Measuring Resource Demand on Linux

Resource allocation, Goldilocks style

Rik van Riel
Red Hat, Inc
riel@redhat.com

Abstract

Linux, and other Unix systems, have long had pretty good measurement systems for resource use. This resource use data, together with tools like top and vmstat, has allowed system administrators to effectively gauge system performance and determine bottlenecks. However, this needs to be done manually and is more art than science and nobody knows exactly how much resources a particular workload needs.

The result? Most machines have way more resources than needed for the workload they run. This is not a problem with dedicated computers, but once virtualization is introduced people will ask the question "how many virtual machines can I run per physical system?"

From this question alone it is obvious that resource demand is not the same as resource use, and resource demand should probably be measured separately by the operating system. In this paper I will introduce ways to measure resource demand for CPU, memory and other common resources, examine why resource demand is often different from resource, and explain how system administrators can benefit from having resource measurements.

1 New problems

Virtualization is the buzzword of the day, but besides its promises of reduced hardware cost, reduced power use and world peace, it has the potential to introduce almost as many problems as it solves.

The most obvious one is that when two servers get consolidated onto one, you now have the operating systems from both servers to manage, as well as the host operating system. Now the sysadmin has to take care of three OSes instead of two. Luckily system management is a fairly well understood problem and this problem can be reduced or solved with automated management tools and the use of stateless Linux.

Consolidation of multiple workloads on one system brings another problem to the foreground. In order to increase the utilization of systems and reduce the number of active physical computers as much as possible, the number of virtual machines per physical computer needs to be maximized, which in turn means that each virtual machine only receives the minimum amount of resources it needs.

Unfortunately current Unix and Linux systems are only geared towards measuring current resource use. However, if a virtual machine has 512MB of memory allocated to it, there is no way to tell whether it is enough for the workload it is running, or too much, or just right. Without a way to know how much resources each virtual machine really needs, dynamically reassigning resources from one virtual machine to another cannot be done reliably, and each virtual machine will need to get excess resources allocated to it, just in case. In short, resource demand is not the same as resource use, and operating systems will need to measure both.

This paper will cover resource demand measurement of the following resources:

- CPU
- Memory
- Network I/O
- Disk I/O

2 CPU

A CPU shortage is easily detected from inside a virtual machine: the processes in the system do not get enough CPU time, the runqueue starts growing longer, idle time is low and users start complaining that the applications are reacting really slowly. However, on a virtualized multicore or multi-processor system, this could have multiple causes, each of which needs a different solution.

The most obvious cause would be that the CPUs in the system are simply too slow for this workload. The only fix in this case would be a hardware upgrade.

A second cause could be that while the system has enough CPU power, the virtual machine cannot use enough because it does not run on enough CPUs simultaneously. For example, the physical computer has 8 CPU cores, but the virtual machine only has 2 virtual CPUs. In this case the performance problem can be fixed by adding more virtual CPUs to the virtual machine, allowing it to use more CPU cores simultaneously.

A third situation is that the system has enough CPU power, the virtual machine has the potential to use all the CPUs, but it does not get scheduled in often enough because other virtual machines are using the CPU time. The easy fix in this scenario is migrating virtual machines to other physical computers.

Because each scenario needs a different solution, and the first solution is a lot more expensive than the other two, it is important that the operating system and the hypervisor keep statistics that allow the system administrator to distinguish the different cases from each other.

2.1 CPU steal time

CPU steal time is a concept from IBM's S390 mainframes, and has been present in S390 Linux for a while. CPU steal time denotes the time that:

- A virtual CPU had runnable tasks, but
- the virtual CPU itself was not running.

This occurs whenever the hypervisor schedules another virtual CPU, usually from another virtual machine, on the physical CPU. In short, it measures the contention on the CPU between multiple virtual machines. Linux running on Xen also shows the CPU steal time, which is the very last number in the `cpuN` lines in */proc/stat* (see figure 1).

These columns represent user time, nice time, system time, idle time, iowait time, hardirq time, softirq time and steal time respectively.

```
$ cat /proc/stat
cpu  82295 80106 166899 154966547 128436 7924 2729 17698
cpu0 82295 80106 166899 154966547 128436 7924 2729 17698
...
```

Figure 1: CPU statistics from /proc/stat on the 2.6 kernel

Astute readers will have noticed that the number of columns in the CPU statistics in */proc/stat* have doubled since the 2.4 kernel. It will be interesting to see how tools like top, vmstat and sar will cope with the new statistics, considering both the needs of system administrators and the limitations of terminal screen space.

