Stepping Towards Noiseless Linux Environment

Hakan Akkan
Ultrascale Systems Research Center
New Mexico Consortium
hakkan@nmt.edu

Michael Lang
Ultrascale Systems Research Center
Los Alamos National Laboratory
mlang@lanl.gov

Lorie M. Liebrock
Department of Computer Science & Engineering
New Mexico Institute of Mining and Technology
liebrock@cs.nmt.edu

ABSTRACT

Scientific applications are interrupted by the operating system far too often. Historically operating systems have been written efficiently to time-share a single resource, the CPU. We now have an abundance of cores but we are still swapping out the application to run other tasks and therefore increasing the application's time to solution. Current task scheduling in Linux is not tuned for a high performance computing environment, where a single job is running on all available cores. For example, checking for context switches hundreds of times per second is counter-productive in this setting.

One solution to this problem is to partition the cores between operating system and application; with the advent of many-core processors this approach is more attractive. This work describes our investigation of isolation of application processes from the operating system using a soft-partitioning scheme. We use increasingly invasive approaches; from configuration changes with available Linux features such as control groups and pinning interrupts using the CPU affinity settings, to invasive source level code changes to try to reduce, or in some cases completely eliminate, application interruptions such as OS clock ticks and timers.

Explained here are the measures that can be taken to reduce application interruption solely with compile and run time configurations in a recent unmodified Linux kernel. Although these measures have been available for a some time, to our knowledge, they have never been addressed in an HPC context. We then introduce our invasive method, where we remove the involuntary preemption induced by task scheduling. Our experiments show that parallel applications benefit from these modifications even at relatively small scales. At the modest scale of our testbed, we see a 1.72% improvement that should project into higher benefits at extreme scales.

Keywords

OS Scheduling, OS Noise, CPU partitioning

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Operational analysis*

1. INTRODUCTION

1.1 Partitioning and core specialization

With this work we take steps towards using core specialization with the Linux kernel. Core specialization is not a new concept; it was used originally when the hardware cores themselves were specialized for various functions, the most well known was the floating point processor, but other specialized cores were used in past mainframe architectures. Today, as the number of cores has increased in processors, many scientific applications can not take advantage of the extra processing power due to limitations such as memory bandwidth and limited parallelism. Allocating all of the cores to these applications does not bring any performance benefit to the application and can extend time-to-solution because of resource contention. By allocating these general purpose cores to specialized functions, we can offload the application cores and gain performance. The simplest technique is to partition the cores between the application and the OS. These dedicated OS cores are carefully chosen to run OS tasks so we can isolate the application from preemptive interruptions and increase performance.

Our approach is to implement a soft-partitioning scheme which isolates the application from the OS, as shown in figure 1. By using increasingly intrusive methods to isolate the application, we can correlate the effort of the changes with respect to the performance gained from these efforts: from administrative configuration changes, kernel configuration parameters and finally, modifying the kernel itself to minimize the amount of non-application tasks that run on an application core.

This is a reasonable approach when the number of cores are high and the resource usage of these cores are minimal as in most OS tasks [16]. With this investigation we try to see how close Linux can be pushed towards lightweight kernels and modern kernels that have the notion of core specialization [9, 4, 10]. The result is the isolation of the application from OS jitter or noise.

1.2 OS Noise

The performance issues of OS noise have been known for over a decade and been studied extensively. Detrimental effects of noise on scalability of massively parallel applications is becoming more important with the ever-increasing

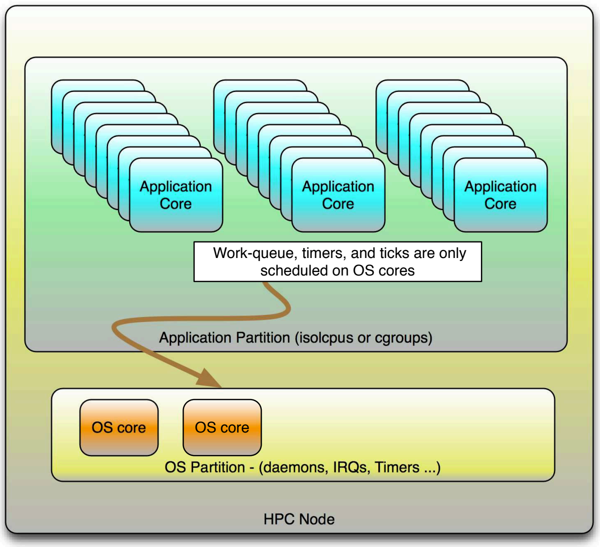

Application Core
Application Core
Application Core
Work-queue, timers, and ticks are only scheduled on OS cores
Application Partition (isolcpus or cgroups)
OS core
OS core
OS Partition - (daemons, IRQs, Timers ...)
HPC Node

