Shoot first and stop the OS noise

Dealing with microsecond latency requirements

Christopher Lameter
Linux Foundation
cl@linux-foundation.org

Abstract

Latency requirements for Linux software can be extreme. One example is the financial industry: Whoever can create an order first in response to a change in market conditions has an advantage. In the high performance computing area a huge number of calculations must occur consistently with low latencies on large number of processors in order to make it possible for rendezvous to occur with sufficient frequency. Games are another case where low latency is important. In games it is a matter of survival. Whoever shoots first will win.

An operating system causes some interference with user space processing through scheduling, interrupts, timers, and other events. The application code sees execution being delayed for no discernible reason and a variance in execution time due to cache pollution by the operating system. Low latency applications are impacted in a significant way by OS noise.

We will investigate issues in software and hardware for low latency applications and show how the OS noise has been increasing in recent kernel versions.

1 Introduction

Operating system noise is something of a mystery to most user space programmers. The expectation by those writing the application is that the operating system is simply letting the application run. Users see the main function of the Operating System to provide resources for the program to run effectively. In the case of OS noise the Operating System itself becomes a problem because the OS is interfering with the application by making uses of the processor for maintenance tasks or operating system threads that also have to run on the processor. Running OS code may impact the application which will not perform as expected. The execution times of critical code segments in the application may vary a lot without any discernible reason. One hears complaints from users that the OS should just get out of the way. The problem becomes more severe as the number processors increases and as interconnects become faster.[1] The timing requirements for critical sections move from milliseconds to microseconds. Modern processors can perform a significant amount of work in a microsecond provided that there are no latencies associated with the data needed for processing. Therefore, processor caches have a significant effect on the latencies of critical code segments. The OS code causes disturbances in the processor caches that has an effect long after the application continues execution.

The problem was first noticed first in High Performance Computing.[2] HPC applications typically go through a highly parallel calculation phase and then a rendezvous phase in which the results of the calculations are exchanged. It was noted that the calculation phase was rarely performing within the boundaries expected. The problem became worse as the number of processor increased. The investigation found that the rendezvous phase will be delayed if any one of the processes is held up due to OS interference (like for example a timer interrupt). The more processors exist the more likely the chance that OS interference will cause a delay. This is especially severe on Linux due to the staggering of the timer interrupts over all processors in the system. As a result the timer interrupts will not run all at the same time (which would potentially overload the interconnect between the processors). The more processors a system has the shorter the period that no timer interrupt is running on any processor and the more likely that the

[1] Infiniband hardware can f.e. perform transfers between machine in 1-3 microseconds!

[2] See especially Petrini, 2003

rendezvous phases are delayed due to a single processor being hit with OS interference. This can lead to severe performance regressions so that some vendors have started to modify the scheduler to have special synchronized periods dedicated to OS processing in order to be able to execute concurrently on all processors without OS interference during other times.[3]

In the financial industry we see a arms race to lower latencies.[4]

Latencies for the exchange of financial data and for trading used to be measured in milliseconds but that has now come to focus on microseconds. Whoever can react in the fastest way to changing market conditions may take advantage of a favorable trade opportunity. The latency requirements in the financial sector focus more on networking and on the need of fast processing of huge and complex sets of data. The classic decision support system (DSS) paradigm is taken to extremes there. Stopping the noise results in concrete market advantages.

A similar move is seen in the gaming industry. There also a growing focus on smaller and smaller intervals for critical processing develops. If interactive computer games are played over the Internet then the focus is mostly on millisecond latencies since the WAN links do not allow smaller latencies. This limits the richness of interactivity of computer games. Recently there has been an increased move towards putting interactive games on LANs where players are in local proximity (LAN parties). Gaming software can exchange large sets of information in sub millisecond time frames in such configurations. There it is likely that we will also facing issues with microsecond latency demands in the future and therefore OS noise is also becoming factor. OS noise there can determine whoever will be able to shoot first. One side effect of latency in shooter games is that the bullet of the slower machine seems to hit the target (since the slower machine was not able to acquire the updated position of the enemy) but the enemy takes no damage since the person has already moved on in the game servers reckoning and the game server determined that the shot missed the target. The enemy is hit, it dies a horrible death on the screen of the shooter and then suddenly continues running down the corridor.[5]

[3] See Beckman 2008, 5. Tsafir 2008, section 4. Petrini 2003, 10

[4] F.e. 29West–a major player for middleware in the financial area–recently announced a vision for zero latency.

