System Management Interrupts and Their Effects

Manjeet Bhatia, William Huiras, Vishesh Jain, Konstantin Macarenco, Josef Mihalits and Stephanie Sohn

***Abstract*—System Management Interrupts (SMIs) are explored at a fundamental level and their effects are determined upon benchmarks, including Linux kernel compilation, NBench, MPI functions, Netperf, LMBench, and CPU power state reports. Findings reconfirm that performance degradation scales with the latency of the SMI. Additional results show that SMI effects are dampened with multi-threading and that performance degradation is additive when SMIs are enabled upon multiple communicating nodes. Lastly, it was found that SMIs continue to significantly affect power consumption and CPU sleep states.**

CONTENTS

I. INTRODUCTION

A System Management Interrupt (SMI) is a method of entry into System Management Mode (SMM). System Management Mode is a mode that is more privileged than operating systems and hypervisors. SMIs pre-empt the operations of the operating system or hypervisor causing all processor cores to stop and enter SMM. Additionally, such preemption is invisible to these systems. Thus, SMIs not only take time away from the operating system or hypervisor, but they unpredictably interrupt them as well.

There exist several reasons for the existence of SMM. Originally, SMM was created to allow events specific to the machinery to be properly maintained, including power throttling, hardware emulation, and system health checks. More recently, with security compromises of hypervisors, such as Xen, SMM has been leveraged as a platform to detect malicious code injection into hypervisors, such as RIMM (Runtime Integrity Measurement Mechanisms). RIMM is able to periodically check the hypervisor code for changes by entering SMM. The use of SMM is advantageous as it lies in an unaccesible address space untouchable by malware. Nevertheless, the amount of time needed to reside in SMM in order to perform security checks is prohibitively disruptive to the normal functioning of programs in the level of the hypervisor and operating system.

Previous studies have been performed to determine the effects of SMIs [1]. These studies of SMI effects include: the measurement of scheduler tick intervals with differing SMI latencies, the measurement of percent time in C-states, audio and video driver disruption, OpenSSL and time accounting (for billing purposes), and effects upon the hypervisor Xen, including benchmarks such as Linux compilation time, TCP networking, disk reads, and compute-intensive encryption. From these experiments, it is concluded that while shorter length SMIs produce jitter, their effects upon performance are not drastically debilitating. On the other hand, long SMIs (100 ms or greater) are detrimental to performance, increasing the run times of programs, and SMIs are shown to increase energy usage, cause time scaling discrepancies, reduce throughput, and produce audio and video distortions.

We aim to both verify select experiments above, as well as to add to them. The remainder of the paper includes further experiments in the areas of fundamental SMI studies, and the effects of SMIs upon various benchmarks including: CPU bound benchmarks and Linux compilation, MPI (Message Passing Interface) routines, Netperf, LMBench, NBench, and Turbostat.

II. TESTING

A. Hardware and Software

Eight machines were used for experiments with the naming convention in the form of smitestN, where N is the number of a node in the range 2-8 (e.g. smitest2, smitest3, and so on); the first node was named "smitest". An additional external server "wyeasthead" was used for time synchronization experiments. All the nodes have the following configurations: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz, cache size of 8192 KB, and 12Gb RAM. Hyperthreading was disabled on all nodes except for smitest3. Gigabit ethernet devices included Ethernet controller Intel Corporation 82567LM Gigabit Network Connection and Broadcom Corporation NetXtreme II BCM5716 Gigabit Ethernet. For software, all machines ran on CentOS 5.10, kernel version 3.0.4.

B. SMI Driver

Driver software was written to create SMIs. It operates to produce two types of SMIs, named "short" and "long", with latencies of 1-3 ms and 100-110 ms, respectively. The driver also was coded to possess features to:

- Measure SMI latency in TSC (time stamp counter)
- Start/Stop recurring SMIs

- Count the number of SMIs that occurred using an elapsed TSC count

C. SMI Length Investigation

Background The SMI driver uses the TSC counter to measure the average SMI latency, yet we came to conclusion that this approach is not accurate enough due to TSC limitations. TSC on tested systems increases at the constant rate $R = CPU\_top\_speed$, and it reflects only the time passed between two reads, which might be affected by preemption. Thus, experiments were created to measure SMI length.

