Linux Support for NUMA Hardware

Matthew Dobson, Patricia Gaughen, Michael Hohnbaum
IBM LTC, Beaverton, Oregon, USA
colpatch@us.ibm.com, gone@us.ibm.com, hohnbaum@us.ibm.com
Erich Focht
NEC HPCE, Stuttgart, Germany
efocht@hpce.nec.com

Abstract

New large CPU-count machines are being designed with non-uniform memory architecture (NUMA) characteristics. The 2.5 Linux® kernel includes many enhancements in support of NUMA machines. Data structures and macros are provided within the kernel for determining the layout of the memory and processors on the system. These enable the VM subsystem to make decisions on the optimal placement of memory for processes. This topology information is also exported to user-space via `sysfs`. In addition to items that have been incorported into the mainline Linux kernel, there are NUMA features that have been developed that continue to be supported as patchsets. These include NUMA enhancements to the scheduler, multipath I/O, and a user-level API that provides user control over the allocation of resources in respect to NUMA nodes.

1 Introduction

1.1 Non-Uniform Memory Architecture

Demand for greater computing capacity has lead to the increased use of multi-processor computers. Most multi-processor computers are considered Symmetric Multi-Processors (SMP) as each processor is equal and has equal access to all system resources (e.g., memory and I/O busses). SMP systems generally are built around a system bus that all system components are connected to, and is used to communicate between the components. As SMP systems have increased their processor count, the system bus has increasingly become a bottleneck. One solution that is gaining in use by hardware designers is Non-Uniform Memory Architecture (NUMA).

NUMA systems co-locate a subset of the system's overall processors and memory into nodes, and provide a high speed and high bandwidth interconnect between the nodes, see Figure 1. Thus there are multiple physical regions of memory, but all memory is tied together into a single cache-coherent physical address space. The resulting system has the property such that for any given region of physical memory, some processors are closer to it than other processors. Conversely, for any processor, some memory is considered local (i.e., it is close to the processor) and other memory is remote. Similar characteristics may also apply to the I/O busses—that is, I/O busses may be associated with nodes.

While the key characteristic of NUMA systems is the variable distance of portions of memory from other system components, there are numerous NUMA system designs. At one end of the spectrum are designs where all nodes are symmetrical—they all contain memory, CPUs, and I/O busses. At the other end of the spectrum are systems where there are different types of nodes—the extreme case being separate CPU nodes, memory nodes, and I/O nodes. All NUMA hardware designs are characterized by regions of memory being at varying distances from other resources, thus having different access speeds.

To maximize performance on a NUMA platform, Linux must take into account the way the system resources are physically laid out. This includes information such as which CPUs are on which node, which range of physical memory is on each node, and what node an I/O bus is connected to. This type of information describes the topology of the system.

There are several challenges Linux must address to provide NUMA support. These include:

- discovery and internal representation of the system topology
- minimization of traffic over the interconnect between nodes
- localization of memory references
- fair access to locks
- I/O locality
- synchronization of time between nodes
- the location of low address memory (e.g., memory with physical address under 4 GB) all on the first node, on i386™ processor (and potentially other 32-bit processor architectures) based machines
- scheduling of processes and groups of processes on the same node.

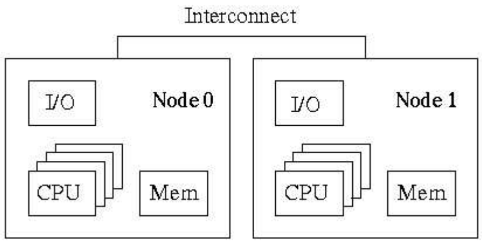

Figure 1: Simple view of NUMA system

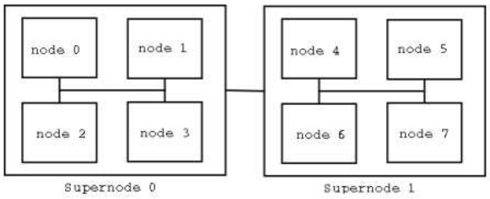

Figure 2: NUMA system with supernodes

Linux running on a NUMA system obtains optimal performance by keeping memory accesses to the closest physical memory. For example, processors benefit by accessing memory on the same node (or closest memory node), and I/O throughput gains by using memory on the same (or closest) node to the bus the I/O is going through. At the process level, it is optimal to allocate all of a process's memory from the node containing the CPU(s) the process is executing on. However, this also requires keeping the process on the same node.

This paper looks at how Linux addresses these NUMA challenges, focusing on the NUMA support that is available in the 2.5 development kernel. In addition, some discussion is included about additional NUMA support that is under development for future Linux releases.

1.2 Hardware Implementations

There are many design and implementation choices that result in a wide variety of NUMA platforms. This variety creates additional challenges for the Linux OS developer, as a single solution is desired to support the many types of NUMA hardware. This section discusses hardware implementations, and provides examples and descriptions of NUMA hardware implementations.