Figure 1: Soft-partitioning with Linux

core counts, parallelism, and system scale. Previous research has identified the OS daemons and services as a source of noise and some of these studies concluded OS preemption interruptions, also known as ticks, to be a significant noise source that negatively affects application scalability. Scaling up to billions of parallel tasks will amplify even the smallest system noise and thus future large-scale and exascale class systems should eliminate all possible noise.

Our work aims to pinpoint the kernel-level noise sources in Linux and compilation/boot/runtime configurations to reduce this noise to a bare minimum. We describe our kernel modifications to completely remove the noise, ticks, the scheduler, and other kernel threads, and then compare it to an unmodified kernel. Our method assigns some cores in the system for OS use as *noise target* cores and tries to confine all non-application tasks to these cores - out of applications' way.

The main contribution of this paper is to show that a tickless Linux kernel can achieve similar noise signatures to light-weight or micro kernels. We describe a prototype tickless Linux system that can make noise-susceptible applications gain back the performance they may lose to OS noise. We also test our noiseless kernel with a relatively small cluster and observe the performance gains at small scales that would project into much higher gains on large-scale systems.

The rest of this paper is organized as follows. Section 2 provides an in-depth analysis of noise sources in Linux kernel. Section 3 describes our source level changes to the kernel to mitigate the noise. Section 4 summarizes the related work in this area. Finally, section 5 concludes this work and gives future directions.

2. LINUX NOISE ANALYSIS

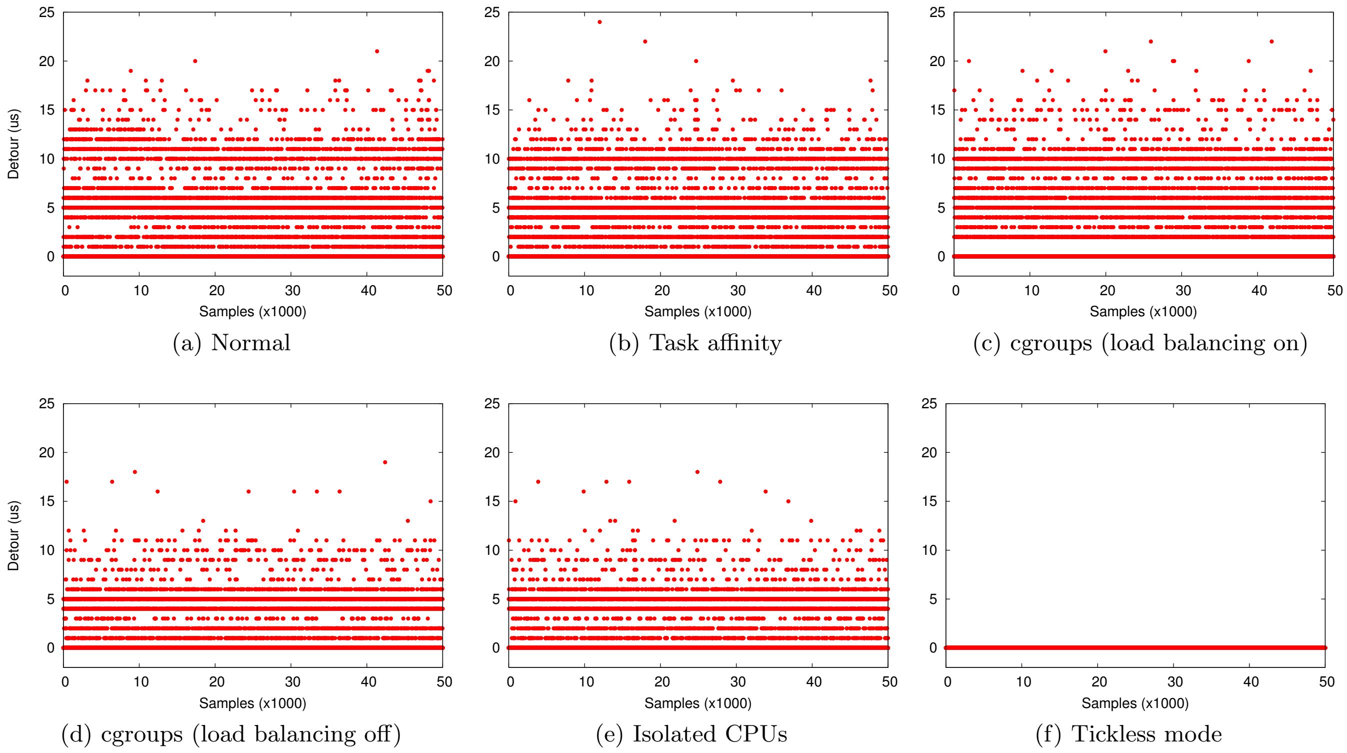
In order to evaluate the OS noise in the Linux kernel, we conducted a series of benchmarks under different compile-time and run-time kernel configurations with the goal of achieving the least possible amount of noise with a stock Linux kernel. Here we only focus on the kernel induced noise as most of the application/daemon induced noise can be eliminated by interfaces already provided by the kernel such as Control Groups [1]. By kernel induced noise we mean anything that OS does which is not directly related to the application process pinned and running on the same CPU, such as the scheduler load balancing, time keeping, and accounting. Therefore, we don't consider the transmission of network packets on behalf of the application as noise. Similarly, we don't consider virtual memory activities such as page faults as noise because they are closely related to application's memory usage patterns and Linux already supports larger pages that would reduce the number of such interrupts by a factor of 500.