Methodology The driver schedules an SMI at the rate x, i.e. one SMI is triggered every x jiffies. In $x86\_64$ systems, one jiffy lasts one millisecond, so the driver triggers one SMI every x ms. If the SMI length and schedule interval equal exactly one second, then after N seconds the difference between elapsed time N and number of occurring SMIs will equal zero.
Let x be driver rate in ms, l be the SMI length in ms, N be the elapsed time in seconds, and C be the amount of SMIs that occurred, then:

$$D = N - C \times \frac{(x+l)}{1000} \tag{1}$$

equals zero, if

$$x + l = 1sec \tag{2}$$

SMI length tests were performed on each machine for schedule interval x starting at one SMI for every 880 ms for long SMIs, and one SMI for every 990 ms for short SMIs, and ending at one SMI for every 1000 ms. Tests were performed for $N = 500$ seconds for long SMIs, and $N = 1000$ seconds for short SMIs. SMI length, l, ranged from 1 - 3 ms for short SMIs. Pseudo-code to perform this is shown below. The algorithm was implemented as a bash script, because the load/unload kernel module is required to change driver rate.

```
#define x 880
set_driver_rate x
start_recurring_smis
// Get smicount: echo 6 >
/proc/smidriver
get_current_smi_count start_smi_c
wait N seconds
get_current_smi_count end_smi_c
save x and (start_smi_c - end_smi_c)
```

Fig. 1. Pseudo code for SMI length algorithm

The algorithm was implemented as a bash script, because load/unload kernel module is required to change driver rate.

Results and Discussion Results of SMI rate studies are shown in Figures 2 and 3. On the graphs it can be seen that there is zero deviation in the range of $x = 997$ - 999 (scheduling of 1 SMI per every 998 ms) for short SMIs and in the range of $x = 890$ - 895 for long SMIs. Therefore interrupt length, l, being $1000 - x$, allows us to determine that the accurate short SMI length is 1 - 3 ms and that the long SMI length is 110 - 105 ms. The only anomaly occurs for the case of smitest3 where, deviation equals zero in the range $x = 960$ - 965 for long SMIs. This yields the length of 40 to 35 ms. We assume that this is caused by hyperthreading, since SMM code runs in parallel and finishes faster.

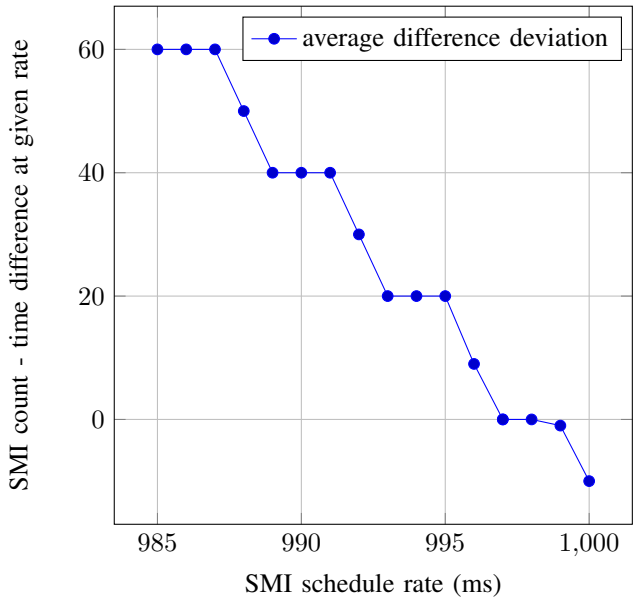

Fig. 2. Short SMIs length difference deviation at SMI rate. 0 deviation occurs when smilength + rate = 1sec

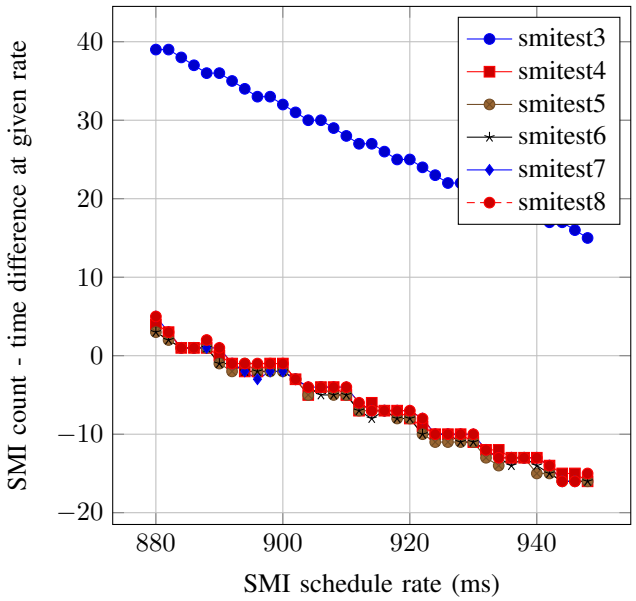

Fig. 3. Long SMIs length difference deviation at SMI rate. 0 deviation occurs when smilength + rate = 1sec

Conclusion and Future Work The above presented technique to measure SMI latency is more accurate than the method of simply reading the TSC counter. It is especially noticeable on a machine with enabled hyperthreading, where the TSC counter fails to notice the difference. In average it is

```
June 6 18:46:50 smitest
June 6 19:45:09 smitest2
June 6 18:54:42 smitest3
June 6 16:26:14 smitest4
June 6 17:43:26 smitest5
June 6 18:37:45 smitest6
June 6 13:12:27 smitest7
June 6 13:00:57 smitest8
June 6 18:44:44 wyeasthead -
    gateway (real time).
```

Fig. 4. Wall time snapshot

40% more accurate on machines with no hyperthreading enabled, and it is 110% better on a machine with hyperthreading enabled. The main disadvantage of this type of new testing is that it requires a significant amount of time to perform. In the future, it would be interesting to enable hyperthreading on more machines, and investigate in depth how hyperthreading affects SMIs.