Types of nodes

The most common implementation of NUMA systems consists of interconnecting symmetrical nodes. In this case, the node itself is an SMP system that has some form of high speed and high bandwidth interconnect linking it to other nodes. Each node contains some number of processors, physical memory, and I/O busses. Typically, there is a node-level cache. This type of NUMA system is depicted in Figure 1.

A variant on this design is to only put the processors and memory on the main node, and then have the I/O busses separate. Another design option is to have separate nodes for processors, memory, and I/O busses which are all interconnected.

It is also possible to have nodes which contain nodes, resulting in a hierarchical NUMA design. This is depicted in Figure 2.

Types of interconnects

There is no standardization of interconnect technology. More relevant to Linux is the topology of the interconnect. NUMA machines exist that use the following interconnect topologies:

- ring topology – each node is connected to the node on either side of it. Memory access latencies can be non-symmetric; that is, accesses from node A to node B might take longer than accesses from node B to node A.
- crossbar interconnect – all nodes connect into a common crossbar.
- point to point – each node has a number of ports to connect to other nodes. The number of nodes in the system is limited to the number of connection ports plus one, and each node is directly connected to each other node.
- mesh topologies – more complex topologies that, like point to point topologies, are built upon each node having a number of connection ports. But unlike point to point topologies, there is not a direct connection between each node. Hypercube and torus topologies are examples of mesh topologies.

The topology provided by the interconnect affects the distance between nodes. This distance needs to be accounted for when Linux is making resource placement decisions.

Latency ratios

One important measurement for determining the “NUMA-ness” of a system is the latency ratio. This is the ratio between memory latency for on-node memory access versus off-node memory accesses. Depending upon the topology of the interconnect, there might be multiple off-node latencies. This latency can be used to analyze the cost of memory references to different parts of the physical address space, and thus be used to influence decisions affecting memory usage.

Specific NUMA implementations

Several hardware vendors are building NUMA machines that run the Linux operating system. This section briefly describes some of these machines, but is not an all inclusive survey of the existing implementations.

One of the earlier commercial NUMA machines is the IBM® NUMA-Q® box. This machine is based upon nodes which contain 4 processors (i386), memory and PCI busses. Each node also contains a management module to coordinate booting, monitor environmentals, and communicate with the system console. The nodes are interconnected using a ring topology. Up to 16 nodes can be connected for a maximum of 64 processors and 64 GB of memory. Remote to local memory latency ratios range from 10:1 to 20:1. Each node has a large remote cache which helps compensate for the large remote memory latencies. Much of the Linux NUMA development has been on these boxes due to their availability.

NEC builds NUMA boxes using Intel™ Itanium™ processors. The most recent system in this line is the NEC TX7. The TX7 supports up to 32 Itanium2 processors in nodes of 4 processors each. The nodes are connected by a crossbar and grouped in two supernodes of four nodes each. The crossbar provides fast access to non-local memory with low latency and high bandwidth (12.8 GB/second per node). The memory latency ratio for remote to local memory in the same supernode is 1.6:1. The remote to local memory latency ratio for outside the supernode is 2.1:1. There is no node level cache. I/O devices are connected through PCI-X busses to the crossbar interconnect and thus are all the same distance to any CPU/node.

The large IBM xSeries® boxes use Intel processors and the IBM XA-32™ chipset. This chipset provides an architecture that supports four processors, memory, PCI busses, and three interconnect ports. These interconnect ports allow point to point connection of up to 4 nodes for a 16 processor system. Also supported is a connection to an external box with additional PCI slots to increase the I/O capacity of the system. The IBM x440 is built on this architecture with Intel Xeon™ processors.

2 Linux NUMA Support

The basic infrastructure for supporting NUMA hardware has been incorporated into the Linux 2.5 development kernel. This support includes topology discovery and internal representation, memory allocation, process scheduling, and timer support. In addition there are kernel extensions in support of NUMA that are not yet included in the mainline kernel, but are being maintained as separate patchsets. These include NUMA-aware multi-path I/O, application-level directed binding of memory to nodes, and scheduler extensions.

2.1 CONFIG Options

Most NUMA support options within the kernel are enabled by the CONFIG_NUMA option. This includes the scheduler extensions, NUMA memory allocation, and topology support. There is also an option, CONFIG_DISCONTIGMEM, that is used for enabling a portion of the NUMA memory support.

2.2 Linux Architecture Support

Linux supports many types of processors and many hardware architectures. Within Linux, when reference is made to an architecture it typically refers to the processor type (e.g., i386, Power4™, alpha, etc.). Subarchitectures are used to refer to a substantially different hardware implementation of a particular architecture. Also, within an architecture there are platforms which are an implementation of the architecture.

For example, the x440 is part of the i386 architecture within Linux. However, the x440 is also a unique subarchitecture within the i386 architecture. Another example is the IA64 architecture which has DIG-64 and HP platforms, and SGI-SN1 subarchitecture.

Throughout this paper references are made to architecture which are more correctly subarchitectures. References to architectures are meant to refer to a specific hardware implementation of a NUMA system.