2.2 Diagnosing the situation

When idle, iowait and steal time are all low, the applications are getting most of the physical CPU time. If the number of running threads is the same as the number of CPUs, the only thing that will improve performance is having faster CPUs.

If the number of running threads or processes is larger than the number of CPUs, allowing the virtual machine to run on more physical CPUs simultaneously, by adding virtual CPUs, may be able to fix the performance problem.

If idle and iowait time are low, but cpu steal time is high, that means your physical CPUs are suffering from contention between multiple virtual machines. Performance can be increased by migrating some of the virtual machines to other physical systems.

Of course, it is possible that every physical server is loaded with one low priority virtual machine to run calculations in the background, for example scientific calculations or financial risk analysis. Since these applications are supposed to eat up all the CPU time that is available, migrating them around will make little sense and CPU steal time on these low priority background virtual machines will simply be a fact of life and not something to worry about.

3 Memory

Memory is a lot harder to reallocate from one virtual machine to another. This is because memory is a non-renewable resource. Every second there is a new second of CPU time to divide between virtual machines, but the amount of memory in a system tends to stay constant.

This means that in order to give memory to one virtual machine, it will have to be taken away from another virtual machine. That in turn involves the balloon driver and the pageout code in the “donor” virtual machine, which can incur a significant latency. Hence, memory allocation between virtual machines focuses around these areas:

- Identify which virtual machines need more memory, and how much.
- Identify which virtual machines have too much memory, and how much.

3.1 Refaults

A virtual (or physical) machine can benefit from more memory when it spends a significant amount of time waiting for memory to be paged in, when that memory was recently evicted. In order to estimate this, two factors need to be considered.

The first is iowait time, or the time the CPUs in the system have tasks that would be runnable if it weren't for the fact that they are waiting on IO to complete.

The second factor is the number of recently evicted pages that got faulted back in, and how many pages got evicted after the page in question got evicted. The second estimate is important because it shows exactly how much more memory the virtual machine would have needed to avoid this page fault. A histogram with this statistic is shown in figure 2.

```
$ cat /proc/refaults
    Refault distance                 Hits
        0 -     32768                 192
    32768 -     65536                 269
    65536 -     98304                 447
    98304 -    131072                 603
   131072 -    163840                1087
   163840 -    196608                 909
   196608 -    229376                 558
   229376 -    262144                 404
   262144 -    294912                 287
   294912 -    327680                 191
   327680 -    360448                  79
   360448 -    393216                  68
   393216 -    425984                  41
   425984 -    458752                  45
   458752 -    491520                  31
New/Beyond     491520                2443
```

Figure 2: Refault statistics from /proc/refault

As an example, consider a page that gets faulted in and was evicted fairly recently, with only 20,000 other pages having been evicted since this page got evicted. In this case, if the virtual machine had 20,000 more pages, all these 20,000 pages would still have been resident in memory and this page fault would not have happened.

Armed with this knowledge and a histogram of refault distance versus the number of faults at that distance, we can calculate roughly how much IO the system would have avoided, if it had certain amounts of memory more than it has currently.

Consider a system that has 80% iowait time, meaning it spends 80% of its time waiting for IO to complete. If half of the IO being done is on pages that were evicted "less than 200MB ago," increasing the amount of memory of that virtual machine by 200MB will reduce the amount of IO necessary by 50%, which could significantly increase the performance of the workload on the system. Figure 3 shows an example of how memory resizing avoids page faults.

If the system has a batch type workload, this could represent a 50% speedup in performance. Because the VM uses a better replacement algorithm than pure LRU, the results could be better than the predicted 50% performance increase.

Conversely, imagine another virtual machine on the same system, running a totally different workload. This workload mostly streams over large quantities of data and rarely touches the same page twice. Because of this, most of its page faults will happen on pages that were never seen before, or on pages that were evicted very long ago. Giving this virtual machine 200MB extra memory is not going to help at all, because it is not accessing a lot of recently evicted data.

Without taking refault distance into account, it would not have been possible to easily distinguish between the first virtual machine, which gets a large performance boost from 200MB extra memory, and the second virtual machine, which would not have gotten any noticable boost from being allocated extra memory.

MEMORY EXPANSION & EVICTED PAGES

REFAULT DISTANCE:
How far from resident memory an evicted page is.

HITS:
How much a range of pages is in demand on the system.
In other words, how many faults have occurred when a page
that has been evicted is requested from resident memory.