D. System Time Delay

Background While performing the SMI length investigation, peculiar wall times were noticed on all the test machines except the gateway machine. Below is a snapshot of all systems' wall time.

This difference in wall clock times could be a coincidence caused by uncoordinated system setup, or it could have been produced by repeated testing, where recurring SMIs, especially long ones, would affect the system's wall time. To investigate if this was indeed the case further timing tests were executed on smitest and smitest6. The time difference was obtained by checking with the machine $wyeasthead$. $wyeasthead$ was chosen for this purpose because it is a gateway machine that has Internet access and thus its current time is correct. Additionally, no SMIs are invoked upon this machine, so its wall time is not affected.

Methodology For every N SMIs invoked, the time difference via the gateway was obtained. Time differences were taken every $N = 1000$ SMIs. Graphs were plotted in the following way: the X-axis is the number of SMIs that have occurred, and the Y-axis is the time difference between test and gateway machines. Tests were performed for long SMIs only, due to the assumption that short SMIs do not drastically affect the system clocks.

Results and Discussion Figure 5 displays results for two consecutive runs where the blue plot represents one run and the red plot represents another. In this figure, wyeasthead is running behind smitest3 (the difference is about 2 min), and as time passes, the difference decreases, i.e. time at smitest approaches the time at wyeasthead. The graph of Figure 6 is the same test (a single run only) but ran on smitest6. In this case, wyeasthead time is ahead, so the difference in time is widening overtime.

Conclusion and Future work The system time gets delayed a time amount corresponding to the SMI length. For this reason, the delay is not noticeable at at first, nevertheless

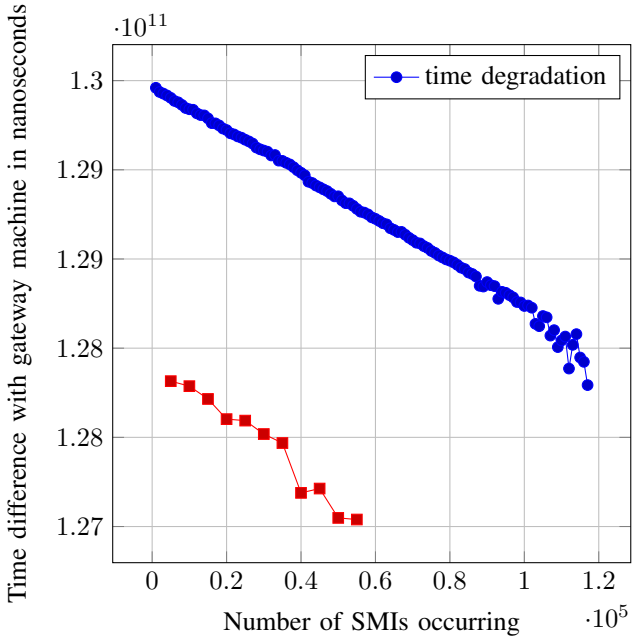

Fig. 5. Time delay: difference between smiteset and wyeasthead gets shorter overtime, since smitest is running ahead

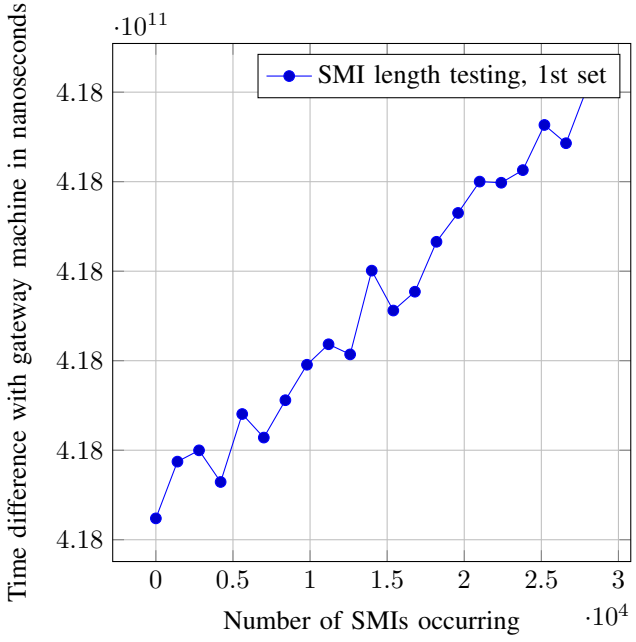

Fig. 6. Time delay: difference increases overtime since Smitest6 is behind wyeasthead

it is accumulative. This can potentially lead to a variety of synchronization problems. Interesting future studies could include performing the same test for different ranges of SMI lengths, to see the degree at which the system is affected.