The easiest way to find information about system interrupts is to read the *procfs* file */proc/interrupts* where total accumulated counts of each interrupt source is listed per-CPU since the system boot. In many systems, the highest count would belong to *Local Timer Interrupts* which is also known as the *timer tick* that the kernel arms on each CPU to update the system status (such as system time), perform I/O work, and to create preemption points. Linux developers limited the frequency of this timer to 100, 250, and 1000 HZ on x86 platform, which is configured at kernel compilation time. Several tasks are run at each tick;

- *Accounting*: Currently running process is charged a *tick* of CPU time,
- *Updating time*: Global kernel time value (*jiffies*) is updated,
- *Timers*: Expired timers (previously set by the kernel or user tasks) are executed,
- *Preemption and scheduling*: Currently running task is preempted if it exceeded its scheduling time quanta or another higher priority task has became runnable,
- *Bottom half handlers and workqueue items*: Long-lived interrupt processing and delayed work items are checked and executed.

In an HPC environment, usually all of the CPUs in a node are used by the application and it competes for CPU time with system services and daemons. Job launchers usually *pin* each process of the application to a single CPU which prevents the kernel from migrating application processes away from their CPUs, thus increasing the locality and reducing the load-balancing overheads. However, system services are usually not pinned and thus migrated often by the kernel on application CPUs for the sake of load-balancing. This means application processes are involuntarily preempted to run system services which introduces delays to the parallel application. This can be remedied by encapsulating system services inside a set of CPUs and running the application in the remaining cores. An obvious problem with this approach is not using all of the computing power in the machine, but Petrini, et al. showed that using one less CPU per node on a 2048-node system constructed from 4 core nodes, reduced application time to solution, even with the 25% reduction in processing power [15]. This is even more prevalent now with the advent of many-core processors and the limited memory bandwidth available to applications.

The kernel *Control Groups* (cgroups) mechanism allows creating virtual hierarchical partitions for subsets of CPUs



(a) Normal (b) Task affinity (c) cgroups (load balancing on)

(d) cgroups (load balancing off) (e) Isolated CPUs (f) Tickless mode

Figure 2: Fixed Work Quantum benchmark results with different configurations

and other partitionable resources in the system primarily for isolation and accounting purposes. Some HPC job launchers (SLURM for instance) leverage cgroups to create a virtual fence around the jobs to prevent them interfering with each other and other system services. cgroups also allow turning off the scheduler load balancing efforts among the given set of CPUs in which case processes have to be pinned to allotted CPUs by the programmer.