3 Topology

There are performance penalties involved in accessing hardware devices (CPUs, memory, disks, network cards, etc.) that are remote to the currently executing CPU. These can be significantly reduced by having a knowledge of the system's topology, and using that information to make good scheduling, allocation, migration, and I/O decisions. Topology information is crucial to the kernel for making good decisions on a NUMA machine, but this information is also important to some user-space applications as well.

Topology information is currently used in the kernel to schedule processes and allocate memory. This has contributed to performance improvements for NUMA architectures throughout the 2.3, 2.4, and 2.5 kernel series.

3.1 Topology Elements

The topology of a system includes all hardware components that make up the system. However, for the context of this paper, topology is restricted to those physical elements which are directly affected by the NUMA characteristics of the system. These elements are nodes, processors, memory, and I/O busses. Physical components not considered here consist of the actual I/O devices which are connected into the system through the I/O busses.

CPUs provide the computing power in the system. The location of the individual CPUs in the overall topology is extremely important for scheduling decisions. The current in-kernel topology API exposes 4 CPU related functions: `cpu_to_node()`, `node_to_cpumask()`, `node_to_first_cpu()`, and `pcibus_to_cpumask()`. Information about these functions is provided in the next section. The two-node system in Figure 1, contains 8 CPUs, 4 on node 0, and 4 on node 1.

A Memory Block, or memblk for short, represents a physically contiguous piece of memory. It is typically used to represent all the memory in a particular memory bank on a particular node. A node is permitted to have either 0 or 1 memory blocks. For example, in Figure 1, we have a two-node system. Its total memory is split into two memblks, one on node 0, one on node 1. In UP/SMP systems, all the memory in the system is represented by a single memory block.

I/O bus elements represent physical I/O busses in the underlying system. These are important elements for operations like scheduling disk I/O, networking tasks, or any I/O intensive process. By utilizing knowledge about I/O locality, processes can ensure they run efficiently by constraining themselves to CPUs and memory blocks on or close to the I/O bus they utilize.

When discussing NUMA, the word node is often overloaded. For the purposes of a Linux topology discussion, a node is solely an abstract container. Nodes are not meant to represent any physical element of the underlying architecture. All elements, including nodes themselves, are children of a node in the system. The node element is designed to be a medium through which queries can be made. For example, in Figure 1, we see a simple illustration of a two node system. Each node has 4 CPUs in it, as well as a block of memory. To find out which memory block is closest to CPU 3, a process can determine that CPU 3 is a child of node 0, and that memblk 0 is also a child of node 0. Thus, for efficiency, a process running on CPU 3 would want to be sure that its memory is allocated from memblk 0.

On some systems nodes can be nested, as seen in Figure 2. This is important with things like hyperthreading and multi-core processors becoming more available, which can be easily represented as a small node with processors, but no memory or I/O busses. Another usage of nested nodes is for hierarchical NUMA systems, such as the NEC TX7, which is built with supernodes that contain nodes.

There exists a strong possibility that there will be a need to introduce new topology elements in the future. Due to the simplicity of the design of the topology subsystem, adding new elements is a straightforward procedure. As long as there is a parent-child relationship between the new element and nodes, the new element should drop right in to the existing infrastructure.

3.2 Topology Kernel Functions

The following is a list of topology-related kernel calls that form the basis of the current topology framework. Along with the description of the call is a default return value for the non-NUMA case. Architecture specific definitions of these kernel calls are provided for each architecture that has NUMA topology support. Most architectures simply use the default asm-generic/topology.h version, usually because the architecture does not support NUMA.

- `cpu_to_node(int cpu)` – Returns the number of the node containing CPU `cpu`. For non-NUMA, defaults to 0.

- `memblk_to_node(int memblk)` – Returns the number of the node containing memory block `memblk`. For non-NUMA systems, defaults to 0.

- `parent_node(int node)` – Returns the number of the node containing node `node`. If the node number returned is `node`, `node` is a top-level node. For non-NUMA, defaults to 0.

- `node_to_cpumask(int node)` – Returns a bitmask of the cpus on Node `node`. For non-NUMA, defaults to `cpu_online_map`.

- `node_to_first_cpu(int node)` – Returns the number of the first CPU on Node `node`. For non-NUMA, defaults to the lowest-numbered CPU in the system.

- `node_to_memblk(int node)` – Returns the number of the Memory Block, if any, on Node `node`. For non-NUMA, defaults to 0.

- `pcibus_to_cpumask(int bus)` – Returns a bitmask of the CPUs closest to PCI bus `bus`. For non-NUMA, defaults to `cpu_online_map`.

- `numa_node_id()` – Returns the number of the node containing the current CPU. For non-NUMA systems, defaults to 0.