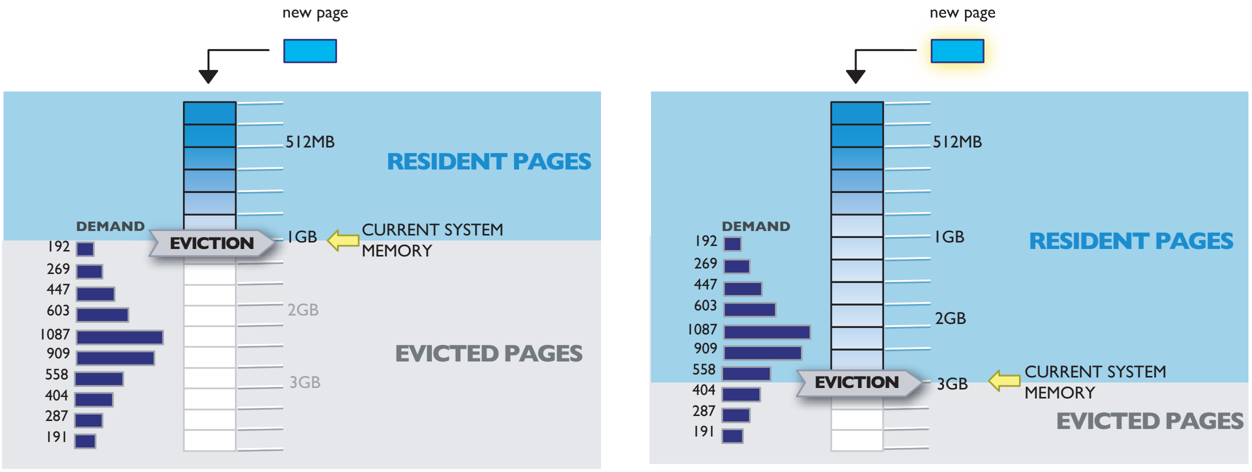

Figure 3: Increasing memory size avoids I/O on pages that would have otherwise been evicted and refaulted.

3.2 Non-resident pages

Keeping track of recently evicted pages and refaults does not require utmost precision, which leaves space for optimizations. The naive implementation of non-resident page tracking would keep the same metadata for a non-resident page as for a page that is resident in memory, e.g. a full `struct page`.

However, all we need when faulting a page back in from swap or the filesystem is to:

- identify the page, with a high degree of certainty,
- estimate roughly how many pages got evicted from memory after the page in question got evicted,
- using a data structure that is small, and
- allows for efficient and SMP scalable lookup.

When evicting a page and when faulting it in later, the kernel knows a number of details about the page, such as the file (address_struct) the page belongs to `(page->mapping)`, the offset of the page into that file `(page->index)` and the inode number of the file. Not only do these details uniquely identify the page with pretty high certainty, they can also be easily hashed into a single value, meaning the information needed to identify a non-resident page only takes up 32 bits.

However, if we were to use traditional lookup methods like a tree or linked list, the space taken up by the lookup pointers alone would triple or quadruple the space taken up by the page identifier alone, and we haven't even stored information about when the page got evicted from memory yet.

Another possibility would be a huge array with a clock hand. Every time a page is evicted from memory, record the hash value identifying the page in the element pointed to by the clock hand, and move the clock hand one position forward. On pagein, scan the array until the hash value identifying the page is encountered. The distance the clock hand has advanced since the page got evicted corresponds to the number of pages that got evicted after this page. Space efficient, but prohibitively expensive time wise if the array contains hundreds of thousands of elements, say one for each page in the system.

If a page is not found in the set of recently evicted pages, we will categorize this fault as being either a page we have never seen before, or a page that was evicted so long ago we no longer track it. This is represented in the last line of */proc/refaults* in figure 2.

A compromise is to use many small arrays (a non-resident bucket, or `struct nr_bucket`), each the size of one or two CPU cache lines and with a clock hand. On pagein we hash `page->mapping` and `page->index` to determine which array to check. The nr_bucket has only up to a few dozen entries, which can be compared with the calculated hash value very quickly since they all sit in the same CPU cache line(s).

```
struct nr_bucket
{
        atomic_t hand;
        u32 page[NUM_NR];
} ____cacheline_aligned;

/* The non-resident page hash table. */
static struct nr_bucket * nonres_table;
static unsigned int nonres_shift;
static unsigned int nonres_mask;
```

Figure 4: An efficient data structure for tracking non-resident pages

The total refault distance, meaning the number of pages that got evicted since this page got evicted, can be estimated by multiplying the distance the clock hand has advanced since the page got evicted (the clock hand local to the nr_bucket) with the number nr_buckets. This works if the hash value is good enough to distribute the evicted pages evenly between the nr_buckets, which appears to be the case in practice.