It is also possible to strictly isolate CPUs in a Linux system using the *isolcpus* boot option which disables load balancing and prevents the scheduler from scheduling any user task or a system service/daemon on the isolated cores unless otherwise told by the task using specific OS interfaces such as sched_setaffinity system call or cgroups. However, kernel tasks and daemons are still created and scheduled on each core in a non-configurable way that still causes noise.

We measured system noise under different conditions using the Fixed Work Quantum (FWQ) benchmark from the Netgauge tool [3]. FWQ measures and successively records the time it takes for a short fixed workload. Figure 2 shows the detours-the amount of delay the benchmark experienced during each work quanta due to noise- under different conditions on a 4-socket 6-core AMD Opteron machine with stock 2.6.38 kernel configured with 100 HZ. All tests collected 50K samples and were run with 16 MPI processes. For the experiments run with task affinity on, we pinned them to cores 3-6 on each NUMA domain - leaving the first two cores free for OS daemon and hardware interrupt usage. Figure 2a shows the detours without any efforts to reduce the noise. Figure 2b shows the detours when task affinity was set. Figures 2c and 2d show the detours when the entire MPI job was restricted in a cgroup with the scheduler load balancing turned on and off, respectively. Finally, Figure 2e shows detours when task CPUs were isolated during machine boot and tasks were explicitly pinned to these cores.

Figure 2 shows that the least noisy runs with an unmodified kernel were achieved when the scheduler load balancing was explicitly turned off either with *cgroups* or isolated CPUs using *isolcpus* boot option. Sole task pinning, which is usually how application isolation is currently done in the HPC world, does not necessarily prevent other tasks and system daemons from sharing the application cores and the level of noise remains the same (Figures 2a and 2b). Figures 2c and 2d show the effects of scheduler load balancing from noise perspective; applications experience a higher level of noise due to the scheduler load-balancing efforts.

To identify events happening during ticks, we used built-in kernel function tracer, ftrace [2], to trace the kernel. This tracer, when turned on, records entry times to each kernel function in a CPU-local buffer with nanosecond resolution. From this data, we calculated the processing length of each tick which is shown in Figure 3 (on an isolated CPU during FWQ benchmark). Data points below 25 nanoseconds are for exactly the same sequence of events: time keeping, accounting, checking for preemption, and, checking for any delayed work. The variance between the samples is due to cache line evictions between different cores accessing same global data structures such as locks or kernel time. Longest processing times were recorded as points between 30-35 us and happened once every second. From the tracing log, we found that these long ticks happened when virtual memory (VM) status updating was triggered. It wakes up a kernel thread to process a work item which in turn calls a function in the memory subsystem to refresh virtual memory usage

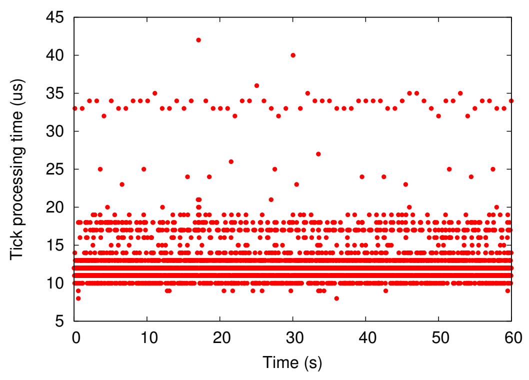


Figure 3: Tick processing times on an isolated CPU

statistics on that CPU.

Kernel threads in Linux are similar to user processes but they run in kernel mode. This means they have scheduling policies and priorities too. When we changed the scheduling policy of the FWQ benchmark from Linux's default completely fair scheduler to real-time scheduler with the highest possible priority, we found that VM status update threads were never scheduled due to their lower priorities, thus removing the values between 30-35us in the Figure 3. If applications use a higher priority than kernel-level non-critical threads, it might be possible to achieve a consistent low-level OS clock tick noise profile with task pinning and turning off the scheduler load balancer. However, even with the real-time policy, FWQ benchmark gave the same results; the real-time scheduler is not enough to reduce the noise footprint of the OS.

We also ran scientific codes and popular HPC benchmarks in the same 24 core machine and found that there is a slight (less than 0.5%) performance gain with the lowest OS noise footprint compared to other configurations. We believe that the insignificance of the performance gain is due to relatively small scales; exascale-grade hardware is expected to contain many more CPUs per node and the performance gain would be amplified in that case. Another interesting finding was the elimination of the variability in the running times of applications. Although the average running times did not change significantly, variability almost disappeared.

In this section we showed the lowest achievable noise footprint in the Linux kernel with runtime and compile time configurations. However, this footprint is very similar to the noise that was identified to be a scalability bottleneck to BSP applications at high scales [17]. We argue that the time spent in tick processing is not needed by the application and it only causes delays and can be removed or moved to spare cores. A complete removal of the ticks might be necessary to achieve a consistent noiseless and scalable system. We address this in the next section.

3. TICKLESS LINUX PROTOTYPE

In this section we describe our modifications to the Linux kernel eliminating any interruptions to the application. Our goal is to move or remove unnecessary and redundant tasks that are carried out on the application tasked CPUs into dedicated OS CPUs while maintaining the correct kernel execution.

As we described in the previous section, the first and foremost source of noise is the clock ticks. Benefits of removing this tick is many-fold. First of all, we prevent parallel applications from getting unnecessarily preempted. This also prevents thrashing the application's cache, because anything that the kernel accesses during the tick will evict some application data in caches. Furthermore, kernel accesses mostly CPU-local data but also changes global kernel data during each tick such as locks or global kernel time. Global data that is changed on a core would cause cache line evictions in other cores in the system, which is a performance penalty and scalability bottleneck. Therefore, it is desirable to avoid the clock tick on all possible cores and move the critical functionality that requires the periodic ticks to a designated CPU.

Most of the actions taken during clock ticks are not relevant in the HPC context, where only one task is running on each CPU. For example, synchronizing the hardware time with the kernel time is irrelevant to the application running on that core; this can easily be, and is already, handled by other CPUs. Also, process accounting in the context of high performance computing is irrelevant to the application. However, there is more to each tick than the idle case in the presence of I/O. Linux processes most of the I/O operations in its *bottom half* handlers, which are used for separating the acknowledgment and processing of hardware interrupts. This reduces latencies for applications that have real-time requirements as the kernel spends less time in uninterruptable mode. Bottom half handlers, a.k.a. *softirqs* in Linux, run before returning from hardware interrupts or are scheduled just like other user processes in some cases. This means if there is no hardware interrupt received on a core at all (such as the timer interrupt), kernel functionality that handles I/O services are not performed on that core. Furthermore, if there is no I/O device interrupt received at a core, there can still be bottom half work items queued by remote CPUs for the sake of load distribution. For example, *Generic Receive Offload* and more recent *Receive Packet Steering* mechanisms in the networking stack map and forward each incoming packet to a remote CPU for the packet's processing rather than handling them all on the CPU where the interrupt from the network device is received. Naively turning off the ticks would not only reverse this load distribution, but also would prevent the bottom half handlers from running on tickless cores and performing critical tasks.

There are also subsystems that require collaboration from each online CPU in the system. For example, the *Read Copy Update* (RCU) mechanism requires each CPU to report a quiescent state in order to start a grace period where deallocation of shared data structures are handled. If a CPU does not report a quiescent state for some period, the kernel forces it to do so by sending an inter-processor interrupt (IPI) to the offending CPU. There are several other mechanisms in the kernel that send IPIs to remote CPUs and are obstacles for the idea of uninterrupted application execution. A recent engineering effort by Ben-Yossef has identified these interrupts and eliminated most of them where possible [5].

We also found that many device drivers and system daemons/services request some work to be done in the future to do some bookkeeping or updating status. Many different classes of device drivers, from audio to InfiniBand, schedule a worker thread with the same frequency as the clock tick for periodic maintenance and other reasons[1]. These work item requests are queued on per CPU lists as well and processed together with the bottom-half handlers.