3.3 Closest Element versus Distance Matrix

The current implementation of the topology system is very helpful if the caller is looking for information relating to the closest element. This choice was made primarily because this made the code small and compact, but also because the majority of consumers of this information simply want the closest element. The other option, and possible future method, is to use a distance matrix approach. Using this approach, each machine type would build a latency matrix representing the distance from element X to element Y. The distance matrix approach allows us much more flexibility when retrieving information, and much more complicated queries can be satisfied. We decided against this approach, however, because it was determined that the added complexity did not offer that much benefit, and likely would have few consumers. However, in hierarchical NUMA systems this type of approach is more likely to be required for optimal performance benefits.

Exporting Topology Information to User Space

Topology information is important to multi-threaded user-space applications. With a large parallel NUMA machine, threads can be coordinated across, but more importantly within, nodes. This can yield significant speedups over a standard SMP version run on the same machine. There is also a proposal to facilitate the sharing of memory regions by establishing bindings for those regions. This would allow multi-threaded applications to specify memory blocks close to the set of CPUs the group of threads is executing on, and guarantee pages faulted into specific memory regions (likely shared) come from those memory blocks.

User-space applications currently have access to NUMA topology information through `sysfs`. The information is laid out following the normal directory structure. Node directories contain CPU and memory block directories, as well as other node directories on machines that take advantage of nested nodes. Each of these directories have files in them, with those files containing various bits of information that can be read and/or written. For example, the nodes contain a `cpumap` file that contains a bitmap of CPUs on that node.

Currently I/O busses are not represented in the topology directory of `sysfs`. Adding this is a future work item.

There is currently no way for a user-space application to determine the CPU it is currently executing on. This data is inherently volatile, as it requires going into kernel-space to get it, and while returning from the kernel it is possible for the process to be switched to another CPU thus invalidating the information that is about to be returned. There are other ways to give user-space access to this information; for example, by mapping a page that is shared between user-space and kernel-space and having the kernel store the CPU that the process is currently executing on at a set location within that page.

4 Memory

This section describes some of the issues encountered during the development of 2.5 to support NUMA memory allocation, and the Linux implementation to address the issues. The purpose of this section is not to provide an in depth look at the Linux memory subsystem—there is documentation available on the net for that [1].

4.1 Discontiguous Memory Support

Each architecture needs to describe its physical layout to the kernel. This includes specifying which address ranges belong to which node, and whether there are holes in between those ranges (a hole is a physical address range for which there is no real memory). CONFIG_ DISCONTIGMEM is currently used to represent a solution to some of these problems. The name of the config option is a bit of a misnomer because the memory may not be discontiguous. In the case of the IBM x440 the memory is contiguous, except for a large hole on the first node.

The core data structure for describing the physical layout is the `pg_data_t`. This data structure currently has a 1:1 mapping to nodes. For each node in the system, there exists one `pg_data_t`. The `pg_data_t` describes the start and end of memory for the node, a pointer to the zones for the node, and related information. Support for multiple `pg_data_t`'s have been in the kernel since 2.4 (although several fixes and optimizations have occured since then). It is up to each architecture to populate these correctly for their system.

The config option, CONFIG_DISCONTIGMEM turns on the functionality for creating multiple `pg_data_t`'s. CONFIG_NUMA turns on the code (i.e. scheduling decisions, allocation decisions) that makes use of the per node `pg_data_t`'s.

Zone Normal Memory on 32-bit Systems

During the setup of memory in system initialization a special allocator is used—the bootmem allocator. This allocator only allocates memory out of what will later become ZONE_NORMAL. Once memory is setup the bootmem allocator is no longer used. The `bootmem_data_t` represents the address range used by the bootmem allocator. The `pg_data_t` for the node containing the memory has a pointer to the `bootmem_data_t` (`bdata`).

One of the early issues ran into during the development of i386 CONFIG_DISCONTIGMEM support was the idea that not all `pg_data_t`'s will have a portion of ZONE_NORMAL, or a `bootmem_data_t`. On i386, ZONE_NORMAL is limited to the first 896 Mb of physical memory because of limitations of the 32-bit architecture. So, a system populated with 1GB of RAM per node will only have a ZONE_NORMAL (and ZONE_DMA) on node 0, the rest of the nodes will only contain ZONE_HIGHMEM. Because the slab allocator only allocates memory from ZONE_NORMAL, and the kernel uses the slab allocator to allocate memory for internal data structure, most kernel related memory will be on node 0. This also means that during early boot only the first node's memory will be used by the bootmem allocator.

Two changes were made to make ZONE_NORMAL only on node 0 work: (1) a dereferencing a null pointer bug was fixed in `__alloc_pages()` that didn't check that the node had ZONE_NORMAL before using it. (2) `alloc_bootmem_node()` and friends needed to be made to only use the first node's `pg_data_t`, because the other nodes would not have a `bootmem_data_t`. Since `alloc_bootmem_node()` is architecture independent, it was important to not put arch specific requirements in the code, so the changes were made in the header files. Thus the creation of CONFIG_HAVE_ARCH_BOOTMEM_NODE.