E. Linux Compilation and NBench

Background Timed Linux compilation is a well known way to test the overall performance of a system. It represents a workload that is common for software development, and

it is computationally, memory, and I/O intense. NBench is a benchmark that was created in mid 90's and consists of following tests:

- Numeric sort - Sorts an array of long integers.
- String sort - Sorts an array of strings of arbitrary length.
- Bitfield - Executes a variety of bit manipulation functions.
- Emulated floating-point - A small software floating-point package.
- Fourier coefficients - A numerical analysis routine for calculating series approximations of waveforms.
- Assignment algorithm - A well-known task allocation algorithm.
- Huffman compression - A well-known text and graphics compression algorithm.
- IDEA encryption - A relatively new block cipher algorithm.
- Neural Net - A small but functional back-propagation network simulator.
- LU Decomposition - A robust algorithm for solving linear equations.

This particular benchmark was chosen due to the lack of super-user privileges on the server installation. It does not have any extra dependencies and is compatible with almost any version of Linux. The main disadvantage of NBench is that it was not designed for multicore computers. To simulate multicore behavior n instances of NBench were executed at the same time, where n is a number of available cores.

Methodology All benchmarks were tested in three modes: No SMI, Short SMIs (one per second), Long SMIs (one per second). Twenty trials were ran for each mode and averaged. In single core tests, each instance of Linux compilation and NBench was timed using the Bash $date$ function which has fine granularity (nanoseconds). To allow for multicore tests, the Linux make file can be executed with option -j n, where n is the number of available cores. Linux compilation multi-core performance was measured the same way as for single core (using Bash $date$).

As mention before, NBench has no native support for multicore, so to simulate multicore, n instances of NBench were forked at the same time, then the main script waited until all n instances completed. This was repeated 20 times. This procedure works because of load-balancing, i.e. the load is spread among all cores, therefore if $n \leq c$, where c is number of cores, then no more than one job is assigned to one core. Throughput is calculated as below:

$$Throughput\_aver = \frac{time\_to\_complete}{number\_of\_cores} \quad (3)$$

Where $number\_of\_cores$ is equal to number of NBench instances, and $time\_to\_complete$ is time for single run, in which all instances of NBench finish.

Results and Discussion Recurring long SMIs have noticeable effects on system performance. In single core tests, they slow down performance by 10-12% approximately Figure 7. In cores with hyperthreading disabled, performance suffers the most, with 75% degradation for NBench and 65% degradation for Linux compilation benchmark Figure 8. Hyperthreading softens the problem; Linux compilation performance degradation is only 12%, but NBench degradation is still 75% Figure 9. This can be explained by: (1) Long SMIs runs complete twice as fast consuming less CPU time, (2) Linux compilation might be able to execute more jobs with hyperthreading on, increasing performance (although it is not the same as having 8 cores) (3) NBench is not meant for multicore, therefore it cannot take advantage of hyperthreading.

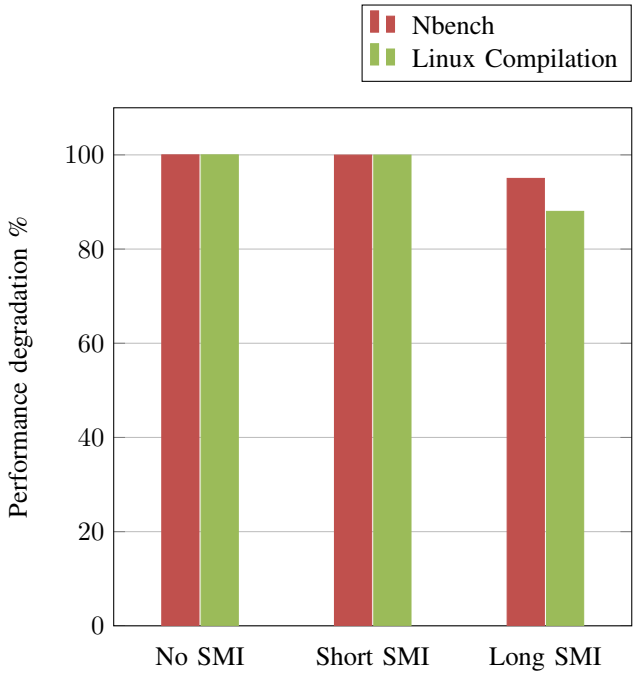

Fig. 7. Single thread performance

Conclusion and Future work There is a large difference in performance that depends upon SMI latency, so all possible ranges of SMI length should be tested to see when noticeable performance degradation begins. This can be an outline for manufacturers and industry to help determine at which point the expansion of SMI indirect use needs to be limited.

F. MPI

Background Message Passing Interface (MPI) is a standardized and portable message-passing system that functions on a wide variety of parallel computers. The standard of MPI provides the syntax and semantics of a core of library routines that allows for the writing of portable message-passing programs in Fortran or the C programming language *[2]*.

The MPI library contains many useful routines that establish the means of communication between parallel computers *[3]* . Two of the most basic routines are MPI_Send, which sends data in a buffer of an indicated size using a blocking mechanism to a specific node, and MPI_Recv, which receives a message in a buffer of an indicated size in a blocking manner

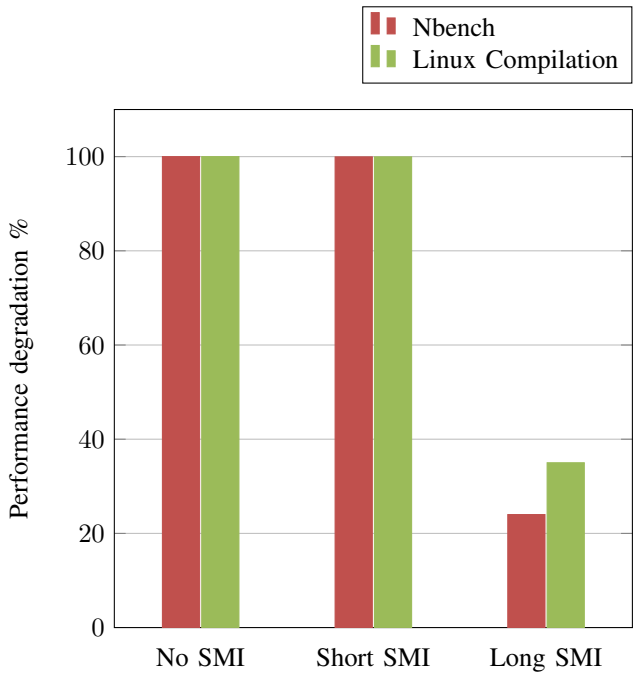

Fig. 8. Four cores, hyperthreading off

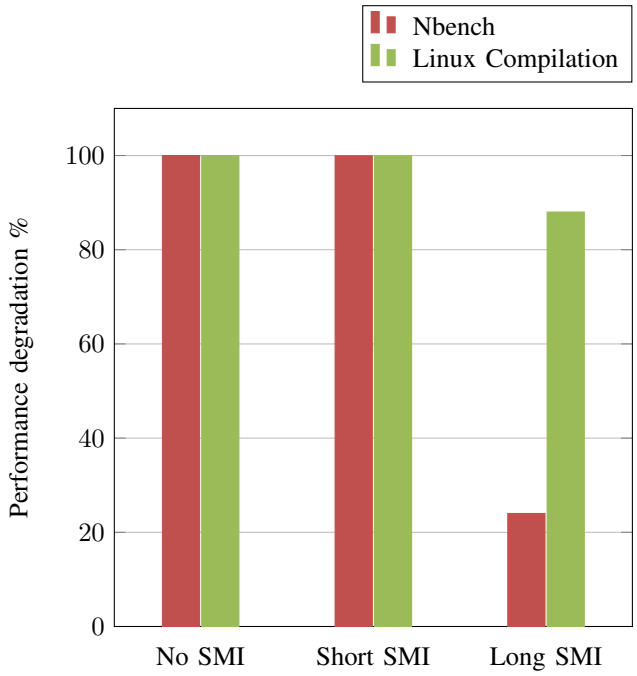

Fig. 9. four cores with hyperthreading on

from a specific node. Other message passing routines can build upon basic send and receive patterns, including routines such as $MPI\_Allreduce$, which combines values from all processes and distributes the result back to all processes, or $MPI\_Alltoall$, which sends data of an array to the correlating numbered node, i.e. data of array index 0 goes to node 0, index 1 to node 1 and so forth; in return, the calling node will receive array data correlating to its node number. Another routine, $MPI\_Barrier$, does not send data, but it is used to synchronize processes of nodes sharing the same global communicator; it causes blocks until all processes in the communicator have reached this routine.

MPI is interesting to examine in the presence of SMIs to determine the effects of the SMIs upon synchronization of parallel communicating computers. With SMIs enabled upon one or more computers in a cluster, it is likely that delays in one computer will exacerbate delays in all other communicating nodes. Using the above mentioned MPI routines, the effects of SMIs are examined to determine their overall implications upon cluster communication.

Methodology Benchmarks were created to examine the effects of SMIs and SMI duration upon MPI routines. The particular routines of interest that were examined included: MPI_Send, MPI_Recv, $MPI\_Barrier$, $MPI\_Allreduce$, and $MPI\_Alltoall$. The latter three were repeatedly called in a loop for either 1,000 iterations or for 1,000,000 iterations. The former two were encoded in a ring program, passing an integer, that is incremented by each node, from one node to the next based upon the rank of the node. The highest ranked node completes the ring by passing the incremented integer back to the lowest ranked node. The entirety of this ring routine was placed in a loop, akin to the aforementioned MPI routines, with iterations of 1,000 or 1,000,000. These benchmarks were tested upon 2, 4 or 8 machines, or upon 1 machine but with 2, 4 or 8 processes communicating upon this one machine. Using the provided driver to enable SMIs, times of each benchmark were collected with no SMIs, with short SMIs enabled (1 SMI per second, 0.05 ms duration of a short SMI), or with long SMIs enabled (1 SMI per second, 100-110 ms). The computer smitest3 was used to launch MPI runs with indicated parameters of the number of machines, which machines, and which executable (Ring, Barrier, AllReduce, or AlltoAll). Different versions of the executables were made to indicate the number of iterations (1,000 or 1,000,000). For tests with 2 nodes, smitest3 and smitest4 were used; for 4 nodes, smitest3, smitest4, smitest5 and smitest6 were used; for 8 nodes, all 8 machines were used. For all runs in Table I, the Linux function date with option for output of time in nanoseconds was used to determine the run time of the benchmark. These runs were an average of 3 trials. For runs in Table II, the Linux function time was used, and SMI counts, as an option provided by the driver, were collected; these runs were an average of 5-8 trials; runs with high time variance were excluded from averages. To enable SMIs for Figure 12, each smitest machine was enabled in turn, sequentially and manually, using the driver options.

Results and Discussion Benchmarks were measured to acquire runtimes. Results are shown in Table I. Figure 10 displays run times for MPI benchmarks ran on 2, 4, or 8 nodes with SMIs enabled on smitest3 only, and MPI routines are encoded to repeat for 1,000 iterations. Run times last around 0.2 to 0.4 seconds. The Ring benchmark runs the longest (0.10 seconds more) compared to the other benchmarks (Barrier, AllReduce, and AlltoAll). Nevertheless, similar results for normalized times occur for all the benchmarks (normalization calculation is time with no SMIs enabled divided by time with SMIs enabled multiplied by 100). Short SMI times

shows on average no or little degradation, normalizing to 99%. Curiously, some short SMI runs show faster run times than those with no SMIs enabled; for this table the highest normalization being 107%. When comparing long SMI runs to no SMI runs, the average normalization is 90%, matching the latency of the long SMI, which is programmed to interrupt for 1/10 of the time. No outstanding trends are noticed when comparing between MPI routine types or number of nodes (2, 4 or 8) involved in the MPI communication.

Figure 11 displays run times for MPI benchmarks ran on 2, 4, or 8 nodes with SMIs enabled on smitest3 only, and MPI routines are encoded to repeat for 1,000,000 iterations. Iterations were increased in an attempt to reduce variance. Run times last around 75 to 300 seconds for Ring and 50 to 150 seconds for the other benchmarks. Again, the Ring benchmark runs for a much longer time compared to the other benchmarks and is more effected with the increase of nodes, likely due to the serial nature of the benchmark. Ring also displays some anomalous trends when comparing to runs between SMIs enabled and not enabled, even showing shorter run times for long SMI runs. The other benchmarks are more consistent and match the trend that long SMI runs take longer by 10%, although the AlltoAll benchmark averages 3% longer runs for long SMIs. More trials are needed to determine for certain the average trend. Additionally, the factor that smitest3 had hyperthreading enabled may have contributed to these particular patterns.

Figure 12 displays run times for MPI benchmarks ran on 2, 4, or 8 nodes with SMIs enabled on all smitest machines, and MPI routines are encoded to repeat for 1,000,000 iterations. This graph provides more convincing proof that enabling of long SMIs can be a debilitating hit upon performance for parallel communicating nodes. On average the long SMI run gives a normalized time of 66% compared to 100% for no enabling. Runs with two nodes give an average normalization of 90%, 4 nodes gives 61%, and 8 nodes gives 47%, thus a trend exists upon increasing the number of communicating machines. Normalized times for the benchmarks average 67% for Ring, 58% for Barrier, 70% for AllReduce, and 69% for AlltoAll. Short SMI runs show average normalized times exceeding 100% (overall 116%) indicating that they on average were faster to complete than when no SMIs were enabled. This is especially the case for runs where 2 nodes were communicating.