This method is space efficient, using one u32 per non-resident page and one clock hand per small array of non-resident pages. If we use a different hash of `page->mapping` and `page->index` for selecting the nr_bucket than the one used for identifying the page, we effectively increase the hash size without needing more storage.

Additionally, the information on whether a page that is faulted in was recently evicted is needed for advanced page replacement algorithms, like 2Q, CAR/CART or CLOCK-Pro. Some of these algorithms need a flag in addition to the page identifier; this flag should fit in one or two bits of the u32, reducing the page identifier to 31 or 30 bits.

3.3 Page references

Being able to identify which virtual machines can and can not benefit from being allocated extra memory allows the system to allocate memory to the right virtual machines. What remains unanswered is the question which virtual machines will not suffer a performance decrease when their amount of virtual memory is reduced. After all, if we want to give extra memory to one virtual machine, that memory will have to be taken away from another virtual machine.

The answer lies in page references. The pageout code inside each virtual machine scans over its memory and evicts the pages that have not been accessed recently and/or frequently.

If a large fraction of the pages being scanned by the memory management pageout code were recently referenced, the virtual machine is using most of its memory and we should not take away memory from this system.

On the other hand, if a virtual machine only accessed a small fraction of its pages, it is not using most of its memory. If this virtual machine is spending a lot of time waiting for recently evicted memory to be paged back in, it could benefit from getting extra memory. However, if the time spent waiting for recently evicted memory is negligable, and it is not using most of the memory it has, then this virtual machine is a good candidate to take memory away from. After all, it does not really need it...

Even an IO bound workload, e.g. a data mining job that rarely accesses the same page twice, can still fulfill these criteria and have memory taken away from it. This is fine, because this workload does not benefit from the memory, and the resulting reduction in IO done by the other virtual machine means more disk bandwidth is left over for this workload.

Virtualization is often used because of the performance isolation qualities it provides, so systems should not reduce the amount of memory allocated to a virtual machine by too much. Quality of service benefits from having decent minimum and maximum memory allocations for each virtual machine, and varying the current amount within that range as needed by the workload.

4 Disk and Network I/O

Network bandwidth allocation can be done in a very similar way to how CPU is allocated, with the difference that the hypervisor has no easy way to control incoming network traffic. Some tricks can be played with TCP, but not all traffic can be controlled. This means that fair sharing of network bandwidth can not be fully implemented by the virtualization software, and more attention will have to be paid to making sure that the workloads on the system do not suffer from network contention, upgrading the network bandwidth before it becomes a bottleneck.

Disk I/O is a little different from CPU an memory, because multiple I/O requests can be outstanding simultaneously. With network or other cluster accessible storage, it is even possible for the storage subsystem to be busy serving requests initiated by other systems. This makes I/O bottleneck monitoring at the virtual machine level or even at the physical server level hard or incomplete, and monitoring should probably be done on the storage subsystem itself.

5 Conclusions

Consolidation is one of the big drivers of virtualization. In order to maximize cost saving, users will want to consolidate their workloads on as few physical systems as needed for their workloads. With live migration, users may even be able to power off server capacity that is not currently loaded.

However, in order to maximize consolidation of multiple workloads, it is necessary to measure not just the amount of resources used by each virtual machine, but also to estimate the amount of resources that each virtual machine really needs.

Changing system structure means that system administrators with a good gut feeling on how to tune physical servers may find that their instincts do not always work on virtual machines. Furthermore, automated system administration tools have no instincts, so direct measurement of resource demand will be a necessity.

Scheduling renewable resources like CPU time, network bandwidth or disk I/O requests is mostly straightforward. On the other hand, reassigning non-renewable resources like disk space or memory takes considerably more effort. This may justify fancy algorithms to allocate the right amount of memory to each virtual machine, and limit the times memory has to be reassigned from one virtual machine to another.

6 References

Song Jiang, Feng Chen, and Xiaodong Zhang *CLOCK-Pro: an effective improvement of the CLOCK replacement* Proceedings of 2005 USENIX Annual Technical Conference (USENIX'05), Anaheim, CA, April 10-15, 2005.

Sorav Bansal and Dharmenda S. Modha *CAR: Clock with Adaptive Replacement* in Proceedings of the USENIX Conference on File and Storage Technologies (FAST), pages 187–200, March 2004.

Johnson, T., Shasha, D.: *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm*, Proceedings of the 20th IEEE VLDB Conf., Santiago, Chile, 1994, pp. 439 - 450

Linux advanced page replacement development page: `http://linux-mm.org/AdvancedPageReplacement`

Proceedings of the
Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin