The Effects of Long SMIs on Intel Hyper-Threading Technology

Chris Giossi, Ben Hamlin, Swati Jagdale, Devin Sills, and Li-yun Wang

Portland State University

1. Introduction

System management mode (SMM) is a special mode on x86-family processors designed for power management and other board-specific software. Although it is relatively unobtrusive when used in moderation for its intended purpose, a recent study by Delgado and Karavanic [3] has shown that it can adversely impact performance as the time spent in SMM increases.

In particular, some runtime proposed integrity measurement mechanisms (RIMMs), a variety of security software, would increase the time spent in SMM beyond what they were designed for. Since running code in SMM preempts all processes, including kernel processes, and suspends execution on all cores of an SMP system, the performance impacts of increased time spent in SMM bears further study.

The question this paper asks is, "How does Intel Hyper-Threaded Technology stand up to the performance impacts of SMM?" Do the shared processor resources hyperthreading entails mitigate the adverse effects of SMM? Or conversely, does SMM have a negative impact on the kernel's ability to take advantage of those shared resources?

Our hypothesis going into this project was neither of these: We reasoned that increased time in SMM would affect hyperthreaded and non-hyperthreaded cores symmetrically. That is, as the frequency of SMIs increased, the system's performance would decrease at the same rate, whether hyperthreading was enabled or not. In graphical terms, we thought this would manifest itself as an upward curve as the frequency of SMIs increased on a system with, say, four non-hyperthreaded cores, and an identical (although perhaps vertically displaced) curve when the kernel had an additional logical core at its disposal.

Our hypthesis was borne out in some cases, but with some interesting counterexamples. We found that, in most instances, the performance degredation due to increased SMI frequency was indeed symmetrical between hyperthreaded and non-hyperthreaded configurations. However, we found that when we ran a benchmark with a very high cache miss rate on a hyperthreaded configuration with very frequent SMIs, its performance became erratic. In addition, another of our benchmarks, which showed improved performance as in general as hyperthreading was enabled, was affected disproportionately by SMM effects when we made SMIs very frequent.

In section 2 we provide some background on SMM and hyperthreading. Next, in section 3, we describe the methods we decided on for testing our hypothesis. Section 4 is the hardware we tested on. In section 5, we present and discuss our results. Finally, we sum up our conclusions in 6.

2. Background

In this section, we provide some background information on SMM and hyperthreading. Although both of these topics could be discussed at great length, our focus in this paper will be on aspects of smm and hyperthreading that might cause them to interact in ways that might have performance impacts.

i. SMM

System management mode (SMM) is a special execution environment on Intel CPUs. Since SMM has higher privileged priority than both user and kernel processes, it can monitor and control the entire system to perform various low-level tasks such as power-management. Some research has also proposed using SMM to enforce system security [3].

The significant feature in SMM is that SMM has the highest privilege in the system, which means other applications cannot access resources in SMM directly. Furthermore, these applications do not have any authority to measure the duration of time spent in SMM. SMM can be triggered by system management interrupts (SMI). There are several reasons to trigger an SMI. These reasons include system errors reporting, system checking, and power capping. For the above reasons, SMI is easily generated and causes system delay when system enters SMM. Due to SMM impact, the whole system performance is decreasing, and the whole system needs longer execution time and has longer completion time.

ii. Hyperthreading

a Hyper-threading (HTT) is Intel's implementation of Simultaneous Multithreading (SMT). SMT is a technique in superscalar architectures that allows multiple instructions to be issued from different threads, thereby introducing thread-level parallelism (TLP) in the CPU [9]. Specifically, HTT exposes two logical cores to the operating system (OS) per physical core. The OS can then schedule different threads on these cores [5]. A hyperthreaded CPU achieves SMT by having duplicate registers per physical core so that each logical core has its own set of registers [6]. Each pair of logical cores also shares the same cache, which has a significant impact on performance [10, 4].

