb *urb);
};
```

Figure 1: The bus and readers.

4 Implementation

The usbmon is implemented as a Linux kernel module, which can be loaded and unloaded at will. This arrangement is not intrinsic to the design; it is intended to serve as a convenience to developers only. Hooks and additional data fields remain built into the USB stack core at all times as long as usbmon is configured on. In a proprietary OS, usbmon would have to be implemented in a different way. It is entirely possible to make the usbmon an add-on that stacks on top of HCDs by manipulating existing function pointers. Such an implementation would make usbmon effectively non-existing when not actively monitoring. However, this approach introduces a significant complexity of tracking of active URBs which had their function pointers replaced, and brings only a marginal advantage for an open-source OS. In present, when usbmon is not running, it adds 8 bytes of memory use per bus (on a 32-bit system) and an additional `if()` statement in submit and callback paths. This was deemed an acceptable price for the lack of tracking.

The key data structure that keeps usbmon together is `struct mon_bus` (See Figure 1). One of them is allocated for every USB bus present. It holds a list of readers attached to the bus, pointer to the corresponding bus structure, statistic counters, reference count, and a spinlock. The manner in which circular buffers are arranged is encapsulated entirely within a reader.

The locking model is straightforward. All hooks execute with the bus spinlock taken, so readers do not do any extra locking. The only time instances of `struct mon_bus` may in-

fluence each other is when buses are added or removed. Data words touched at that time, such as linked list entries, are covered with a global semaphore, `mon_lock`.

The reference count is needed because buses are added and removed while user processes access devices. Captured events may remain in buffers after a bus was removed. The count is implemented with a predefined kernel facility called `kref`. The `mon_lock` is used to support kref as required by API.

5 Interfaces

The usbmon provides two principal interfaces: the one into the USB core and the other facing the user processes.

The USB core interface is conventional for an internal Linux kernel API. It consists of registration and deregistration routines provided by the core, operations table that is passed to the core upon registration, and inline functions for hooks called by the core. It all comes down to the code shown in Figure 2.

As was mentioned previously, only one type of interface to user processes exists at present: text interface. It is implemented with the help of a pseudo filesystem called "`debugfs`" and consists of a few pseudo files, same group per every USB bus in the system. Text records are produced for every event, to be read from pseudo files. Their format is discussed below.

6 Use (user mode)

A common way to access usbmon without any special tools is as following:

```
# mount -t debugfs none /sys/kernel/debug
# modprobe usbmon
# cat /sys/kernel/debug/usbmon/3t
dfa105cc 1578069602 C Ii:001:01 0 1 D
dfa105cc 1578069644 S Ii:001:01 -115 2 D
d6bda284 1578069665 S Ci:001:00 -115 4 <
d6bda284 1578069672 C Ci:001:00 0 4 = 01010100
........
```

The number 3 in the file name is the number of the USB bus as reported by `/proc/bus/usb/devices`.

Each record copied by `cat` starts with a tag that is used to correlate various events happening to the same URB. The tag is simply a kernel address of the URB. Next words are: a timestamp in milliseconds, event type, a joint word for the type of transfer, device number, and endpoint number, I/O status, data size, data marker and a varying number of data words. More precise documentation exists within the kernel source code, in file `Documentation/usb/usbmon.txt`.

This format is terse, but it can be read by humans in a pinch. It is also useful for postings to mailing lists, Bugzilla attachments, and other similar forms of data exchange.

Tools to ease dealing with usbmon are being developed. Only one such tool exists today: the USBMon (written with upper case letters), originally written by David Harding. It is a tool with a graphical interface.

7 Lessons

When compared to tcpdump(8) or Ethereal(1), usbmon today is rudimentary. Despite that, in the short time it has existed, usbmon helped the author to quickly pinpoint several bugs that otherwise would take many kernel rebuilds and gaining an understanding of unfamiliar system

```
struct usb_mon_operations {
    void (*urb_submit)(struct usb_bus *bus, struct urb *urb);
    void (*urb_submit_error)(struct usb_bus *bus, struct urb *urb, int err);
    void (*urb_complete)(struct usb_bus *bus, struct urb *urb);
    void (*bus_add)(struct usb_bus *bus);
    void (*bus_remove)(struct usb_bus *bus);
};

extern struct usb_mon_operations *mon_ops;

static inline void usbmon_urb_submit(struct usb_bus *bus, struct urb *urb)
{
    if (bus->monitored)
        (*mon_ops->urb_submit)(bus, urb);
}

static inline void usbmon_notify_bus_remove(struct usb_bus *bus)
{
    if (mon_ops)
        (*mon_ops->bus_remove)(bus);
}

int usb_mon_register(struct usb_mon_operations *ops);
void usb_mon_deregister(void);
```

Figure 2: The interface to the USB core.

log messages. Having any sort of usable unified tracing is helpful when developers have to work with an explicitly programmed message passing bus.

A large part of usbmon's utility comes from being always enabled, which requires its overhead to be undetectable when inactive and low enough not to change the system's behaviour when active. So it probably is unreasonable to implement an equivalent of usbmon for PCI: performance overhead may be too great; the level of messages is too low; there are no standard protocols to be parsed by upper level tools. But developers of subsystems such as SCSI or Infiniband are likely to benefit from introduction of "scsimon" or "infinimon" into their set of tools.

8 Future Work

The usbmon and user mode tools have a long way to go before they can be as developed as tcpdump(8) is today. Below we list issues which are prominent now.

- When USB buses are added and removed, tools have to be notified, which is not done at present. As a workaround, tools rescan file `/proc/bus/usb/devices` periodically. A solution may be something as

simple as select(2) working on a special file.

- Users often observe records which should carry data, but do not. For example:

```
c07835cc 1579486375 S Co:002:00 -115 0
d2ac6054 1579521858 S Ii:002:02 -115 4 D
```

  In the first case, setup packet of a control transfer is not captured, and in the second case, data part of an interrupt transfer is missing. The 'D' marker means that, according to the flags in the URB, the data was not mapped into the kernel space, and was only available for the DMA. The code to handle these cases has yet to be developed.

- The raw text data is difficult to interpret for people. So, it is desirable to decode the output to higher level protocols: SCSI commands, HID reports, hub control messages. This task belongs to the tools such as USBMon.

- Some tool developers express preferences for a binary and versioned API to compliment the existing text-based interface to usbmon. These requests need to be addressed.

References

Van Jacobson et al. *tcpdump(8), the manual.* In tcpdump version 3.8.2, 2004.

Greg Kroah-Hartman. *kobjects and krefs.* In Proceedings of the Linux Symposium (former OLS) 2004.

Adopting and Commenting the Old Kernel Source Code for Education

Jiong Zhao
University of TongJi, Shanghai
gohigh@gmail.com

Trent Jarvi
University of Denver
taj@www.linux.uk.org

Abstract

Dissecting older kernels including their problems can be educational and an entertaining review of where we have been. In this session, we examine the older Linux kernel version 0.11 and discuss some of our findings. The primary reason for selecting this historical kernel is that we have found that the current kernel's vast quantity of source code is far too complex for hands-on learning purposes. Since the 0.11 kernel has only 14,000 lines of code, we can easily describe it in detail and perform some meaningful experiments with a runnable system effienctly. We then examine several aspects of the kernel including the memory management, stack usage and other aspects of the Linux kernel. Next we explain several aspects of using Bochs emulator to perform experiments with the Linux 0.11 kernel. Finally, we present and describe the structure of the Linux kernel source including the `lib/` directory.

1 Introduction

As Linus once said, if one wants to understand the details of a software project, one should "RTFSC—Read The F**king Source Code." The kernel is a complete system, the parts relate to each other to fulfill the functions of a operating system. There are many hidden details in the system. If one ignores these details, like a blind men trying to size up the elephant by taking a part for the whole, its hard to understand the entire system and is difficult to understand the design and implementations of an actual system. Although one may obtain some of the operating theory through reading classical books like the "The design of Unix operating system," [4] the composition and internal relationships in an operating system are not easy to comprehend. Andrew Tanenbaum, the author of MINIX[1], once said in his book, "teaching only theory leaves the student with a lopsided view of what an operating system is really like." and "Subjects that really are important, such as I/O and file systems, are generally neglected because there is little theory about them." As a result, one may not know the tricks involved in implementing a real operating system. Only after reading the entire source code of a operating system, may one get a feeling of sudden enlightened about the kernel.

In 1991 Linus made a similar statements[5] after distributing kernel version 0.03, "The *GNU* kernel (Hurd) will be free, but is currently not ready, and will be too big to understand and learn." Likewise, the current Linux kernel is too large to easily understand. Due to the small amount of code (only 14,000 lines) as shown in Figure 1, the usability and the consistency

with the current kernel, it is feasible to choose Linux 0.11 kernel for students to learn and perform experiments. The features of the 0.11 kernel are so limited, it doesn't even contain job control or virtual memory swapping. It can, however, still be run as a complete operating system. As with an introductory book on operating systems, we need not deal with the more complicated components such as *VFS*, *ext3*, networking and more comprehensive memory management systems in a modern kernel. As students understand of the main concepts concerning how an operating system is generally implemented, they can learn to understand the advanced parts for themselves. Thus, both the teaching and learning become more efficient and consume considerably less time. The lower barrier to entry for learning can even stimulate many young people to take part in and involve in the Linux activities.

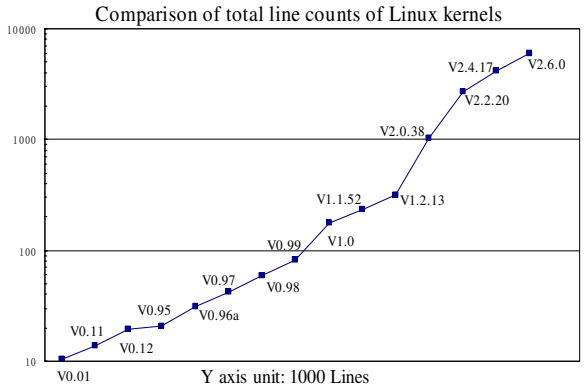

Figure 1: Lines of code in various kernel versions

From teaching experience and student feedback, we found the most difficult part of studying the 0.11 kernel is the memory management. Therefore, in the following sections we mainly deal with how the 0.11 kernel manages and uses memory in the protected mode of the Intel *IA-32* processor along with the different kinds of stacks used during the kernel initialization of each task.

2 Linux Kernel Architecture

The Linux kernel is composed of five modules: task scheduling, memory management, file system, interprocess communication (IPC) and network. The task scheduling module is responsible for controlling the usage of the processor for all tasks in the system. The strategy used for scheduling is to provide reasonable and fair usage between all tasks in the system while at the same time insuring the processing of hardware operations. The memory management module is used to insure that all tasks can share the main memory on the machine and provide the support for virtual memory mechanisms. The file system module is used to support the driving of and storage in peripheral devices. Virtual file system modules hide the various differences in details of the hardware devices by providing a universal file interface for peripheral storage equipment and providing support for multiple formats of file systems. The *IPC* module is used to provide the means for exchanging messages between processes. The network interface module provides access to multiple communication standards and supports various types of network hardware.

The relationship between these modules is illustrated in Figure 2. The lines between them indicates the dependences of each other. The dashed lines and dashed line box indicate the part not implemented in Linux 0.1x.

The figure shows the scheduling module relationship with all the other modules in the kernel since they all depend on the schedules provided to suspend and restart their tasks. Generally, a module may hang when waiting for hardware operations, and continue running after the hardware operation finishes. The other three modules have like relationships with the schedule module for similar reasons.

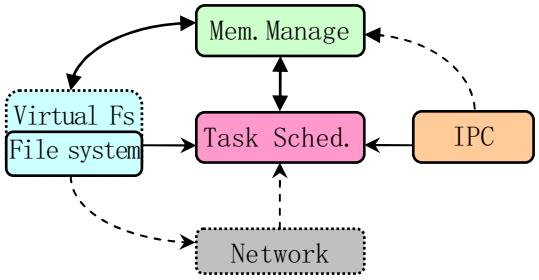

Figure 2: The relationship between Linux kernel modules

The remaining modules have implicit dependences with each other. The scheduling subsystem needs memory management to adjust the physical memory space used by each task. The IPC subsystem requires the memory management module to support shared memory communication mechanisms. Virtual file systems can also use the network interface to support the network file system (*NFS*). The memory management subsystem may also use the file system to support the swapping of memory data blocks.

From the monolithic model structure, we can illustrate the main kernel modules in Figure 3 based on the structure of the Linux 0.11 kernel source code.

3 Memory Usage

In this section, we first describe the usage of physical memory in Linux 0.1x kernel. Then we explain the memory *segmentation*, *paging*, *multitasking* and the *protection* mechanisms. Finally, we summarize the relationship between virtual, linear, and physical address for the code and data in the kernel and for each task.

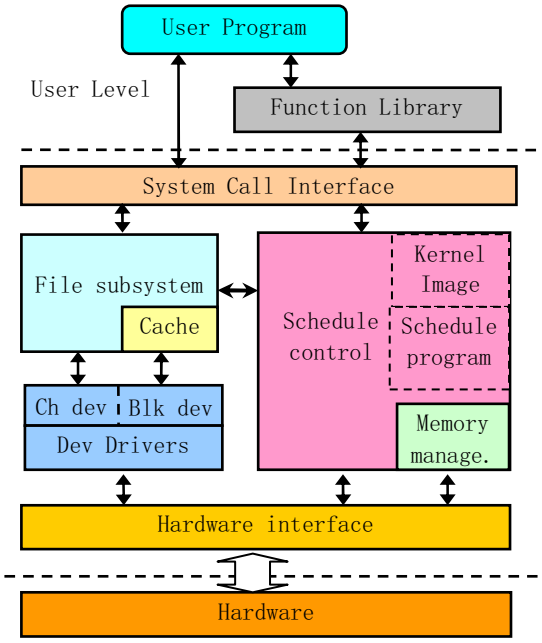

Figure 3: Kernel structure framework

3.1 Physical Memory

In order to use the physical memory of the machine efficiently with Linux 0.1x kernel, the memory is divided into several areas as shown in Figure 4.

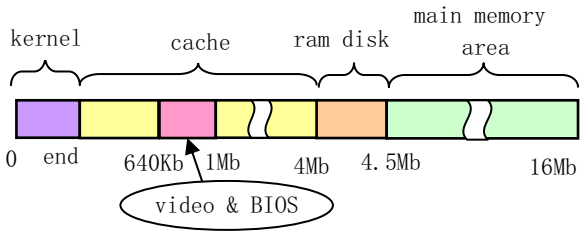

Figure 4: The regions of physical memory

As shown in Figure 4, the kernel code and data occupies the first portion of the physical memory. This is followed by the cache used for block devices such as hard disks and floppy drives eliminating the memory space used by the adapters and *ROM BIOS*. When a

task needs data from a block device, it will be first read into the cache area from the block device. When a task needs to output the data to a block device, the data is put into the cache area first and then is written into the block device by the hardware driver in due time. The last part of the physical memory is the main area used dynamically from programs. When kernel code needs a free memory page, it also needs to make a request from the memory management subsystem. For a system configured with virtual *RAM disks* in physical memory, space must be reserved in memory.

Physical memory is normally managed by the processor's memory management mechanisms to provide an efficient means for using the system resources. The Intel 80X86 CPU provides two memory management mechanisms: Segmentation and paging. The paging mechanism is optional and its use is determined by the system programmer. The Linux operating system uses both memory segmentation and paging mechanism approaches for flexibility and efficiency of memory usage.

3.2 Memory address space

To perform address mapping in the Linux kernel, we must first explain the three different address concepts used in *virtual* or *logical* address space, the CPU *linear* address space, and the actual *physical* address space. The *virtual* addresses used in virtual address space are addresses composed of the *segment selector* and *offset* in the segment generated by program. Since the two part address can not be used to access physical memory directly, this address is referred to as a virtual address and must use at least one of the address translation mechanisms provided by CPU to map into the physical memory space. The virtual address space is composed of the *global address space* addressed by the descriptors in global descriptor table (*GDT*) and the *local address space* addressed by the local descriptor table (*LDT*). The index part of a *segment selector* has thirteen bits and one bit for the table index. The Intel 80X86 processor can then provide a total of 16384 selectors so it can addresses a maximum of 64T of virtual address space[2]. The logical address is the offset portion of a virtual address. Sometimes this is also referred to as virtual address.

Linear address is the middle portion of address translation from virtual to physical addresses. This address space is addressable by the processor. A program can use a *logical address* or *offset* in a segment and the base address of the segment to get a linear address. If *paging* is enabled, the linear address can be translated to produced a physical address. If the *paging* is disabled, then the linear address is actually the same as physical address. The linear address space provided by Intel 80386 is 4 GB.

Physical address is the address on the processor's external address bus, and is the final result of address translation.

The other concept that we examine is *virtual memory*. Virtual memory allows the computer to appear to have more memory than it actually has. This permits programmers to write a program larger than the physical memory that the system has and allows large projects to be implemented on a computer with limited resources.

3.3 Segmentation and paging mechanisms

In a segmented memory system, the logical address of a program is automatically mapped or translated into the middle 4 GB linear address space. Each memory reference refers to the memory in a segment. When programs reference a memory address, a linear address is produced by adding the segment base address with

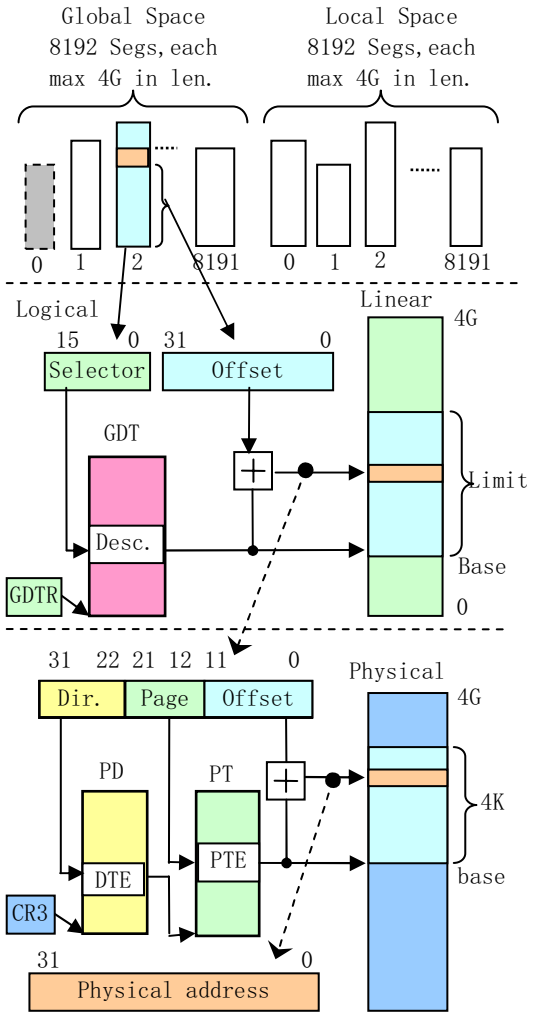

Figure 5: The translation between virtual or logical, linear and physical address

the logical address visible to the programmer. If *paging* is not enabled, at this time, the linear address is sent to the external address bus of the processor to access the corresponding physical address directly.

If *paging* is enabled on the processor, the linear address will be translated by the *paging* mechanism to get the final physical corresponding physical address. Similar to the segmentation, paging allow us to relocate each memory reference. The basic theory of paging is that the processor divides the whole linear space into pages of 4 KB. When programs request memory, the processor allocates memory in pages for the program.

Since Linux 0.1x kernel uses only one *page directory*, the mapping function from linear to physical space is same for the kernel and processes. To prevent tasks from interfering with each other and the kernel, they have to occupy different ranges in the linear address space. The Linux 0.1x kernel allocates 64MB of linear space for each task in the system, the system can therefor hold at most 64 simultaneous tasks (64MB * 64 = 4G) before occupying the entire Linear address space as illustrated in Figure 6.

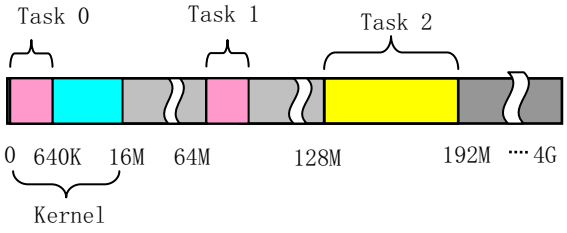

Figure 6: The usage of linear address space in the Linux 0.1x kernel

3.4 The relationship between virtual, linear and physical address

We have briefly described the memory segmentation and paging mechanisms. Now we will examine the relationship between the kernel and tasks in virtual, linear and physical address space. Since the creation of *tasks 0* and *1* are special, we'll explain them separately.

3.4.1 The address range of kernel

For the code and data in the Linux 0.1x kernel, the initialization in `head.s` has already set the limit for the kernel and data segments to

be 16MB in size. These two segments overlap at the same linear address space starting from address 0. The *page directory* and *page table* for kernel space are mapped to 0-16MB in physical memory (the same address range in both spaces). This is all of the memory that the system contains. Since one page table can manage or map 4MB, the kernel code and data occupies four entries in the *page directory*. In other words, there are four secondary page tables with 4MB each. As a result, the address in the kernel segment is the same in the physical memory. The relationship of these three address spaces in the kernel is depicted in Figure 7.

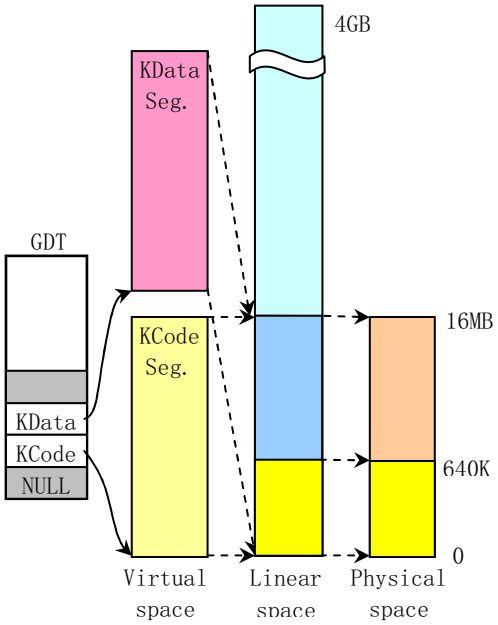

Figure 7: The relationship of the three address spaces in a 0.1x kernel

As seen in Figure 7, the Linux 0.1x kernel can manage at most 16MB of physical memory in 4096 page frames. As explained earlier, we know that: (1) the address range of kernel code and data segments are the same as in the physical memory space. This configuration can greatly reduce the initialization operations the kernel must perform. (2) *GDT* and Interrupt Descriptor Table (*IDT*) are in the kernel data segment, thus they are located in the same address in both address spaces. In the execution of code in `setup.s` in real mode, we have setup both temporary *GDT* and *IDT* at once. These are required before entering protected mode. Since they are located by physical address `0x90200` and this will be overlapped and used for block device cache, we have to recreate *GDT* and *IDT* in `head.s` after entering protected mode. The segment selectors need to be reloaded too. Since the locations of the two tables do not change after entering protected mode, we do not need to move or recreate them again. (3) All tasks except *task 0* need additional physical memory pages in different linear address space locations. They need the memory management module to dynamically setup their own mapping entries in the *page directory* and *page table*. Although the code and static data of *task 1* are located in kernel space, we need to obtain new pages to prevent interference with *task 0*. As a result, *task 1* also needs its own page entries.

While the default manageable physical memory is 16MB, a system need not contain 16MB memory. A machine with only 4MB or even 2MB could run Linux 0.1x smoothly. For a machine with only 4MB, the linear address range 4MB to 16MB will be mapped to nonexistent physical space by the kernel. This does not disrupt or crash the kernel. Since the kernel knows the exact physical memory size from the initialization stage, no pages will be mapped into this nonexistent physical space. In addition, since the kernel has limited the maximum physical memory to be 16MB at boot time (in `main()` corresponding to `startkernel()`), memory over 16MB will be left unused. By adding some page entries for the kernel and changing some of the kernel source, we certainly can make Linux 0.1x support more physical memory.

3.4.2 The address space relationship for task 0

Task 0 is artificially created or configured and run by using a special method. The limits of its code and data segments are set to the 640KB included in the kernel address space. Now *task 0* can use the kernel page entries directly without the need for creating new entries for itself. As a result, its segments are overlapped in linear address space too. The three space relationship is shown in Figure 8.

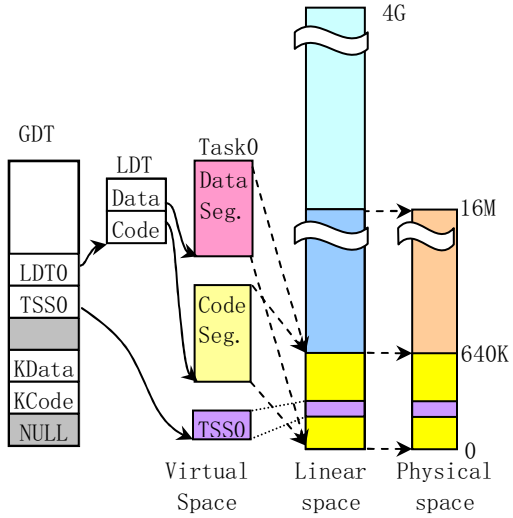

Figure 8: The relationship of three address spaces for task 0

As *task 0* is totally contained in the kernel space, there is no need to allocate pages from the main memory area for it. The kernel stack and the user stack for *task 0* are included the kernel space. *Task 0* still has read and write rights in the stacks since the page entries used by the kernel space have been initialized to be readable and writable with user privileges. In other words, the flags in page entries are set as *U/S=1, R/W=1*.

3.4.3 The address space relationship for task 1

Similar to *task 0*, *task 1* is also a special case in which the code and data segment are included in kernel module. The main difference is that when forking *task 1*, one free page is allocated from the main memory area to duplicate and store *task 0*'s page table entries for *task 1*. As a result, *task 1* has its own *page table* entries in the *page directory* and is located at range from 64MB to 128MB (actually 64MB to 64MB + 640KB) in linear address space. One additional page is allocated for task 1 to store its *task structure (PCB)* and is used as its kernel mode stack. The task's *Task State Segment (TSS)* is also contained in task's structure as illustrated in Figure 9.

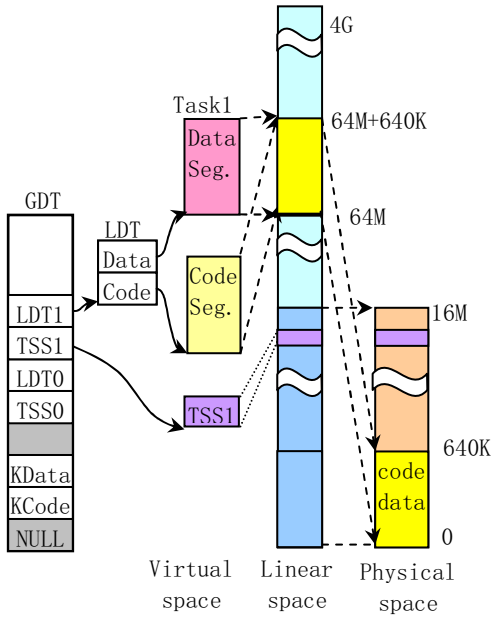

Figure 9: The relationship of the three address spaces in task 1

Task 1 and *task 0* will share their user stack `user_stack[]` (refer to `kernel/sched.c`, lines 67-72). Thus, the stack space should be "clean" before *task 1* uses it to ensure that there

is no unnecessary data on the stack. When forking *task 1*, the user stack is shared between *task 0* and *task 1*. However when *task 1* starts running, the stack operating in *task 1* would cause the processor to produce a page fault because the page entries have been modified to be read only. The memory management module will therefor need allocate a free page for *task 1's* stack.

3.4.4 The address space relationship for other tasks

For *task 2* and higher, the parent is *task 1* or the *init* process. As described earlier, Linux 0.1x can have 64 tasks running synchronously in the system. Now we will detail the address space usage for these additional tasks.

Beginning with *task 2*, if we designate `nr` as the task number, the starting location for *task* `nr` will be at `nr` * 64MB in linear address space. *Task 2*, for example, begins at address 2 * 64MB = 128MB in the linear address space, and the limits of code and data segments are set to 64MB. As a result, the address range occupied by *task 2* is from 128MB to 192MB, and has 64MB/4MB = 16 entries in the page directory. The code and data segments both map to the same range in the linear address space. Thus they also overlap with the same address range as illustrated in Figure 10.

After *task 2* has forked, it will call the function `execve()` to run a shell program such as bash. Just after the creation of *task 2* and before call `execve()`, *task 2* is similar to *task 1* in the three address space relationship for code and data segments except the address range occupied in linear address space has the range from 128MB to 192MB. When *task 2's* code calls `execve()` to load and run a shell program, the page entries are copied from *task 1* and corresponding memory pages are freed and

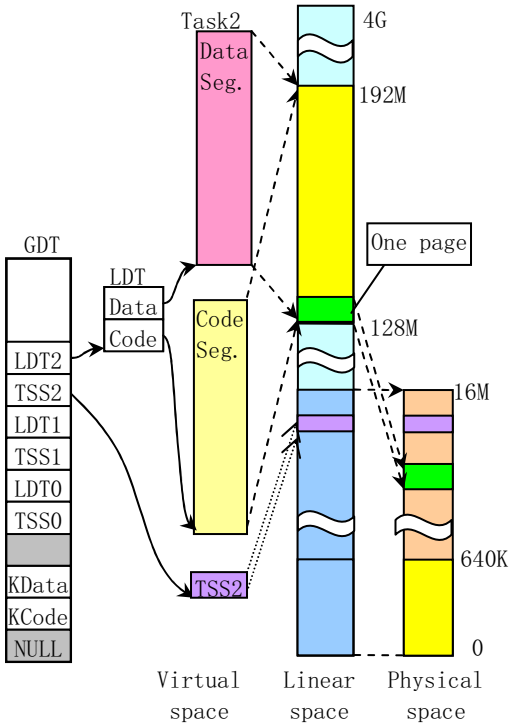

Figure 10: The relationship of the three address spaces in tasks beginning with task 2

new page entries are set for the shell program. Figure 10 shows this address space relationship. The code and data segment for *task 1* are replaced with that of the shell program, and one physical memory page is allocated for the code of the shell program. Notice that although the kernel has allocated 64MB linear space for *task 2*, the operation of allocating actual physical memory pages for code and data segments of the shell program is delayed until the program is running. This delayed allocation is called demand paging.

Beginning with kernel version 0.99.x, the usage of memory address space changed. Each task can use the entire 4G linear space by changing the page directory for each tasks as illustrated in Figure 11. There are even more changes are in current kernels.

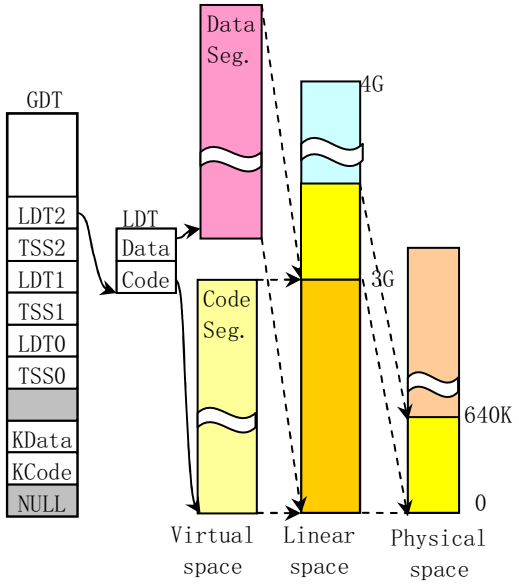

Figure 11: The relationship of the three address space for tasks in newer kernels

4 Stack Usage

This section describes several different methods used during the processing of kernel booting and during normal task stack operations. Linux 0.1x kernel uses four different kinds of stacks: the temporary stack used for system booting and initialization under real address mode; The kernel initialization stack used after the kernel enters protected mode, and the user stack for *task 0* after moving into task 0; The kernel stack of each task used when running in the kernel and the user stacks for each task except for *tasks 0 and 1*.

There are two main reasons for using four different stacks (two used only temporarily for booting) in Linux. First, when entering protected from real mode, the addressing method used by the processor has changed. Thus the kernel needs to rearrange the stack area. In addition, to solve the protection problems brought by the new privilege level on processor, we need to use different stacks for kernel code at level 0 and for user code at level 3 respectively. When a task runs in the kernel, it uses the kernel mode stack pointed by the values in `ss0` and `esp0` fields of its *TSS* and stores the task's user stack pointer in this stack. When the control returns to the user code or to level 3, the user stack pointer will be popped out, and the task continues to use the user stack.

4.1 Initialization period

When the *ROM BIOS* code boots and loads the bootsect into memory at physical address `0x7C00`, no stack is used until it is moved to the location `0x9000:0`. The stack is then set at `0x9000:0xff00`. (refer to line 61–62 in `boot/bootsect.s`). After control is transferred to `setup.s`, the stack remains unchanged.

When control is transferred to `head.s`, the processor runs in protected mode. At this time, the stack is setup at the location of `user_stack[]` in the kernel code segment (line 31 in `head.s`). The kernel reserves one 4 KB page for the stack defined at line 67–72 in `sched.c` as illustrated in Figure 12.

This stack area is still used after the control transfers into `init/main.c` until the execution of `move_to_user_mode()` to hand the control over to *task 0*. The above stack is then used as a user stack for *task 0*.

4.2 Task stacks

For the processor privilege levels 0 and 3 used in Linux, each task has two stacks: kernel mode stack and user mode stack used to run kernel code and user code respectively. Other than the privilege levels, the main difference is that the size of kernel mode stack is smaller than that of the user mode stack. The former is located

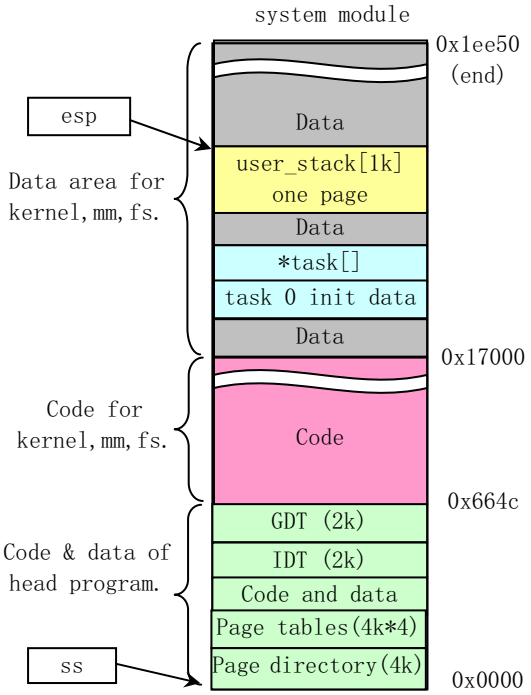

Figure 12: The stack used for kernel code after entering protected mode

at the bottom in a page coexisting with task's structure, and no more than 4KB in size. The later can grow down to nearly 64MB in user space.

As described, each task has its own 64MB logical or linear address space except for *task 0* and *1*. When a task was created, the bottom of its user stack is located close to the end of the 64MB space. The top portion of the user space contains additional environmental parameters and command line parameters in a backwards orientation, and then the user stack as illustrated in Figure 13.

Task code at privilege level 3 uses this stack all of the time. Its corresponding physical memory page is mapped by paging management code in the kernel. Since Linux utilizes the *copy-on-write*[3] method, both the parent and child process share the same user stack memory until one of them perform a write operation on the

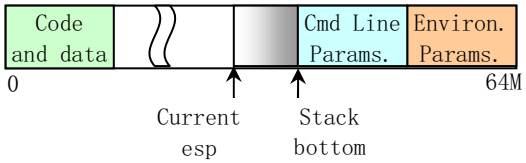

Figure 13: User stack in task's logical space

stack. Then the memory manager will allocate and duplicate the stack page for the task.

Similar to the user stack, each task has its own *kernel mode stack* used when operating in the kernel code. This stack is located in the memory to pointed by the values in `ss0`, `esp0` fields in task's *TSS*. `ss0` is the stack segment selector like the *data selector* in the kernel. `esp0` indicates the stack bottom. Whenever control transfers to the kernel code from user code, the kernel mode stack for the task always starts from `ss0:esp0`, giving the kernel code an empty stack space. The bottom of a task's kernel stack is located at the end of a memory page where the task's data structure begins. This arrangement is setup by making the privilege level 0 stack pointer in *TSS* point to the end of the page occupied by the task's data structure when forking a new task. Refer to line 93 in `kernel/fork.c` as below:

```
p->tss.esp0 = PAGE_SIZE+(long)p;
p->tss.ss0 = 0x10;
```

`p` is the pointer of the new task structure, `tss` is the structure of the task status segment. The kernel request a free page to store the task structure pointed by `p`. The tss structure is a field in the task structure. The value of `tss.ss0` is set to the selector of kernel data segment and the `tss.esp0` is set to point to the end of the page as illustrated in Figure 14.

As a matter of fact, `tss.esp0` points to the byte outside of the page as depicted in the fig-

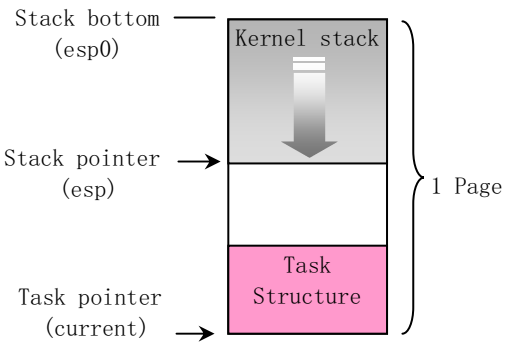

Figure 14: The kernel mode stack of a task

ure. This is because the Intel processor decreases the pointer before storing a value on the stack.