Runtimes were obtained for tests upon only one machine, but with multiple processes, specifically using 2, 4, or 8 processes. This was done in order to possibly gain insight into whether SMIs affect performance of a multi-core machine. Originally, trials were attempted with only 1,000 iterations (graphed results are not provided). Although run times averaged longer than times obtained for the same tests ran upon multiple machines (averaging 1 second for almost all runs versus 0.2 to 0.4 seconds for multi-machine runs), the variance for these runs was quite drastic with run times of around 0.4 or 0.7 seconds periodically occurring amidst the same runs of around 1 second duration; even in the case of AllReduce with 2 processes (short SMIs enabled), all 3 trials were 0.05 seconds. When these aberrations were removed most runs showed very slight differences between enabling SMIs and not.

Thus, the same measurements were ran with iterations of 1,000,000 as is shown in Figure 13. Normalized times gave 105% for short SMI-enabled runs and 94% for long SMI-enabled runs. Significant differences were not observed between normalized times for 2, 4, and 8 processes, although 8 process runs tended to be slower by around 3%. Long SMI runs tended to be around 90% when considering the benchmark averages, except for Barrier which ran faster than normal and averaged a very high 106%.

Further tests were conducted with the one computer/multiple process set-up and are shown in Table II. These runs were only performed for the AlltoAll benchmark. Times were analyzed simply in comparison with times with no SMIs enabled (in the columns "% Normalized Time), and times were incorporated with SMI counts to calculate "% Remaining CPU clocks" in the same fashion as done in [1]. Upon learning that SMI latencies may not be consistent for smitest3 versus all other smitest machines (later discovered to be due to the hyperthreading setting enabled for smitest3 only), as explained in Section II-C, similar tests were conducted upon smitest5. Two prominent features are evident. First, 2 process runs average less than half the time on smitest5 (although an occasional run did yield around 1.5 seconds with one SMI occurring on this machine). This leads to the difference in % remaining CPU clocks. Second, there is a drastic difference in run times with 8 processes; additionally the % normalized time varies largely for this run, with 93% for smitest3 and 76% for smitest5 although % remaining CPU clocks are nearly identical. Thus, it appears that the use of hyperthreading allows for the runs with process number greater than number of cores to finish faster and to finish within similar time (only 8% longer) to a run with no SMIs enabled.

Conclusion and Future Work

Tests with MPI programs, although not thoroughly conclusive, do provide some additional insight into the effects of SMIs upon parallel, communicating computers in a cluster. The most prominent conclusion is that long (100 ms) SMIs incur an additive penalty upon performance. For the benchmarks tested with long SMIs enabled upon all machines, the ratio between time with long SMIs enabled versus no SMIs enabled can be as much as 2.5 times longer, with 8 node communication. Less dramatic effects are witnessed when SMIs are enabled upon only 1 machine of the cluster, nevertheless a performance penalty is still incurred; such penalties are relative to single core, single machine benchmarks, usually matching the latency of the SMI. Another observation is that short SMI runs sometimes produce faster run times, which is not easily explainable; perhaps some change in normal scheduling occurs because of these interrupts or the accounting of time is incorrect. Hyperthreading is another variable that may be a factor to possibly ameliorate the effects of long SMIs upon a run.

For future work, all tests should be performed again because all tests except that using smitest5 in TableII were performed using smitest3, and smitest3 had hyperthreading enabled. Also, more trials need to be performed in order to reduce variance. Further testing can be performed with the

TABLE I
MPI BENCHMARK TESTS

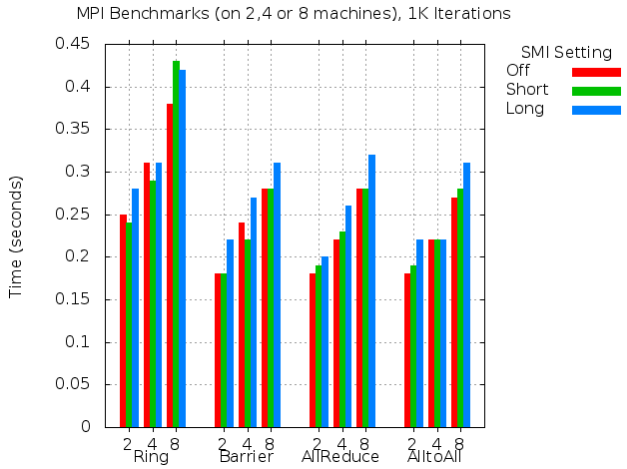

Fig. 10. Multi-machine communication, 1K iterations, SMIs enabled on smitest3 only

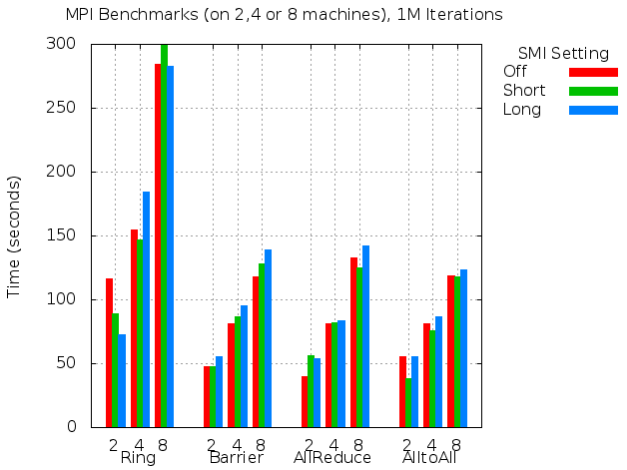

Fig. 11. Multi-machine communication, 1M iterations, SMIs enabled on smitest3 only