Page to Node Translation

Finding the node a memory address belongs to is used throughout the kernel. The core routine doing the translation from address to node id, is `pfn_to_nid()`. Because it is called often, it is important that the translation is fast. On some architectures, the physical layout is such that the first 64GB of address space belongs to the first node (whether or not there really is 64GB of RAM), second 64GB for the second, and so on. This makes the algorithm for figuring out what node an address belongs to very simple, and very fast: if the address is between 0-64GB it's node 0. But on i386 NUMA architectures that have been tested, it is not as clear. Because the memory is contiguous, there isn't a nice GB to node translation. The solution was to create a mapping of addresses to node IDs on 256MB address ranges. The map is created during memory setup and allows for fast translations.

4.2 Node Aware Memory Allocation

One of the features enabled by CONFIG_NUMA is that the system makes NUMA-aware memory allocation decisions. The current policy is when memory is allocated, the kernel tries to allocate from the local node. If that fails, the allocator will allocate from the other nodes. The exception is in the case of a system with ZONE_NORMAL only on the first node; in an i386 NUMA box for example, memory allocated from ZONE_NORMAL will only be allocated from the first node.

The allocation policies only apply to new memory. Should a process migrate across node, the memory related to the process will not be migrated. Although if the memory is swapped out, when the pages are swapped back in they will be swapped to the node the process has migrated to. One thing to keep in mind, is that migrating the memory is expensive; however, if a page is being accessed often, it would be a performance benefit to move it to the local node.

When the kernel is attempting to allocate memory, and the system is low on pages, `kswapd` will be woke to address the low memory issue. Without NUMA awareness `kswapd` may free up lots of memory by swapping pages out, but it may not make available memory local to the node that is in need of memory. The solution was to make `kswapd` per node. `kswapd` monitors the memory on the local node. When memory needs to be freed, it's freed from the local node. Rmap [3] made this change to `kswapd` possible, because of the ability to find the virtual address(es) associated with a physical address (local to the node).

4.3 Node Local Kernel Data Structures

For kernel data structures that are frequently accessed and have node specific information, it makes sense to have their data structures in node local memory. On most architectures, when the bootmem allocator is available, it is possible to allocate memory on a specific node through the use of `alloc_bootmem_node()`. However, on i386 the bootmem allocator only allocates from node 0, so `alloc_bootmem_node()` doesn't work for allocating per node and all memory is allocated from node 0. Because of the limited lifespan of the bootmem allocator, `alloc_bootmem_node()` is not a complete solution. No other generic mechanism is available at this time for allocating data structures on a per node basis. A possible solution for a generic mechanism is currently in development by Bill Irwin [2].

To work around this 32-bit architecture limitation, for the specific case of the `mem_map` and `pg_data_t`, Martin Bligh has successfully made these two types of data structures reside in node local memory. That is, these structures are located on the node for the memory that they are describing.

The first phase was to make `mem_map` per node. This was done by reserving pages at the top of the node and decrementing the size of the address space by the size of `mem_map`, and then making use of that reserved space when `mem_map` was set up. Nothing special had to be done for node 0, because it is where the bootmem allocator gets it memory for node 0's data. So, for node 0 the the normal bootmen allocator can be used. Phase two was to make pg_data_t per node. This was done using the same method as for the mem_map.

4.4 Replication

Since kernel text is read-only on production systems, there is little downside to replicating it and placing a copy on each node. This does consume extra memory, but kernel text is relatively small and memories of NUMA machines relatively large. Kernel-text replication requires special handling from debuggers when setting breakpoints and from /dev/mem in cases where users are sufficiently insane to modify kernel text on the fly. In addition, kernel-text replication means that there is no longer a single "well-known" offset between a kernel-text virtual address and the corresponding physical address.

This functionality is present in some architectures (e.g., sgi-ip27) in the 2.4 kernel. Also, the IA64 discontigmem patch provides kernel text replication support for IA64. It is not likely to show up in the i386 tree because of the limitations of the architecture.

4.5 Memory Binding

As mentioned in other sections of this paper, writing code to run on a NUMA machine can require changes to take advantage of the interesting hardware configurations these machines offer. Memory Binding is one API that we feel large user-space programs will be able to use to make significant performance improvements for NUMA. The idea behind Memory Binding is that processes can selectively bind ranges of their virtual memory space to particular blocks of memory, according to different allocation policies. For example, a large database program that has many threads could bind its threads to CPUs on two nodes, and also bind a large section of its shared memory to the memory that belongs on those two nodes. By setting a policy that enforces an equal distribution of pages, the database could be sure that all its shared pages are at least on the same set of nodes as its processes, and that the memory is evenly spread across those nodes. The Memory Binding API is available as a patch from:

```
http://www-124.ibm.com/linux/
patches/?patch_id=753
http://www-124.ibm.com/linux/
patches/?patch_id=754
```

5 NUMA scheduler

5.1 Introduction

As explained in the introductory section, accessing the memory of a remote node implies taking penalties in memory access latency and bandwidth. Therefore, it is desirable to keep processes on or near the node on which their memory (or most of it) is allocated.