4.3 The stacks used by task 0 and task 1

Both *task 0* or *idle* task and *task 1* or *init* task have some special properties. Although *task 0* and *task 1* have the same code and data segment and 640KB limits, they are mapped into different ranges in linear address space. The code and data segments of *task 0* begins at address 0, and *task 1* begins at address 64MB in the linear space. They are both mapped into the same physical address range from 0 to 640KB in kernel space. After calling the function `move_to_user_mode()`, the kernel mode stacks of *task 0* and *task 1* are located at the end of the page used for storing their task structures. The user stack of *task 0* is the same stack originally used after entering protected mode; the space for `user_stack[]` array defined in `sched.c` program. Since *task 1* copies *task 0*'s user stack when forking, they share the same stack space in physical memory. When *task 1* starts running, however, a page fault exception will occur when *task 1* writes to its user stack because the page entries for *task 1* have been initialized as read-only. At this moment, the kernel will allocate a free page in main memory area for the stack of *task 1* in the exception handler, and map it to the location of *task 1*'s user stack in the linear space. From now on, *task 1* has its own separate user stack page. As a result, the user stack for *task 0* should be "clean" before *task 1* uses the user stack to ensure that the page of stack duplication does not contain useless data for *task 1*.

The kernel mode stack for *task 0* is initialized in its static data structure. Then its user stack is set up by manipulating the contents of the stack originally used after entering protected mode and emulating the interrupt return operation using `IRET` instruction as illustrated in Figure 15.

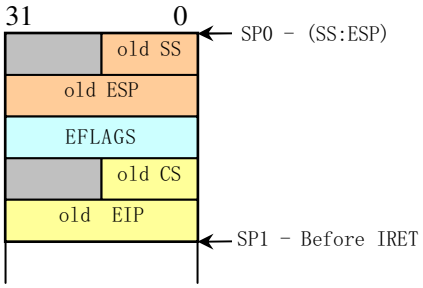

Figure 15: Stack contents while returning from privilege level 0 to 3

As we know, changing the privilege level will change the stack and the old stack pointers will be stored onto the new stack. To emulate this case, we first push the *task 0*'s stack pointer onto the stack, then the pointer of the next instruction in *task 0*. Finally we run the `IRET` instruction. This causes the privilege level change and control to be transferred to *task 0*. In the Figure 15, the old *SS* field stores the data selector of *LDT* for *task 0* (0x17) and the old *ESP* field value is not changed since the stack will be used as the user stack for *task 0*. The old *CS* field stores the code selector (0x0f) for *task 0*. The old *EIP* points to the next instruction to be executed. After the manipulation, a *IRET* instruction switches the privileges

from level 0 to level 3. The kernel begins running in *task 0*.

4.4 Switch between kernel mode stack and user mode stack for tasks

In the Linux 0.1x kernel, all interrupts and exceptions handlers are in mode 0 so they belong to the operating system. If an interrupt or exception occurs while the system is running in user mode, then the interrupt or exception will cause a privilege level change from level 3 to level 0. The stack is then switched from the user mode stack to the kernel mode stack of the task. The processor will obtain the kernel stack pointers `ss0` and `esp0` from the task's *TSS* and store the current user stack pointers into the task's kernel stack. After that, the processor pushes the contents of the current *EFLAGS* register and the next instruction pointers onto the stack. Finally, it runs the interrupt or exception handler.

The kernel *system call* is trapped by using a software interrupt. Thus an `INT 0x80` will cause control to be transferred to the kernel code. Now the kernel code uses the current task's kernel mode stack. Since the privilege level has been changed from level 3 to level 0, the user stack pointer is pushed onto the kernel mode stack, as illustrated in Figure 16.

If a task is running in the kernel code, then an interrupt or exception never causes a stack switch operation. Since we are already in the kernel, an interrupt or exception will never cause a privilege level change. We are using the kernel mode stack of the current task. As a result, the processor simply pushes the *EFLAGS* and the return pointer onto the stack and starts running the interrupt or exception handler.

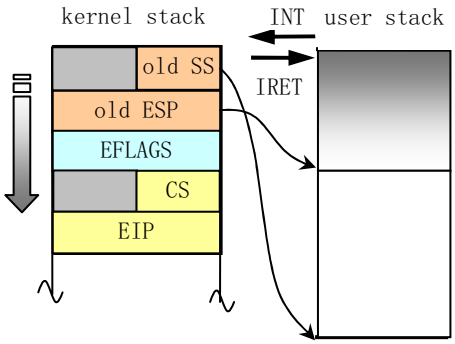

Figure 16: Switching between the kernel stack and user stack for a task

5 Kernel Source Tree

Linux 0.11 kernel is simplistic so the source tree can be listed and described clearly. Since the 0.11 kernel source tree only has 14 directories and 102 source files it is easy to find specific files in comparison to searching the much larger current kernel trees. The main `linux/` directory contains only one Makefile for building. From the contents of the Makefile we can see how the kernel image file is built as illustrated in Figure 17.

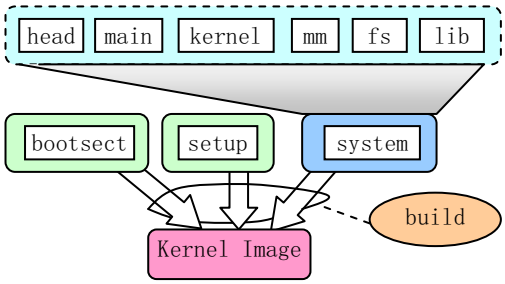

Figure 17: Kernel layout and building

There are three assembly files in the `boot/` directory: `bootsect.s`, `setup.s`, and `head.s`. These three files had corresponding files in the more recent kernel source trees until 2.6.x kernel. The `fs/` directory contains source files for implementing a *MINIX* version

1.0 file system. This file system is a clone of the traditional UN*X file system and is suitable for someone learning to understand how to implement a usable file system. Figure 18 depicts the relationship of each files in the `fs/` directory.

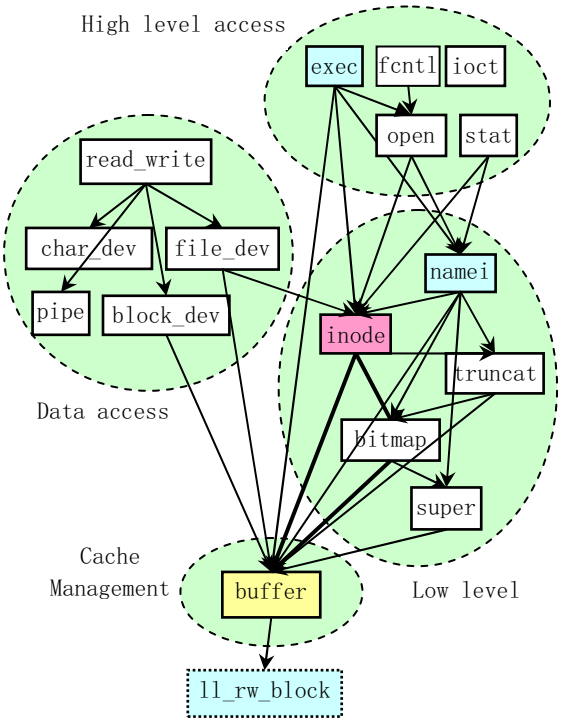

Figure 18: File relationships in fs/ directory

The `fs/` files can be divided into four types. The first is the block cache manager file `buffer.c`. The second is the files concerning with low level data operation files such `inode.c`. The third is files used to process data related to char, block devices and regular files. The fourth is files used to execute programs or files that are interfaces to user programs.

The `kernel/` directory contains three kinds of files as depicted in Figure 19.

The first type is files which deal with hardware interrupts and processor exceptions. The second type is files manipulating system calls from

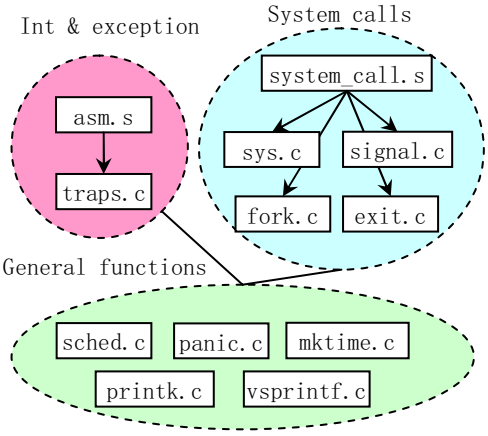

Figure 19: Files in the kernel/ directory

user programs. The third category is files implementing general functions such as scheduling and printing messages from the kernel.

Block device drivers for hard disks, floppy disks and ram disks reside in a subdirectory `blk_drv/` in the `kernel/`, thus the Linux 0.11 kernel supports only three classical block devices. Because Linux evolved from a terminal emulation program, the serial terminal driver is also included in this early kernel in addition to the necessary console character device. Thus, the 0.11 kernel contains at least two types of char device drivers as illustrated in Figure 20.

The remaining directories in the kernel source tree include, `init`, `mm`, `tools`, and `math`. The `include/` contains the head files used by the other kernel source files. `init/` contains only the kernel startup file `main.c`, in which, all kernel modules are initialized and the operating system is prepared for use. The `mm/` directory contains two memory management files. They are used to allocate and free pages for the kernel and user programs. As mentioned, the mm in 0.11 kernel uses demand paging technology. The `math/` directory only contains math source code stubs as 387 emula-

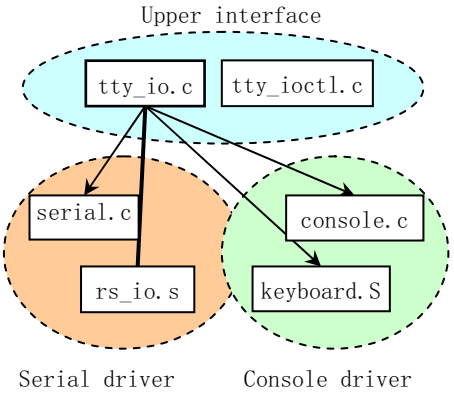

Figure 20: Character devices in Linux 0.11 kernel

tion did not appear until the 0.12 kernel.

6 Experiments with the 0.1x kernel

To facilitate understanding of the Linux 0.11 kernel implementation, we have rebuilt a runnable Linux 0.11 system, and designed several experiments to watch the kernel internal activities using the *Bochs PC emulator*. *Bochs* is excellent for debugging operating systems. The *Bochs* software package contains an internal debugging tool, which we can use to observe the dynamic data structures in the kernel and examine the contents of each register on the processor.

It is an interesting exercise to install the Linux 0.11 system from scratch. It is a good learning experience to build a root file system image file under Bochs.

Modifying and compiling the kernel source code are certainly the most important experiments for learning about operating systems. To facilitate the process, we provide two environments in which, one can easily compile the kernel. One is the original *GNU* gcc environment under Linux 0.11 system in Bochs. The other is for more recent Linux systems such as *Red Hat 9* or *Fedora*. In the former environment, the 0.11 kernel source code needs no modifications to successfully compile. For the later environment one needs to modify a few lines of code to correct syntax errors. For people familiar with *MASM* and *VC* environment under windows, we even provide modified 0.11 kernel source code that can compile. Offering source code compatible with multiple environments and providing forums for discussion helps popularize linux and the linux community with new people interested in learning about operating systems :-)

7 Summary

From observing people taking operating system courses with the old Linux kernel, we found that almost all the students were highly interested in the course. Some of them even started programming their own operating systems.

The 0.11 kernel contains only the basic features that an operating system must have. As a result, there are many important features not implemented in 0.11 kernel. We now plan to adopt either the 0.12 or 0.98 kernel for teaching purposes to include job control, virtual *FS*, virtual console and even network functions. Due to time limitations in the course, several simplifications and careful selection of material will be needed.

References

[1] Albert S. Woodhull Andrew S. Tanenbaum. *OPERATING SYSTEMS: Design and Implementation*. Prentice-Hall, Inc., 1997.

[2] Patrick P. Gelsinger John H. Crawford. *Programming the 80386*. SYBEX Inc., 1987.

[3] Robert Love. *Linux Kernel Development*. Sams Inc., 2004.

[4] M.J.Bach. *The Design of Unix Operating System*. Prentice-Hall, Inc., 1986.

[5] Linus Torvalds. LINUX – a free unix-386 kernel. October 1991.