In practice, the performance benefits of HTT are application-dependent. This is a result of the assumptions underlying HTT. Intel's implementation uses the heuristic that, in general, a single thread will not always use CPU resources while in the pipeline. Cache misses, for example, will cause an instruction to wait.

HTT aims to close such gaps in CPU-resource usage by executing another thread during these gaps. Therefore, to benefit from HTT, there must be some resource utilization gaps for HTT to fill. Leng, et al., found that applications performing intensive floating-point operations do not benefit from HTT because they already utilize CPU resources efficiently [7]. A NASA study concluded that structured applications, which "...access adjacent elements of the underlying data structures...", are usually cache-optimized by the compiler such that HTT has few opportunities to improve throughput by filling gaps [11]. I/O-bound applications, however, can benefit from HTT since one thread can execute while another is waiting on I/O [7].

Additionally, because HTT-cores share

cache, the cache behaviour of two threads on an HTT-core will affect performance. For example, two cache-friendly threads can compete with one-another and cause more cache misses than would otherwise occur with a single core and thread. However, if the two threads exhibit cache-synergy, i.e., they happen to require one another's cache items, then they may both benefit. Database researchers used this latter point to reduce cache misses and improve database performance. For example, they used one of the HTT-threads to pre-load the data that the second HTT-thread would require for computation. This "workahead" concept allowed them to reduce cache misses (of the second thread) by 97% and improve their performance by 33% [2].

3. Methodology

In attempting to measure the interaction between SMM and HTT, we developed an experimental procedure and used a variety of benchmarks. In this section, we describe the methods we used and our motivations for using them.

Our experiment had two main goals:

- observing the interaction between HTT and SMM and
- testing how performance differed with and without system management mode and benchmarks we used.

We accomplished this by testing several benchmark with and without HTT and with various SMI durations and frequencies.

Setting our "CPU configuration" involved using the sysfs interface to turn specific processors online and offline. Offlining a core's HTT sibling while leaving the physical core online causes the kernel to ignore the HTT sibling for scheduling purposes.

Specifically, we used settings with 1–8 logical cores: the first four being physical cores (no HTT) and the next four (5-8) being their HTT siblings. For example, five cores means four physical and one logical HTT core, six cores means four physical and two logical HTT cores, and so on. Using these configurations, we were able to isolate the effects of additional physical cores from the effects of HTT.

Beyond using physical and logical cores to test benchmarks, we also generated different SMI duration and frequency by modifying Delgado, et al.'s SMI driver [3] to observe the impact of different SMI durations and frequencies. We modified the SMI driver to test long and short duration SMIs issued in various frequencies (in jiffies). Testing various frequencies allowed us to chart the slope of SMI's impact and understand the interaction between various SMI configurations and HTT.

i. Convolve

As mentioned in ii, HTT siblings share the same on-chip cache, and the performance of HTT can be affected by cache behaviour. With this in mind, we designed Convolve, a benchmark that times concurrent matrix operations. The operation this benchmark performs is extremely common in image processing and in computer vision algorithms such as SIFT. Both of these applications are frequently performance sensitive.

In particular, Convolve simulates running a gaussian filter over an image by dividing the image into subimages, which it then doles out to as many threads as it

has available. Each thread then performs convolution using the same kernel matrix. Each thread has its own memory to write to, so there is no overhead from locking — time is spent exclusively in one of the following three activities:

- spawning threads,
- performing loads from shared memory and loads from and stores to each thread's writable memory, and
- performing ALU operations (integer multiplications and additions).

Both the image and the kernel matrix are generated outside of the timed section, which uses the Linux `clock_gettime()` function with `CLOCK_MONOTONIC`.

The size of the image, the size of each thread's subimage, the size of the kernel, and the number of threads scheduled simultaneously are all configurable. This allowed us to tune the parameters of our tests to exhibit widely varying cache behaviors. For this experiment, we settled on two configurations: one with approximately 70% cache misses (cache-unfriendly), and the other with approximately 1% cache misses (cache-friendly), both out of about 20-million cache references. The parameters for the cache-friendly configuration were

- *image size:* 0.5 megapixels
- *subimage size:* 4x4 pixels
- *kernel matrix size:* 61x61

whereas for the cache-unfriendly configuration, we used

- *image size:* 16 megapixels
- *subimage size:* 1 megapixel
- *kernel matrix size:* 3x3

In both cases, we limited the number of concurrently-running threads to 24.

In both cases, we ran our test using 1–8 logical CPUs (1–4 physical, 5–8 HTT) and for SMIs generated every 1500–50 jiffies (in 50-jiffy increments). We ran each test 3 times and averaged the results. The outcome is presented in section 5.

ii. UnixBench

UnixBench[15] is a collection of benchmarks designed to evaluate the performance of Unix-like systems. It runs a set of tests designed to measure different aspects of the system and then generates an index score rating the system against a base system. On multi-core machines, it runs each set of benchmarks twice. The first run is with a single task in parallel. The second run is with a task per core. The benchmark takes 1 hour to complete with the default settings. This combined with our methodology of testing multiple CPU configurations along with different SMI frequencies led to tests taking more time than we had available. To remedy this, we selected a subset of the benchmarks that we felt best tested hyper-threading and increased the change in SMIs we were measuring (for example, instead of measuring 100 jiffy intervals, we switched to 500 jiffy intervals). The benchmarks we decided to include follow:

- Dhrystone - Performs various string manipulations, testing the performance of the CPU.
- Whetstone - Measures floating point performance via various C mathematical functions (sin, cos, sqrt, etc.).
- Pipe Throughput - Measures the performance of how well processes can read and write from a pipe.
- Pipe-based Context Switching - Measures the performance of two processes communicating through a pipe. The two processes pass an

increasing integer to each other through a shared pipe.
- System Call Overhead - Measures how quickly processes can enter and exit system calls. This benchmark seemed particularly useful since SMIs can pre-empt system calls, theoretically causing a performance hit in this benchmark. [15]

This collection of benchmarks should give us a good indication of CPU and cache performance in our system, two areas where we expect to see some impact from SMIs. The rest of our methodology for UnixBench followed similarly to the other tests. We ran UnixBench in a loop for each of the targeted CPU configurations. For each configuration, we started the SMI delay at 1600 jiffies. We decremented the delay by 500 jiffies each iteration, until we hit a delay of 100 jiffies. Once the loop was completed, we took the total index score for each iteration and graphed the results, as presented in 5.

iii. Linux Kernel Compile

Linux kernel compile time is a widely accepted benchmark used as a proxy for real-world performance [14, 8]. This is because the Linux kernel compile is a sufficiently complex task to test how the CPU and other components perform in complex, real-world situations. The kernel compile is mostly CPU-bound, but also places stress on the cache and memory subsystems [8, 13]. Further, the kernel compile "...exercises most functions that get exercised by normal benchmarks..." [1] so it is a good approximation of expected system performance. Additionally, this benchmark is incredibly easy to run and therefore extremely reproducible and comparable. Any one of the nearly 80 million Linux users can run it [12]. It can be used to compare the performance of different systems and also different kernel versions.

Our Linux kernel compilation benchmark tested compile time under different SMI rates, SMI duration, and core counts. We increased the SMI rate by decreasing the number of jiffies we waited before issuing another SMI. Specifically, we tested compile time while issuing an SMI every 1000 jiffies, then 700, 400, and 100 jiffies. We tested these ranges for long and short duration SMIs. Lastly, we tested each of these scenarios with only 1 core, then 2 cores, until we had HTT on at 5 cores up to 8 cores, which corresponds to 4 physical cores plus 4 logical cores. Our results section will discuss the details of our findings, but generally, we found short SMIs have almost no effect on compile time and therefore do not discuss short-SMI results. Long SMIs did have an effect on compile duration, which dropped at a diminishing rate as more cores were utilized.

iv. SMI jitter

Karavanic and Delgado found that SMM causes clock tick jitter. We attempted to reproduce their results in our research. Ultimately, we identified clock jitter but it was difficult to tell if the jitter was caused naturally or by the SMIs. This was due to the fact that our kernel was tickless.

In future research, we will look into clock jitter on kernels with a regular tick versus jitter on tickless kernels. A tickless kernel does not send regular scheduler ticks to cores that are idle and also does not send regular scheduler ticks to cores that have only one runnable task [10]. This means the clock can already exhibit quite a

lot of irregularity, so detecting irregularity associated with just SMM is difficult. A further complication arises from scheduling SMIs via jiffies, which are updated on scheduler ticks and therefore also irregular.

To profile clock activity, we used ftrace to trace the `scheduler_tick()` function. Then, to circumvent the issue of scheduling SMIs using jiffy counts, we simply generated large bursts of SMIs and measured the trace file history for the effect. Ultimately, it is difficult to separate SMI jitter from tickless behaviour, but this ftrace method was able to detect clock jitter and will be used in future work to compare tickless to regularly-scheduled kernels during high SMM activity. Furthermore, it is clear from Karavanic and Delgado's work on SMM that high levels of SMIs causes performance degredation, clock jitter, and idle-CPU wakeup [3].

4. Hardware

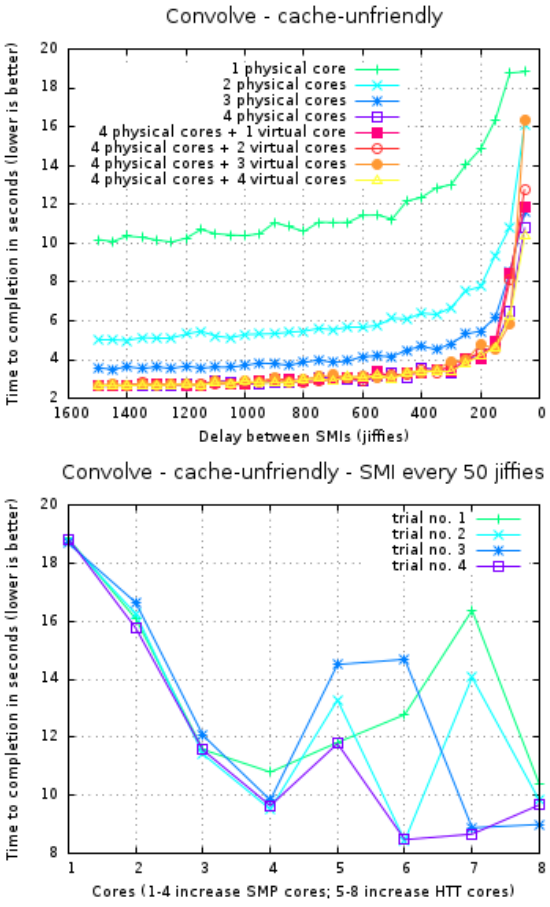
We performed all our tests on Dell Poweredge R410 servers. These use an Intel Xeon E5620 quad-core cpu with HTT in an Intel 5500 chipset with 4MB L1, 8MB L2, and 24MB L3 caches. The servers we ran our tests on had 11GB of main memory. The OS we tested on was Fedora with a real-time kernel (version 3.17.4-301).

One important thing to note is that all our tests were run on a tickless kernel. We believe that this did not have a drastic effect on our results, since most of our benchmarks kept the CPUs it was running on busy, and a busy tickless kernel scedules regular ticks just like a tickfull one.

5. Results and Discussion

i. Convolve

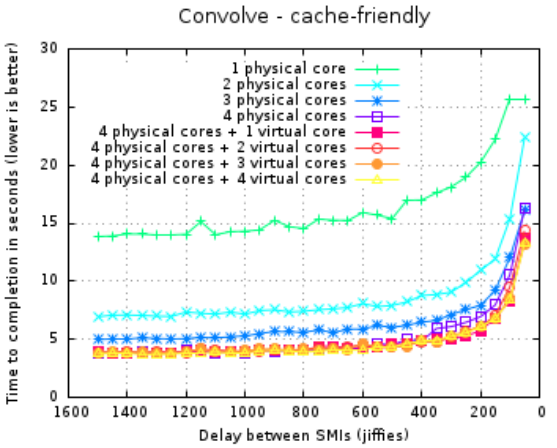
These graphs are the results of our "cache-unfriendly" (≈70% cache misses) and "cache-friendly" (≈1% cache misses) Convolve experiments. The y-axis is time to completion, so lower is better. In the upper graph in each pair, each line represents a CPU configuration, and the x-axis is how often an SMI is generated. The lower graph highlights how performance scales to CPU configuration at the highest SMI frequency we tested (50 jiffies), so the x-axis is number of cores and each line represents a separate trial.

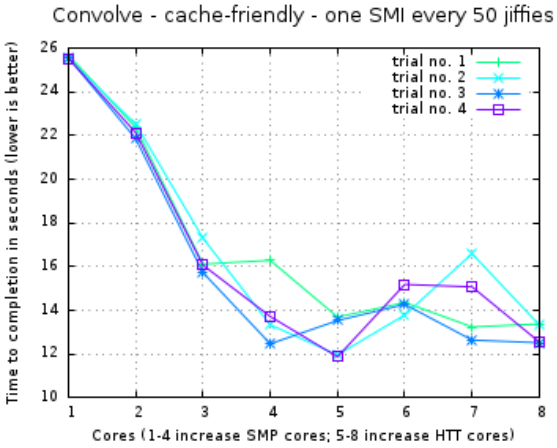


Our cache-unfriendly configuration did not benefit greatly from hyperthreading (hence configurations 5–8 largely coincide with configuration 4). This was counter to our expectations: The latency introduced by frequent main-memory acceses should allow more opportunity for the CPU to interleave processes on HTT siblings. We believe the lack of performance impact from HTT was due to cache contention — HTT siblings share an on-chip cache [5], so the performance gains HTT might have provided were offset by more freqeuent capacity misses.

Up to the highest frequency, SMP and HTT configurations were affected symmetrically, as our hypothesis predicted. As you can see in the upper graph, the performance drop caused by increasing SMI freqency is consistent across all our CPU configurations. These results were consistent across multiple trials, as shown by the left half of the lower graph.

At the highest SMI frequency, however, our results diverged from our expectations. Although the performance scaled consistently with the number of physical cores, once hyperthreaded cores were introduced, the time to completion became erratic. The completion times of 5–8 cores were sometimes better than 4 cores, sometimes much worse, and the results were inconsistent across both CPU configurations and even different trials with the same configuration.



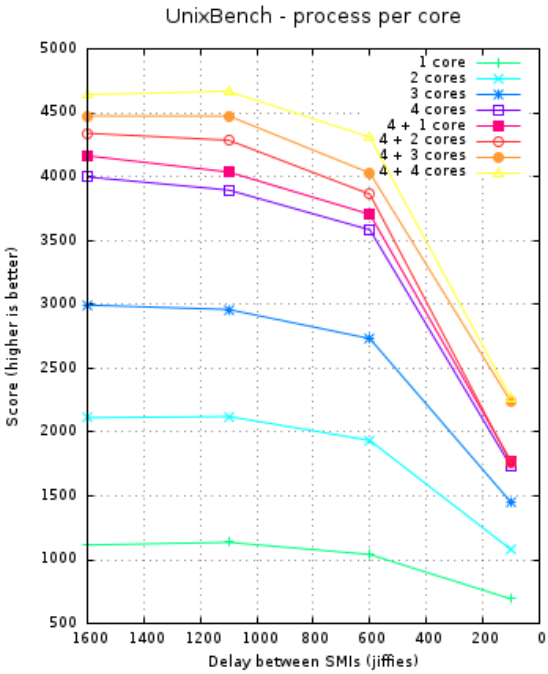


Once again, the cache-friendly configuration showed minimal benefits from HTT. This was in line with our predictions, however, following the results of Saini, et al., that cache-friendly applications frequently do not benefit from HTT [11].

This experiment showed similar results to the cache-unfriendly one with regard to the effects of increasing SMI frequency. The performance degradation is once again consistent up to the highest SMI frequency. The erratic behavior at the highest frequency is less marked, but still present.

The most interesting result of this experiment was the erratic performance we encountered with hyperthreading turned on at the highest SMI frequency. Since

the erratic behavior seems to decrease as cache misses decrease, we postulate that the effect is tied in with cache performance. We also cannot rule out the possibility that Linux's `CLOCK_MONOTONIC` is affected by SMI frequency, although this would not explain the increased effects on our cache-unfriendly configuration. In any case, we were unable to explain our results to our own satisfaction, and we feel that they bear further study.



ii. UnixBench

The below results only correspong to using long SMIs. We tried to record performance score by increasing physical cores from 1 to 4 and logical cores from 1 to 4 along with delay in short SMIs decreasing, but we did not see any change in the performance score as we decreased SMIs. Hence, we concluded that the short SMIs do not seem to have an effect on performance index of Unixbench.

This graph shows the performance of UnixBench at different SMI frequencies and CPU configurations. The y-axis is the benchmark's score, so this time, higher is better. As you can see, this benchmarks shows performance gains from hyperthreading, and as per our hypothesis, HTT configurations and SMP configurations are affected symmetrically, for the most part. At the highest SMI frequency (far right), however, HTT configurations are disproportionately affected.

As the delay between SMIs decreases, the performance score of UnixBench decreases. The threshold between 600 jiffies and 100 jiffies causes the greatest effect on UnixBench performance.

As the number of cores increases, the effect of SMIs becomes greater. Between the threshold of 600 jiffies and 100 jiffies, the slope between trials becames steeper as we increased the number of physical cores.

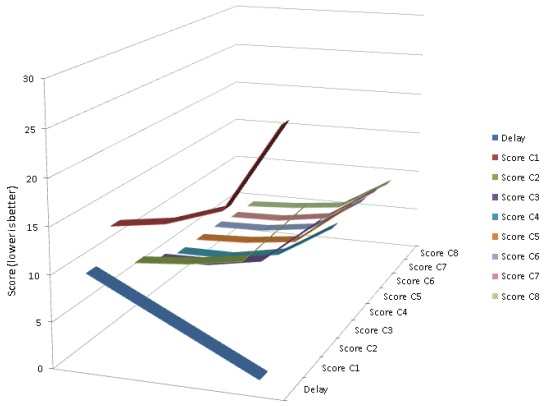
SMIs don't seem to have a great effect on hyper-threading performance. The drop in performance is similar between 4 physical cores and 4 physical cores plus any number of logical cores.

UnixBench performance seems to group around the same points for 5 and 6 cores, and has a similar effect at 7 and 8 cores. We concluded that this was due to the shared caches of hyper-threaded CPUs. We believe that cache misses effect cores 5 and 6 equally due to them sharing a cache. If 7 and 8 have a separate cache, then one

would expect a performance increase from 6 cores to 7 cores due to increasing the number of caches available. Cores 5 and 6 thrashing the cache would have no effect on 7 and 8's caches, and vice versa.

iii. Linux Kernel Compile

The Linux kernel compile time was effected by long duration SMIs. As the frequency of long duration SMIs increased, Linux compile time also increased. Please note the following graph of our Linux kernel compile time results.



The graph shows delay decreasing from 10 to 1, which is scaled down 100x for viewability. This is really a decrease from 1000 to 100. This represents the shorter amount of time we waited to issue new SMIs. Therefore, as the delay goes down, the SMI rate is going up. As the delay to issue new SMIs goes down, all other scores are seen going up. The score is the time-to-completion for a Linux kernel compile (lower is better). The graph shows the score for each core configuration (C1 is 1 core, C2 is 2 cores, and so on). Cores 1-4 are physical cores and cores 5-8 include all 4 physical cores plus additional logical cores from hyperthreading. Therefore, any effects just from additional physical cores would show up in cores 1-4 and any any effects from HTT would show up in cores 5-8. The graph shows that as more cores are added, the compile time goes down, but SMIs are still having an effect. At around 4 cores, the benefit from additional cores is minimal. We conclude that HTT and SMI do not interact in an unpredictable manner for the Linux kernel compile. For example, SMI impacts the kernel compile time on 4 physical cores in the same manner as it would impact the compile time on the same 4 physical cores with hyperthreading enabled.

6. Conclusion

For the most part, our hypothesis that increasing the frequency of SMIs would affect hyperthreaded and non-hyperthreaded configurations symmetrically was borne out. Or results differed from our expectations, however, in the case of hyperthreaded configurations with very frequent SMIs (50–100 jiffies).

In this case, we saw that UnixBench lost the performance benefits it had received from hyperthreading, and our convolve benchmark showed erratic behavior proportionate to the "cache-unfriendliness" of the process. We found ourselves unable to thoroughly explain the results we got, but we feel that they bear more study.

References

[1] "Benchmarking Procedures and Interpretation of Results." http://www.tldp.org/HOWTO/Benchmarking-HOWTO-2.html. Accessed June, 2015.

[2] J. Cieslewicz. "Improving database performance on simultaneous multithreading processors" In *Proceedings of the International Conference on Very Large Data Bases* (VLDB), pages 49–60. 2005.

[3] B. Delgado and K. Karavanic. "Performance Implications of System Management Mode." In *2013 IEEE International Symposium on Workload Characterization* (IISWC), pages 163–173. Portland, OR, USA, 2013.

[4] W. Hassanein, L. Rashid, M. Mehri, and M. Hammad. "Characterizing the Performance of Data Management Systems on Hyper-Threaded Architectures." In *Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD). 2006.

[5] "Hyper-threading." http://en.wikipedia.org/wiki/Hyper-threading. Accessed June, 2015.

[6] "Intel Hyper-Threading Technology Technical User's Guide." Intel, 2003.

[7] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini. "An Empirical Study of Hyper-Threading in High Performance Computing Clusters" Dell Computer Corp., USA, 2014.

[8] "Linux Kernel Compile." http://www.anandtech.com/show/7757/quad-ivy-brigde-ex-60-cores-120-threads/10. Accessed June, 2015.

[9] "Multithreading (computer architecture)." http://en.wikipedia.org/wiki/ Multithreading_(computer_architecture). Accessed June, 2015.

[10] "NO_HZ: Reducing Sceduler Clock Ticks." https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt. Accessed June, 2015.

[11] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas. "The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications" In *Procedings of the 18th International Conference on High Performance Computing* (HiPC). Bengaluru, India, December, 2011.

[12] "The Estimation of Linux Users." https://www.linuxcounter.net/statistics. Accessed June, 2015.

[13] "The Seven Second Kernel Compile." http://es.tldp.org/Presentaciones/200211hispalinux/blanchard/talk_2.html. Accessed June, 2015.

[14] "Timed Linux Kernel Compilation." https://openbenchmarking.org/test/pts/build-linux-kernel. Accessed June, 2015.

[15] UnixBench README https://github.com/kdlucas/byte-unixbench. Accessed June, 2015.