Although there are benefits to isolating application cores from clock ticks, there are negative effects as well for the aforementioned cases. We believe that most of these issues can be remedied by only allowing bottom half processing on the OS cores without performing any other task. Bottom half items are usually processed within a short time budget to prevent the starving of the userspace application for CPU time. A soft-partitioning scheme in this case can be leveraged to create one or two OS cores per NUMA domain that only handle bottom half items on behalf of the application cores on the same domain. On these OS cores, the time budget could be increased or completely removed to allow the kernel to make larger batch processing. This may help restoring any bandwidth lost to concentration of the previously distributed load. In addition, scientific applications tend to leverage non-blocking communication to overlap the calculation and communication for better performance. If the I/O processing can be done in an OS core and the application core is not perturbed in the mean time, we can remove the noise from the application core and completely parallelize the communication.

Timer related events are managed by the *high-resolution timers infrastructure* (hrtimers) in the kernel and our approach to create a tickless Linux leverages this system. hrtimers manage a per-CPU based queue of timers ordered by the expiry time and they arm their corresponding hardware timer chips to interrupt the CPU only at the nearest event. The scheduler, for example, initializes an hrtimer to call the main scheduler tick function during the machine boot and forwards that timer with HZ frequency at every tick. We made a trivial modification to that function to forward the timer much further in the future rather than with HZ frequency. This change only triggers when the application requests a tickless environment with a system call and it is the only runnable process on its core at that moment. In that case, the CPU switches to a *no tick* mode where further timer and work item requests are queued to the corresponding OS core, which is always calculated as the first CPU in the application core's NUMA domain. With this environment, applications still received interrupts due to IPIs sent from remote CPUs for the services that require global collaboration from all CPUs. We identified one of these services as the RCU subsystem and implemented necessary hooks to prevent interruption of *no tick* CPUs. After these modifications, application cores did not interrupt tasks running on them until those tasks exit or explicitly turned off the *no tick* mode. As shown in the Figure 2f, further benchmarks proved that the cores were noiseless with flat FWQ plots.

We then run several benchmarks to evaluate the tickless kernel on our modest 236-node quad-node InfiniBand cluster. LANL's Parallel Ocean Program (POP) code is known for its sensitivity to noise and has been a common target for many researchers who investigated the effects of OS noise on parallel applications [7, 14]. We also used POP in our test to be able to compare our results to the literature. We left the first CPU on each node as the OS core and used the

[1]The HZ macro appears just under 3000 times in about 1200 different files in the device drivers source tree. This shows the alarming level of dependency on periodic ticks by device drivers.

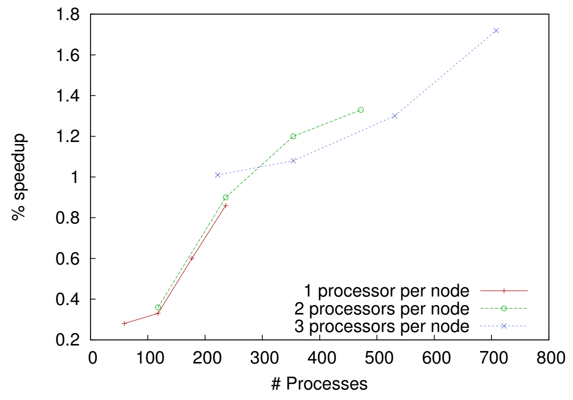


Figure 4: Increase in POP's performance with the tickless kernel

remaining three CPUs for the application.

Figure 4 shows that POP performed up to 1.72% faster in our 236-node InfiniBand cluster when the tickless mode was turned on. The fact that the performance gains increased with the increasing scale suggests that a tickless Linux kernel would mitigate the scalability issues related to OS noise at extreme scales as described in the literature. We also found that application runtime variability was nearly diminished with the tickless kernel, which we consider invaluable.

As we mentioned above, the downside to removing the ticks is the unprocessed bottom half handlers. Consequently, our tickless kernel did not allow the applications to make progress when we used the Ethernet network in the cluster. We then modified the kernel to process all network packets in the OS cores, however, this change reduced the network traffic bandwidth such that applications were slowed down by about 10% when the *no tick* mode was on. Our future work includes optimizing this behavior with increased batching on the OS cores as we described above.

Our modifications to the Linux kernel aimed to test and prove the viability and effectiveness of a tickless mode in HPC context. Hence, our prototype does not try to address all of the functionality that were handled during ticks. For example, when OS CPUs go idle and turn off the ticks as well in addition to the application cores, there is an absence of CPUs that update the global jiffies value which breaks the time keeping machinery. We are currently in the process of addressing such deficiencies and improving the stability of our prototype. Nevertheless, our hypothesis, the tickless Linux, appears to be viable.

We argue that the Linux timing system and scheduler can be improved by allowing the core scheduler to make decisions on the time based events in the system. Currently, the scheduler is just another subsystem that depends on the timer/hrtimer infrastructure and is invoked upon a timer expiry. However, if the scheduling decisions were made at the point at which expiring timers hit first, then the scheduler would be able to control and adjust all timer based events in the system based on the scheduling policy and other tunables suitable for different platforms such as laptops, desktops, small clusters, and HPC environments. Such a design would even ease the implementation of *smart timers* as Tsafrir, et al. described [17].

4. RELATED WORK

4.1 Partitioning

Recently, an HPC lightweight OS, Nix, introduced the idea of *application core* (AC) where applications are exclusive owners of their CPUs and are never preempted, which essentially eliminates all the OS noise [12]. ACs never run kernel code or receive interrupts. Our prototype borrows this idea from Nix and makes it available to the much wider user base of the Linux kernel.

Several papers [15] discuss disabling unneeded daemons and mention using one less core per node to allow the OS to naturally schedule noise on the free core. Others [8] have changed priorities of application threads to force co-scheduling of noise, but these methods are not really partitioning as they do not directly enforce the placements and they do not address kernel level noise.

Tessellation is a research kernel developed for space-time partitioning targeted at manycore processors [10]. Some ideas include partitioning of application from OS and taking into account energy and quality of service. Tessellation is still in an early development stage.

Several vendors have been providing specialized solutions for system partitioning. For example, IBM's supercomputers from the BlueGene family assign one processor per compute node for OS and allocate the rest for the application's use. However, these operating systems are tailored for specific hardware and generally provide reduced flexibility in OS services for the sake of performance. A tickless Linux may as well provide the same level of application isolation without a similar restrictive environment.

4.2 Noise

Much research has been done to study the effects of different types of OS noise on parallel applications. Early work [15] examined the cost of unsynchronized system daemons that communicate frequently and found that they can cause slowdowns to BSP applications. Performance improvements were observed in such systems by leaving one core idle for OS tasks.

One of the most thorough studies was by Hoefler [7], which used an analytical model and simulator to estimate noise effects at scale and showed that dependencies in point-to-point communications can be a problem in addition to collective communications. Ferreira, et al. manually injected high/low frequency short/long duration noise into a large scale parallel application on a noise-free light-weight kernel system to characterize the application's response to various noise profiles [6]. They also investigated absorption of noise when applications are paused in communication waits.

The work by Tsafrir [17] suggests the biggest issue is high frequency noise such as the OS clock ticks. A similar study by Morari, et al. implemented a tracing toolkit into the kernel to disambiguate every single noise event [13]. This tracer can provide a function-by-function quantitative analysis of kernel noise events which might be useful for pinpointing the main contributing noise factors per application. Our work differs from these studies by providing methods to mitigate these noise sources with configuration and source level changes rather than only analyzing the noise.

Oakridge National Laboratory and Cray have investigated a reduced noise kernel [14], but this work mostly focused on moving non-kernel tasks to core 0 and with no intrusive kernel changes described. They move noise generating tasks to one single OS core and make it a job-based decision whether to allocate this single core for system services, depending on the application's sensitivity to noise.

The idea of tickless kernel in fact is already implemented in the Linux kernel for different reasons. It was introduced into the mainline kernel in version 2.6.21-rc1 and is selected with the CONFIG_NO_HZ compile-time option. This mechanism is only activated when there are no runnable tasks on the CPU - an idle CPU. It is focused on power savings as the ticks are turned off only when the core is idle.

There is very recent work by Tilera to isolate ticks and another very recent draft for inclusion into the Linux from Weisbecker [18] to add a *Nohz* option to the cgroup subsystem. Our prototype is separate from these implementations but we recently started a collaboration to integrate a tickless mode into the mainline kernel.

5. CONCLUSION AND FUTURE WORK

With this work we analyzed the Linux kernel with increasing levels of configurations to reduce the OS-level noise. We showed that it is necessary to implement source-level modifications to achieve a pure noiseless system. We then described a series of modifications to the kernel to eliminate the OS clocks ticks by creating a soft-partition for the application that is mostly isolated from the kernel. These modifications resulted in a tickless Linux prototype, which we use to show that OS clock ticks are detrimental for scalability of bulk-synchronous parallel applications and it is possible to achieve a noise-free system with one of the most popular commodity operating systems.

All previous investigations have agreed on the relationship between the increasing effects of the noise and the scale of the machine. The probability of one of the processes being interrupted increases with the number of processes in the system. Our improvements are small at the scales we have been able to test, but as all literature show, noise effects are amplified at extreme scales. We emphasize that 82.8% of the current top 500 supercomputers are running Linux and they all are losing compute cycles to ticks and limiting application scalability [11].

Contrary to previous research, which either investigated the application behavior in the presence of noise, or explained the frequency and extent of things that run during the ticks function-by-function, we explored methods to move the noise away from application CPUs. We hope that our work will help system software and OS developers in making decisions that might be critical in HPC context.

We identify our future improvements to this work as:

- Moving the process time accounting and the time keeping functionality from the timer interrupt code to general kernel entry/exit points so that it can be done in *no tick* mode as well with each system call/interrupt,
- Moving the bottom half processing from application cores to OS cores with a higher batch processing opportunity,
- Identifying critical and non-critical kernel threads that need to be moved to OS cores,
- Optimizing the cases where requests made by the application itself are not forwarded to OS cores if they can be handled in the application core,

- Testing our kernel with much higher scales and reporting the results back to the Linux and HPC community.

6. ACKNOWLEDGMENT

The authors would like to extend thanks to Abhishek Kulkarni, Hugh Greenberg, and Latchesar Ionkov. This work was performed at the Ultrascale Systems Research Center (USRC), a collaboration between Los Alamos National Laboratory and the New Mexico Consortium (NMC). This publication has been assigned the LANL identifier LA-UR-12-21128.

7. REFERENCES

[1] Kernel control groups documentation. `http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, Mar. 2012.

[2] Kernel function tracer documentation. `http://kernel.org/doc/Documentation/trace/ftrace.txt`, Mar. 2012.

[3] Netgauge - a network performance measurement toolkit. `http://www.unixer.de/research/netgauge/`, Mar. 2012.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *SOSP'09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM.

[5] G. Ben-Yossef. Reduce cross CPU IPI interference. `http://lkml.org/lkml/2012/2/9/59`, 2012.

[6] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Nov. 15–21, 2008.

[7] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[8] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *SC'03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society.

[9] S. M. Kelly, R. Brightwell, and J. V. Dyke. Catamount software architecture with dual core extensions. In *Proceedings of the 48th Cray User's Group Conference (CUG 2006)*, Lugano, Switzerland, May 8–11, 2006.

[10] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *HotPar09*, 2009.

[11] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top500 report for November 2011.

[12] R. Minnich. NIX 64-bit kernel is available. `http://9fans.net/archive/2011/09/145`, 2011.

[13] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero. A Quantitative Analysis of OS Noise. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11.

[14] H. S. Oral, F. Wang, D. A. Dillow, R. G. Miller, G. M. Shipman, D. E. Maxwell, J. L. Becklehimer, J. M. Larkin, and D. Henseler. Reducing application runtime variability on Jaguar XT5. In *Cray Users Group Conference*, 2010.

[15] F. Petrini, D. Kerbyson, and S. Pakin. 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*. IEEE Computer Society, 2003.

[16] J. C. Sancho, M. Lang, and D. J. Kerbyson. Characterizing the impact of using spare-cores on application performance. In *Euro-Par 2010*, Aug.–Sept.

[17] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th ACM International Conference on Supercomputing (ICS05)*. ACM Press.

[18] F. Weisbecker. Nohz cpusets. `http://lkml.org/lkml/2012/4/30/380`, May 2012.