[5] See http://en.wikipedia.org/wiki/Multiplayer

2 What is Operating System Noise

What exactly is Operating System noise? The common definition in use is any disturbance caused by the OS making use of a processor. I would like to extend that definition to cover everything not under the control of the application that has a negative impact on performance and latencies observed by code running in user mode. This goes beyond the strict notion of OS noise and more towards a notion of general noise (maybe better called *system noise*) that impact on an applications performance. Noise is not only the result of interruption of code execution (be it the periodic timer interrupt, device interrupts, software interrupts, faults and so on) but also memory subsystem disturbances due to the Operating Systems putting pressure on the cache subsystems of the processor which causes application cache line refetches.[6]

The use of on chip resources of a CPU by an OS or another application are important. Resources commonly available are the processor caches, the TLB entries, page tables and various register copies. Contention on all these levels can reduce the performance of the application. If the OS scans through a large list of objects in a regular way[7] then a large number of TLB entries may be evicted that have to be re fetched from memory later. If memory becomes scarce and the OS evicts pages from memory then the eviction may have a significant latency effect since the evicted pages will have to be re-read from secondary storage (such as a hard disk) when it is needed again.

Processors are also not isolated from each others. The notion of "CPU" that the Linux OS has is basically a hardware managed execution context. These can share caches with other "CPUs" on various levels. If cache sharing occurs between multiple of the CPUs then a process on another processor can cause cache lines of the application to be evicted, can use TLB lines and other processor resources that cause latencies for the application. The most significant effects occurs if multiple execution context share all resources of the processor like for example in hyper-threading. Operating system schedulers (like the Linux scheduler) currently only have simple handling of these dependencies and rely

[6] According to Beckman cache line eviction is the major effect increasing the latency of critical sections. But that may depend on the type of load running.

[7] Like for example done by the SLAB allocator in Linux

mainly on heuristics. This leads to a situation in which increasing load customarily leads to a general slowdown of all processes running on the machine once all cpus are actively processing data.

3 Latency Overview

In order to talk in a meaningful way about latencies it is important to know what these time frames represent in reality. One thing that is often forgotten is that telecommunication or general signal latencies are limited by the speed of light (300.000km/sec). The relativistic limits become significant when signals have to run over long distances. Whenever signals must travel across a WAN link latencies in the range of milliseconds become unavoidable. Signals travel over fiber optic or copper links at the speed of around 200.000km/sec. Since the earth has a circumference of 40.000km: A signal that is supposed to reach any point on the earth (the earth is round so we can reach any point within 20.000km) must account for a minimal latency of 1/10th of a second.

Here is a list of latencies and how they apply to networking and OS events. Each latency includes the notion of a distance that a signal can have traveled in that latency period:

3.1 1 second

- Time needed for light to reach the moon.

3.2 100 milliseconds

- A signal can reach all of the earths surface.
- Minimum human reaction speed.
- Timer interrupt interval for Linux systems configured with 100 HZ.
- Half of the TCP retry interval and SYNACK interval
- Typical Internet latency for high speed consumer grade links

3.3 10 milliseconds

- 2000km distance. Reach surrounding metropolitan areas.
- Timer interrupt interval for systems configured with 1000 HZ.
- Major fault (page needs to be read in from disk).
- Rescheduling to a different process on the same CPU.

3.4 1 millisecond

- 200km distance. Reaches systems in your city.
- Sound travels 34 centimeters. A signal from a speaker reaches your ear.
- Average seek time of a hard disk.
- Camera shutter speed.

3.5 100 microseconds

- 20km. Signal confined to local LAN or building.
- Maximum tolerable interrupt hold off.
- Best Ethernet ping pong times on 1G between neighboring systems.

3.6 10 microseconds

- 2km. Signal confined to local LAN.
- Minor page fault (Copy on write after fork).
- Duration of time interrupt.
- Duration of typical hardware interrupt.
- Typical IRQ hold off period if kernel disables interrupts.
- Duration of a system call.
- Context switch.
- Relativistic time distortion in GPS systems that needs to be compensated for.

3.7 1 microsecond

- 200m. Local LAN.
- Resolution of *gettimeofday()* system call.
- Duration of a vsyscall
- PTE miss and reloading of TLB
- Start of hardware interrupt processing

3.8 100 nanoseconds

- 20m. Within your room.
- Cache miss. Time needed to fetch data from memory.
- TLB miss.

Signalling latencies are currently a major restriction for building large supercomputers. The latency of memory subsystems can only be reduced if the subsystems are packed in a dense way. If the memory is over 20m away from the processor (even less in reality) then the time it takes the signal to travel across the wire will take up a major portion of the latency. It really does not matter how fast the memory is if its physically too far away.

Similarly one has to be careful of offers of "faster" DSL lines or network connections. "Faster" not mean that the speed of the data going across the wire is increased. It means that the number of bytes that can be transmitted at one time is increased. "Faster" DSL means a higher capacity link not that there will be any real increase in transmission speed. For gamers the distance to the game server is really important. If you live far away from the population centers then one is usually at a severe disadvantage due to signal run latencies. Others will shoot first and you will shoot and hit them but they wont get killed. The only solution for gamers to be able to shoot first is to figure out where the game serves are located and move nearer to them.

I keep getting questions about how to make things faster in terms of latency but the laws of physics are squarely in the way here. There is no solution in sight. Maybe someone can rework physics to show us how its done? Then maybe we can quantum tunnel signals, use warp drive bend time and space to fix this issue. If we can do that then we can likely also do all the other nice stuff that shows up in space fiction movies.

4 Characteristics of Noise

There are a couple of way to characterize noise. Noise can be seen as an interruption of the execution of the application. Noise in this form is a simple DoC (Denial of CPU) by the OS and can be measured by repeatedly taking time stamps. If the difference is higher than usual then some outside influence interrupted the process and stole processing time. It is typical to set some limitation of a boundary over which a delay is long enough to be considered a notable noise event. The important characteristics of noise that emerge from this method are the noise duration and their frequency.

Noise also has an influence on the execution speed of code through additional cache misses, TLB misses and internal processing within the processor. It is far more difficult to measure these effects. If the execution speed of a given segment of code is known then the deviation can be determined by also measuring the execution speed of a segment. However, that is only possible for code segments that can be executed repeatedly with the same data. The noise can then be quantified in the percentage of slowdown in the code segment due to noise interference.[8]

Noise interacts with the application in various ways. There are applications that are sensitive to certain types or noise and can tolerate others. Typically one assumes a linear correlation of the noise due to application performance. However the noise can resonate with processing intervals of the application which can then lead to a butterfly effect that amplifies the delays in the applications. The intervals of communication of the application are of importance. If the application does not frequently exchange information with other processes then the impact of fine grained noise (as usually presented by an OS) is minimal. However, if information exchanges occur frequently then the final grained noise can affect the critical communication paths and significant effects can develop.[9]

Some researchers have found that noise below the 1 microsecond boundary usually does not cause significant harm.[10] In the following surveys we will adopt 1 microsecond as a boundary for an OS event that is considered significant for our investigations.

[8] The measuring points will add additional latencies and cause more disturbance of processor resources in addition to the OS noise

[9] Petrini, 2003

[10] Tsafir 2008, under section 3.2 *Granularity*

4.1 Sources of noise

- The Linux scheduler is a prime cause for OS noise. Even if a process runs on an otherwise idle system: The scheduler will reschedule another process on any processor at least once a second (involuntary context switches). These context switches can be avoided by setting a real time priority (SCHED_RR or SCHED_FIFO). But real time priorities still do not stop the OS from processing maintenance code on the same processor. Especially the scheduler softirq will still be executed from the timer interrupt to keep statistics and check if other processes should be run on the processors. The scheduler is currently not designed to leave a running process alone.