The old (pre 2.5) Linux scheduler wasn't aware of the NUMA structure of a machine. Processes could migrate to any CPU in the system if the CPU was less loaded. On NUMA machines with many CPUs, two scalability problems were additionally limiting the performance: the CPUs were competing for the runqueue lock and the time needed for selecting the task to be scheduled next was linearly growing with the length of the runqueue.

The scalability problems were mostly solved by the $O(1)$ scheduler[4]. Like other approaches [5, 6] it implements per CPU runqueues[1] avoiding the lock starvation problem. Additionally it implements an $O(1)$ search algorithm for the task to be scheduled next. The scalability problems for SMP machines were solved, but the $O(1)$ scheduler was not

[1] There are actually two runqueues per priority level per CPU.

NUMA-aware, either. An idle CPU could easily steal a task from the node where its memory was allocated letting it run with degraded memory performance.

5.2 NUMA scheduler approaches

The first notable Linux NUMA scheduler was the one Andrea Arcangeli made on top of the old Linux scheduler[7]. It implemented per node runqueues and scheduled across node boundaries only after failing to find an optimal CPU within the same node. Being built on top of the old Linux scheduler this approach suffered of very similar scalability limitations.

Another approach was the extension of the IBM MQ scheduler [5] to allow rescheduling only inside pools of CPUs [8]. A loadbalancing module was added which allowed periodic rebalancing across the pool boundaries.

The first NUMA scheduler on top of the $O(1)$ scheduler was designed and implemented by Erich Focht [9]. Tasks were assigned a home node at creation time (either at `fork()` or at `exec()`, selectable for each task), allocated their memory on (or near) the home node, and were attracted by the home node CPUs. The tasks were node-affine. Because the scheduler changes were too complex for inclusion into the 2.5 kernel baseline, Erich Focht, Michael Hohnbaum, Martin Bligh, and Andrew Theurer collaborated with the target to strip down and rewrite the node-affine scheduler to a slim NUMA variant acceptable for inclusion. The result was included into the 2.5.59 kernel and is described in the following section.

5.3 NUMA scheduler in the 2.5 kernel: implementation

When stripping down the node-affine scheduler, the goals were to keep the changes to the $O(1)$ scheduler as small as possible, and to add NUMA awareness by making it difficult for a task to change the node while trying to keep the node load well-balanced. This was achieved by three patches.

Initial load balancing at `exec()`

The NUMA support for the memory subsystem described in section 4 ensures that memory pages are allocated from the node on which the page-faulting task is running[2]. Normally processes allocate most of their memory right after creation; therefore, the choice of the initial node and CPU is very important for getting well-balanced nodes.

Initial load balancing implies some overhead because it involves scanning the current node loads and determining the best CPU on which the freshly created task should be scheduled. This can be done either at `fork()` or at `exec()`. Doing it at `fork()` (and `clone()`) has the advantage that multi-threaded jobs lead to a balanced machine as well. This might be desireable on machines with good latency ratios between the nodes. On the other hand, every small and short living thread picks up the initial balancing overhead, unnecessarily migrates pages to other nodes by copy on write (COW), and finds a cold instruction cache.

Doing initial balancing at `exec()` avoids the COW problem because all pages are dropped at that stage. Short-living threads which don't `exec()` benefit from a warm instruction cache. But long running memory intensive multi-threaded programs might pick up performance penalties due to the unbalanced nodes.

The implementation adds the array `static atomic_t node_nr_running[MAX_NUMNODES]` to keep track of the num-

[2] If the current node has sufficient free memory.

ber of tasks running on each node. `kernel/sched.h` is extended by three functions:

- `sched_best_cpu()`: Finds the least loaded CPU on the least loaded node using the current runqueue lengths.
- `sched_migrate_task()`: Migrates a task to a certain CPU.
- `sched_balance_exec()`: Called by `do_execve()`, it moves the current task to the least loaded CPU.

Intra-node load balancing

The `load_balance()` and `find_busiest_queue()` functions of the $O(1)$ scheduler have been modified to restrict the search for the busiest CPU to the set given by the new `cpumask` argument. In the NUMA scheduler this mask uses topology information and usually limits the search to the current node. To be precise: all calls to `load_balance()` except the one from the timer interrupt are balanced only within the current node.

Cross-node load balancing

Even with a perfect initial load balancing, a machine can easily end up with poorly balanced nodes, e.g. nodes with more running tasks than available CPUs and idle nodes. In such cases, it is preferrable to use the idle CPUs for doing real work even if the tasks running on them need to access memory from other nodes. It is better if a task runs slower on a remote node instead of waiting for the CPU on its own node. The cross-node balancing occurs periodically during the timer interrupt, with the current settings (kernel 2.5.67) this means: an idle CPU will try node-rebalancing every 5 ticks (5ms on a `HZ=1000` system); a busy CPU will do it every 20s.

There continues to be debate as to the frequency of the busy rebalance, with some believing the busy rebalance is occuring much too infrequently. It is felt that the current frequency, while showing advantages on simple benchmarks is not optimal for real world conditions.

Node rebalancing is achieved by a change in `scheduler_tick()` and three additional routines:

- `rebalance_tick()`: Decides when to balance within the node and when across the node boundaries. In the later case it will first try an intra-node rebalance.
- `balance_node()`: Calls `load_balance()` with `cpumask` set to the least-loaded node plus the current CPU.
- `find_busiest_node()` Finds busiest node and uses a geometrically decaying weight for the load measure: $load_t = load_{t-1}/2 + nr\_node\_running_t$. This flattens out sudden load peaks.

5.4 Current limitations and future developments

The NUMA scheduler currently implemented in the Linux kernel is far from being complete. The degree of NUMA-awareness of the scheduler gives clear performance boosts for “simple” load situations like parallel kernel compiles or an arbitrary but more or less constant number of similar and long running `exec`’d processes. The limitations are shown in environments with long running jobs, and in suddenly varying loads, or with long running multithreaded applications like OpenMP.

The stripped down version of the node-affine scheduler strongly reflects the influence of former IBM work [8]. Some of the useful features of [9] were lost, among them the capability of a process to remember the node on which its memory resides and to return to that node. A scheduler with such features is in production on NEC TX7 IA64 servers and shows significant benefits in production environments. Thus possible extensions of the 2.5 NUMA scheduler could be:

- An option to allow particular tasks to initially balance their children at `fork()`.
- A method of keeping track of where one task's memory is.
- A method of pushing tasks to the node where most of their memory resides.

6 Locking

In contrast to much of the other NUMA work, NUMA-aware locking is not about making a per-node lock, but rather it is about preventing lock starvation on highly-contended locks. Lock starvation occurs when the contention on a given lock is so high that by the time a CPU releases the lock, at least one other CPU on that same node is requesting it again. NUMA latencies mean that these local CPUs can acquire the lock when it becomes available faster than remote CPUs can. On some architectures, the CPUs on the node where the lock is located can monopolize the lock, completely starving CPUs on other nodes.

The best solution is to reduce lock contention, but NUMA-aware locks can be an interim fix while the locking design is reworked.

Two fair locking primitives are:

- `mcs locks` [10]

  mcs locks are queued locks. The primitive enforces fairness because requesters are queued. The queuing ensures that the local CPU does not have an advantage on getting the lock. It's first come, first served.

- `NUMA-aware locks` [11]

  NUMA-aware locks enforce fairness by using a round-robin system amongst nodes waiting for a lock. The implementation was written so that the fairness algorithm could be modified to fit the need. This means, if round robin proves inefficient, another method can be inserted.

The work in the area of NUMA-aware locking is currently not active. As previously mentioned, the Linux solution for a highly-contended lock is to break the lock up, and so far lock starvation has not been seen to be a problem. Therefore a need for a NUMA-aware lock has not been established.

7 I/O

As with many aspects of writing software to run efficiently on NUMA platforms, I/O code benefits from fine-tuning for these machines. The following section goes into more detail about: why I/O subsystems require NUMA considerations, the current state of Linux support of I/O on NUMA hardware, and where it might be going.

7.1 I/O Locality

As discussed in the topology section, on NUMA machines I/O busses are ususally spread across nodes. When scheduling I/O we attempt to ensure that the memory being used for the I/O is close to the specific I/O bus we are using. Cross-node I/O requests suffer a performance penalty when compared to I/O requests that are constrained to a single node. Cross-node I/O travels across the node interconnect busses and has the potential to consume interconnect bandwidth, thus degrading the performance of other processes. If the memory buffers used for I/O are physically located in memory far from the I/O bus, there will also be delays for cross-node memory access. Ideally, the requesting process executes on a CPU on the node with the memory and I/O bus, thus eliminating any inter-node accesses.

7.2 Multi-Path I/O

While Multi-Path I/O, or MPIO for short, is not a new concept, it can be a particularly powerful tool on a NUMA platform. MPIO involves using multiple I/O adaptors (i.e., SCSI cards, network cards) to gain multiple paths to the underlying resource (i.e., hard disks, the network), thus increasing overall bandwidth. On SMP platforms, potential speedups due to MPIO are limited by the fact that all CPUs and memory typically share a bus, which has a maximum bandwidth. On NUMA platforms, however, different groups of CPUs, memory, and I/O busses have their own distinct busses. This allows potentially achieving larger aggregrate I/O throughput by allowing each node to independently reach its maximum bandwidth. An ideal MPIO on NUMA setup consists of an I/O card (SCSI, Network, etc.) on each node connected to every I/O device, so that no matter where the requesting process runs, or where the memory is, there is always a local route to the I/O device. With this hardware configuration, it is possible to saturate several PCI busses with data. This is even further assisted by the fact that many machines of this size will be using RAID or other MD devices, thus further increasing the potential bandwidth by using multiple disks.