- The Linux timer interrupt occurs with HZ frequency every second. Typically kernels are run with 1000 HZ meaning that a noise event occurs every millisecond. The timer interrupt in turn may run various regular maintenance tasks that increase the length of the events impacting the application.

  The kernel has an option to enable a tickless system (CONFIG_NO_HZ) but the tick is only switched off if a processor is idle. A busy processor will invariably get hit by the timer tick. The description of CONFIG_NO_HZ as enabling a "tickless system" is a bit misleading.

  There are other options for how to schedule OS maintenance events. Solaris only has a timer interrupt on processor 0. All other processors are left alone. The scheduler executing on processor 0 schedules the processes running on the other processors. On Solaris it is important to run processes on other processors in order to reduce OS noise.[11]

- Cache disturbances

  If multiple cpus (hardware execution context) share the same caches then another executing process on other cpus has access to processor resources necessary for execution. This is particularly significant if the processor supports hyper threading. All caches are shared then. L2 and L3 caches are also frequently shared between multiple processors.

- TLB miss

  TLB misses occur when the cache of virtual to physical mappings of the processors get exhausted. This is common if a threads memory accesses are sparse or are randomly covering large memory areas. Pointer chasing is a typical application that creates TLB misses. If the working set of a process becomes larger than the TLB coverage then it is possible that every memory access requires a new TLB fetch. If the TLB use of an application is high then OS processing may cause key TLB entries to be evicted.

  TLB resources are typically shared between multiple cpus meaning that the full TLB coverage is not available for single processor.

- Major page fault

  Major page faults involve bringing in a page from secondary storage (usually a hard disk). These are also a major causes of latency. Major faults are avoided by read ahead functionality of the file system. If the system detects linear reads from a disk device then multiple of pages are read in anticipating future faults. If read ahead has been performed for a page then only a minor fault will be generated.

  Major faults can accumulate if the OS starts to evict pages from memory that have been rarely used. If the pages that are missing from the process are relatively sparsely spread over large areas of secondary storage then the read ahead logic will be ineffective and each page fault may cause long latencies. From the application perspective these are not discernible from OS noise. The application accesses a memory location which results in an unexpected major delay.

- Minor page fault

  A minor fault is making a page visible to a process that has never accessed it before. If another process or read ahead has already brought a page into memory then a minor fault involves settings up the page tables so that the page becomes visible in the address space of a process.

  A minor fault can also occur when writing to the memory of a page. In that case we may have to copy the page (Copy-On-Write = COW) or update page dirty statistics.

- System threads

  The Linux kernel itself creates threads that are used for scheduling background write out, event handling and so on. File systems and other kernel

[11] Radojkovic 2007, 5

subsystems create their own background processes. These are usually fixed to a specific processor. The way to keep these quiet is not do perform actions that require background activities on a processor. The activities of these threads will shut down after some idle time of the subsystems.

It is fairly typical for these threads to only cause minor delays. However, the scheduler has to perform a context switch to the threads and back. The overhead increases significantly if large lists have to be processed (LRU expiration of inodes, slab object expiration).

- User space background daemons

  The user space background daemons are mostly created during boot up and have various administrative functions. It is possible to bind these processes to specific processors through the *taskset* tool. These background daemons can cause particularly long hold offs. Notorious examples are logging daemons that can issue fsync() system calls to force the log messages onto disk which may cause long delays due to synchronous write outs to disk.

5 Utilities to measure noise

I found no tools to measure noise under Linux so I created a series of test programs available at http://gentwo.org/ll. A small introduction to some of their features.

5.1 Low latency library

The low latency library (*ll*) contains basic function to obtain time stamps in a variety of magnitudes in an inexpensive way via the time stamp counts. Logic is included to determine the processors characteristics and the cache layouts from user space. These are basic necessities for measuring intervals in an accurate way with the least impact on a user space program and for tuning a user space application to the cache size or number of cores available.

5.2 latencytest

latencytest is a tool that continually retrieves the value of the time stamp counter and compares with the last time stamp obtained before. If a certain threshold (default 1 microsecond) is reached then the event is registered as a noise event. Latencytest produces a histogram and prints out statistics regarding the intervals observed. Latencytest monitors various scheduler statistics about itself and will note if the scheduler moves the process to another processor or performs a context switch away from the process.

Latencytest is a test load that can be run while other system activity is going on or with special scheduler parameters to see how the scheduler would change the treatment of a process.

The code used to determine how scheduling can affect the test load can be used as sample code to instrument a user space program. Typically these would be used to determine characteristics of key critical code segments.

5.3 latencystat

latencystat is able to display latency statistics of any process in the system via the */proc/<pid>/schedstat* information. It is comparable to *vmstat* and displays continually how long a process ran without another process having been scheduled and determines the average wait time from the point that a process became runnable until the scheduler gave the processor to the process.

5.4 trashcache

trashcache is a program that runs forever and does random memory accesses in order to trash the processor caches. Running *trashcache* on a sibling CPU can be used to gauge the impact of CPU cache thrashing on a user space process.

5.5 OS diagnostics

The operating system itself has counters for scheduler events that one can query f.e. via the *getrusage()* system call. Example code can be found in the source code for the *latencytest* tool.

5.6 udpping

udpping is another tool to measure OS noise by sending network packets back and forth between two systems. The UDP ping pong is the fastest time to communicate between two hosts using the IP stack. Noise shows up as variances of transfer times between both system.

6 Some OS noise measurements under Linux

The tools above can be used to measure the OS noise characteristics under Linux. Here we measured a completely idle system and see what the effect of the OS has on a simple test load (latencytest tool). It is important to note that this is the best scenario that can ever happen for the given kernel version. There will be numerous additional disturbances through cross-cache effects, system and user space daemons and so on if the system would be running a realistic load. What we measure here is a best case scenario. Everything else is guaranteed to be worse than what we measure.[12]

First a test running *latencytest* for various kernel versions. The tests are run for 10 seconds each and we record events longer than 1 microsecond. Most of the events recorded are timer interrupts of a duration longer than 1 microsecond. Timer interrupts may occur that are less of one microsecond in duration but these low latencies are only reached during favorable conditions when not much work is to be done from the timer interrupt and if the queue of functions to call is small. The test load does not have a large cache footprint (fits nicely into the L1 cache) meaning that most of the cache lines used for the timer interrupt will remain in memory. The processor in use here is a Penryn, dual quad core (Xeon X5460) at 3.16Ghz.

Version	Test 1	Test 2	Test 3	Sum
2.6.22	383	540	667	1590
2.6.23	2738	2019	2303	7060
2.6.24	2503	573	583	3659
2.6.25	302	359	241	902
2.6.26	2503	2501	2503	7507
2.6.27	2502	2503	2478	7483
2.6.28	2502	2504	2502	7508
2.6.29	2502	2490	2503	7495
2.6.30	2504	2503	2502	7509

Table 1: Latency events >1 microseconds for a number of kernel versions

The number of noise events was initially quite low. With 2.6.23 (which introduced a new scheduler) we see a significantly higher number of noise events. Things improved with 2.6.24. In 2.6.25 we had a significant reduction of the OS noise to the smallest value seen. However, that was lost in 2.6.26.

[12] For a worse case run a latencytest during a kernel compile

The tests were run on a "tickless" system (CONFIG_NO_HZ is set) because the description for a "tickless" system given was that the timer interrupt only occurs as necessary. However, as seen here: The timer interrupt seems to occur regularly.

Each test run 10 seconds and in those 10 seconds 2500 time 1 HZ intervals occur since the kernel was configured with 250 HZ. Therefore what we see here are must be timer interrupts causing noise. The timer interrupt in 2.6.22 and 2.6.25 ran less than 1 microseconds otherwise the latencytest tool would have registered them.

The next test shows the average length of the noise events registered.

Version	Test 1	Test 2	Test 3	Average
2.6.22	2.55	2.61	1.92	2.36
2.6.23	1.33	1.38	1.34	1.35
2.6.24	1.97	1.86	1.87	1.90
2.6.25	2.09	2.29	2.09	2.16
2.6.26	1.49	1.22	1.22	1.31
2.6.27	1.67	1.28	1.18	1.38
2.6.28	1.27	1.21	1.14	1.21
2.6.29	1.44	1.33	1.54	1.44
2.6.30	2.06	1.49	1.24	1.60

Table 2: Duration of Latency events >1 microseconds for a number of kernel versions

The tests shows that the time spend in the timer interrupts gradually increases. Interestingly 2.6.22 and 2.6.25 have much longer noise durations. The long durations may be a consequence of the OS batching multiple events in fewer timer events. The remaining timer events have less processing to do and therefore their processing time may under the 1 microsecond boundary. A significant portion of timer events in 2.6.22 and 2.6.25 took less than 1 microsecond.

We see the effect of kernel bloat in 2.6.28, 2.6.29 and 2.6.30. The average time spend in the timer interrupt gradually increases. This causes more and more regressions for latency sensitive applications. The question is what is worse: Batching events to have fewer noise above 1 microsecond of longer duration or having more events with a smaller duration.

The above results suggests a simple way to reduce the frequency of OS noise in the Linux kernel: Reduce the frequency of the timer interrupts. In the following measurements we let *latencytest* run for 60 seconds

and measure the number of noise events: Once with SCHED_OTHER which allows the scheduler to schedule other processes on the processor (although there is nothing else running on the system). And the second time with a real time priority SCHED_FIFO which does not allow the scheduler to take away the processor and give it to another process (but the kernel can still execute any of its threads and thereby create OS noise).

HZ	Events	Duration	CSw	RT events	RT dur.
18	1088	2.76	76	893	2.41
60	6042	2.62	95	6013	2.43
100	6012	2.47	103	6011	2.60
250	15024	2.52	94	15012	2.56
300	18022	2.16	65	18021	2.31
1000	60047	2.10	63	60043	2.20
4000	240139	2.00	61	240145	2.12

Table 3: Kernel Latency events depending on the timer interrupt frequency

The kernel supports timer interrupt frequencies (HZ) from 16 HZ to 4000 HZ. The arch specific configuration on x86 only allowed for 100-1000 HZ. A patch was used to extend the range of timer interrupt rates.

The effect of RT priorities is a bit disappointing. RT priorities do not significantly reduce the OS noise. RT scheduling prohibits context switches but these have a minor impact here. Pretty worrying is that in ranges higher than 250 HZ the overhead for RT scheduling increases and the timer interrupts become longer for RT scheduling despite the additional context switches that occur for SCHED_OTHER.

The duration of the timer events slightly decreases as the number of timer interrupts per second is increased. However, the change in duration does not seem to be that significant. This suggests that it may be best to reduce HZ as much as possible especially since high resolution timers are used in various places in the kernel now were accurate reaction to timeouts is important.

Since we saw that CONFIG_NO_HZ does not eliminate the HZ frequency interrupts while a process is executing it is interesting to see how a kernel would behave without CONFIG_NO_HZ. Surprisingly OS noise is significantly reduced by switching CONFIG_NO_HZ off. The number of involuntary context switches is reduced. Average durations are significantly reduced. The number of events over 1 microseconds drops by half for 1000 HZ and 4000 HZ. Switching off idle processor seems to be a very scheduler intensive activity. It is good for power consumption but it does not reduce the system noise as one would have expected.

HZ	Events	Duration	CSw
18	1084	7.37	62
60	6016	2.02	63
100	6037	1.46	98
250	15016	1.87	61
300	18018	1.40	62
1000	34597	1.30	79
4000	151744	1.25	92

Table 4: Kernel Latency events for a system with ticks during 60 seconds

7 Conclusion

The noise is there in the Linux kernel and it is gradually increasing as the kernel gets bloated with new features. I think it is necessary to keep an eye on the latencies created by the OS since we are seeing regressions when using newer kernels for latency sensitive applications.