There is a patch, currently against 2.5.59, that implements MPIO for the SCSI Mid-Layer in Linux. The SCSI layer is in the midst of many changes right now, some of which affect algorithms this patch was based on. This patch [12] is maintained by Patrick Mansfield, and is being discussed in vastly more detail at another presentation at OLS.

7.3 Interrupt Routing and Balancing

Interrupt handling is another area where ignoring NUMA locality issues can be costly. When dealing with interrupts, it is important that they are handled locally. Some architectures and APIC setups prevent interrupts from being handled remotely by their design, but for those that don't, we must make sure that interrupts are kept local. What this means is that if, for example, an I/O device raises an interrupt, it should be handled by a CPU on the same node as the I/O device. At the same time, we don't want every interrupt occurring on a particular node to be handled by the same CPU. Currently the Linux kernel takes advantage of the balance IRQ functionality, which changes the destination of individual IRQs to a different CPU after a certain number of ticks. This code is not aware of NUMA topology, though, and thus may sometimes make poor IRQ destination decisions. There is significant work to be done still in this area for NUMA support.

On some chipsets, IRQ balancing is provided by the hardware, for example the 460GX realted chipsets (used by the NEC TX7). This chipset provides either a fixed redirection or can redirectable within a target node based on priorities.

8 Timers

On UP systems, the processor has a time source that is easily and quickly accessible, typically implemented as a register. On SMP systems, the processors' time source is usually synchronized as all of the processors are clocked at the same rate, and thus synchronization of the time register between processors is a straight forward task.

On NUMA systems synchronization of the processors' time source is not practical as not only does each node have its own crystal providing the clock frequency, but there tend to be minute differences in the frequencies that the processors are driven at which thus leads to time skew.

On multi-processor systems it is imperative that there is a consistent system time. Otherwise time stamps provided by different processors cannot be relied upon for ordering and if a process is dispatched on a different processor it is possible that there can be unexpected jumps (backward or forward) in time.

Ideally, the hardware provides one global time source with quick access times. Unfortunately, global time sources tend to require off-chip access and often off-node access which tend to be slow. Clock implementations are very architecture specific, with no clear leading implementation amongst the NUMA platforms. On the x440, for example, the global time source is provided by node 0 and all other nodes must go off-node to get the time.

In Linux 2.5, the i386 timer subsystem has an abstraction layer that simplifies the addition of a differnt time source provided by a specific machine architecture. For standard i386 architecture machines, the TSC is used which provides a very quick time reference. For NUMA machines, a global time source is used (e.g., on the x440 the cyclone timer).

9 Summary

Much work has been done to provide NUMA support for the Linux kernel. At this point, the basic infrastructure is in place. Performance testing has shown measureable improvements, though they tend to be widely variable dependent upon the workload and the NUMA hardware. As Linux gets used on more NUMA hardware platforms, there are bound to be additional areas exposed which will benefit from additional NUMA optimizations.

Some areas that are actively being worked on or considered for future work are:

- I/O busses in `sysfs` topology
- MPIO
- scheduler enhancements
- interrupt routing and balancing
- kernel data structure placement
- memory binding
- page migration
- timers

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM, NUMA-Q, Power4, and xSeries are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Intel, i386, Itanium and Xeon are trademarks of Intel Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

References

[1] M. Gorman: "Understanding the Linux Virtual Memory Manager," April 2003, `http://www.csn.ul.ie/~mel/projects/vm/guide/html/understand/`

[2] W. Irwin, March 2003 `http://marc.theaimsgroup.com/?l=linux-kernel&m=104660383911943&w=2`

[3] LWN.net, "Speeding up reverse mapping" `http://lwn.net/Articles/9555/`

[4] Ingo Molnár, `http://people.redhat.com/mingo/O(1)-scheduler/`

[5] M. Kravetz, H. Franke: "Implementation of a Multi-Queue Scheduler for Linux," April 2001, `http://lse.sourceforge.net/scheduling/mq1.html`

[6] Davide Libenzi, October 2001, `http://www.xmailserver.org/linux-patches/lnxsched.html`

[7] Andrea Arcangeli, "NUMA," presentation at the UKUUG Manchester, June 2001, `http://www.ukuug.org/events/linux2001/papers/html/AArcangeli-numa.html`

[8] H. Franke et al, "PMQS: Scalable Linux Scheduling for High-End Servers," `http://lse.sourceforge.net/scheduling/als2001/pmqs.ps`

[9] Erich Focht: "Node-Affine NUMA Scheduler," Feb. 2002, `http://home.arcor.de/efocht/sched`

[10] John Stultz: "Nodeless MCS Lock," `http://www-124.ibm.com/linux/patches/?patch_id=218`

[11] "NUMA AWARE LOCKS," `http://lse.sourceforge.net/numa/locking`

[12] Patrick Mansfield: "SCSI Mid-Level Multi-path/port storage," `http://www-124.ibm.com/storageio/multipath/scsi-multipath/index.php`.

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

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