In order to reduce the noise created in the Linux kernel we need to go beyond the real time scheduling policies (SCHED_RR and SCHED_FIFO). The following measures may be useful:

- Not running a timer interrupt if not necessary. Could we have a true tickless system? Currently Linux claims to be tickless but the truth is that a tick still is used when a process is running. A tick makes sense if multiple processes are contending for time on the same processor. If there is no other process at the same or higher priority contending then there is no need for a timer interrupt until the processor voluntarily gives up the time slice or until another process is created that can contend for the processor. We already have high res timers. Is it not possible to calculate how long a process is allowed to run and have the scheduler processing only occur when we reach that point?

  If multiple processors are contending for a processor and we assign a time slice to a processor then there still is no reason to run an timer interrupt before the end of that time slice. The OS needs to have a concept of an on demand timer interrupt that is only enabled on request.

- The scheduler needs to be more aware of the cache relationships between multiple "CPU"s that the OS knows about. The chance is good that threads of the same process will share data and therefore it is essential that the scheduler put threads of the same process on cpus that share CPU caches. If a process is running on one CPU and is marked as latency sensitive (using SCHED_RR and SCHED_FIFO) then scheduling on a sibling needs to be avoided as much as possible to leave the CPU cache undisturbed.

- It may be useful to make processor 0 a special processor that is used for system tasks. It could have a special role that the scheduler is aware of (comparable to Solaris). Processor 0 is already special because it is running a timer interrupt that is tasked with keeping system time. Therefore the noise created by processor 0 is already increased. Non latency sensitive tasks could be scheduled on processor 0 to keep needless noise away from the other cores. High priority tasks can then be scheduled on the other processors as needed whereas lower priority user space tasks (such as the regular daemons) could be mostly scheduled on processor 0.

- Processor 0 could take over tasks from other processors (like scheduling for idle processors). If a processor is busy and no CPU specific events are scheduled on a processor then processor 0 could take over managing the run queue and interrupting the target CPU as the need arises.

8 References

Beckman, P., Iskra, K., Yoshii, K., and Coghlan, S. 2006. "Operating system issues for petascale systems." *SIGOPS Oper. Syst. Rev.* 40, 2 (Apr. 2006), 29–33.

Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., and Nataraj, A. 2008. "Benchmarking the effects of operating system interference on extreme-scale parallel machines." *Cluster Computing* 11, 1 (Mar. 2008), 3–16.

Ferreira, K.B., Bridges, P., and Brightwell, R. 2008. "Characterizing application sensitivity to OS interference using kernel-level noise injection." In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Austin, Texas, November 15–21, 2008).

Van Hensbergen, E. 2006. "P.R.O.S.E.: partitioned reliable operating system environment." *SIGOPS Oper. Syst. Rev.* 40, 2 (Apr. 2006), 12–15.

E. V. Hensbergen. "The effect of virtualization on OS interference." In *Proceedings of the 1st Annual Workshop on Operating System Interference in High Performance Applications*, August 2005.

Petrini, F., Kerbyson, D.J., and Pakin, S. 2003. "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q." In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (November 15–21, 2003). Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, DC, 55.

Radojkovic, P., Cakarevic, V., Verdu Pajuelo, A., Gioiosa, R., Cazorla, F.J., Nemirovsky, M., and Valero, M. 2008. "Measuring Operating System Overhead on CMT Processors." In *Proceedings of the 2008 20th international Symposium on Computer Architecture and High Performance Computing* (October 29—November 01, 2008).

Tsafrir, D., Etsion, Y., Feitelson, D.G., and Kirkpatrick, S. 2005. "System noise, OS clock ticks, and fine-grained parallel applications." In *Proceedings of the 19th Annual international Conference on Supercomputing* (Cambridge, Massachusetts, June 20–22, 2005).

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*