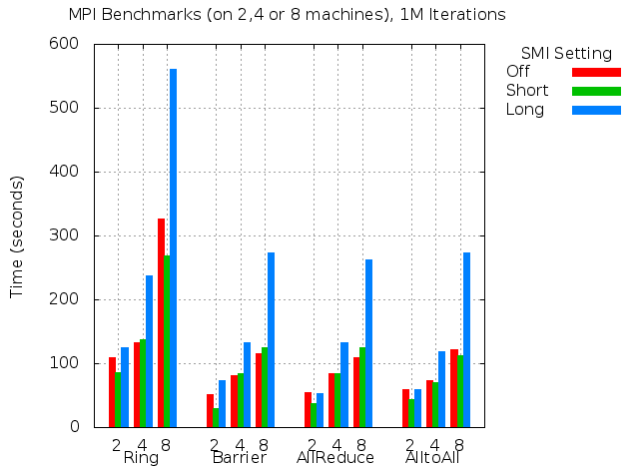

Fig. 12. Multi-machine communication, 1M iterations, SMIs enabled on all smitests

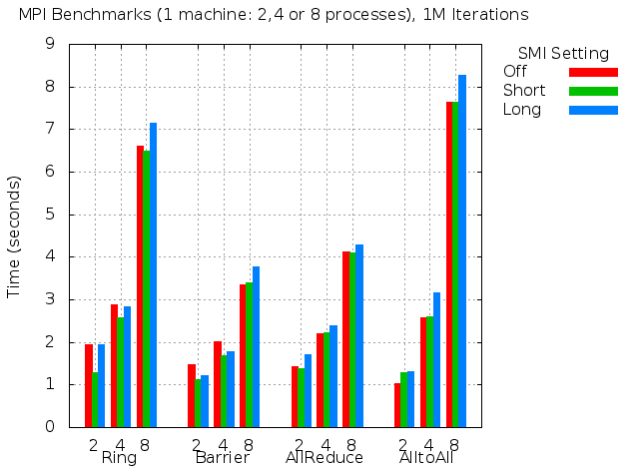

Fig. 13. 1 machine communication only, 1M iterations, SMIs enabled on smitest3

TABLE II
COMPARISON BETWEEN MACHINES WITH AND WITHOUT HYPERTHREADING ENABLED

All to All Benchmark		smitest3 (hyper-threading)				smitest5 (no hyper-threading)			
SMI Type	# Processes	Average Time (seconds)	Average SMI Count	% Remaining CPU clocks	% Normalized Time	Average Time (seconds)	Average SMI Count	% Remaining CPU clocks	% Normalized Time
Short	2	1.59	1	100.0	99.6	0.60	0	100.0	94.5
	4	2.93	3	100.0	100.3	3.65	3	100.0	101.3
	8	7.58	7	100.0	100.3	40.97	42	100.0	100.0
Long	2	1.63	1	93.3	97.2	0.58	0	100.0	97.1
	4	3.16	3	89.6	93.0	3.86	3	92.2	95.6
	8	8.20	7	90.6	92.6	54.15	51	90.5	75.7

enabling of hyperthreading to ascertain its effect in relation to SMI enabling. Possible synchronization of SMI timing may be an interesting test to determine this parameter upon MPI benchmarks possibly akin to studies in reducing noise in high performance computation *[4]*. Lastly, other benchmarks can be tested, such as NAS (NASA MPI) benchmarks or other prominent MPI benchmarks. Especially, the tests in this paper mainly tested the MPI routines, thus MPI communication was particularly tested. It may be interesting to incorporate more computation into the mix or other important features like memory use and disk reads.

G. Netperf

Background The effects of SMM on networking are of particular interest because SMIs may not only affect the local system but also all systems connected to it. Delgado and Karavanic investigated the effects of SMM on TCP throughput and found that long SMIs reduce the throughput at a scale directly related to the time spent in SMM while short SMIs may not have any noticeable effect on throughput at all [1]. We sought to confirm their results and to expand the investigation to UDP transmit.

To gain a better understanding on how adverse effects propagate between two hosts, we performed all tests twice, first with SMIs active on the server (sender) but not on the client (receiver), then with SMIs active only on the client side of the connection. In Figures 14, 15 and in Tables III, IV we use the following notation to refer to these two cases: *Sender(SMI)* $\Rightarrow$ *Receiver* and *Sender* $\Rightarrow$ *Receiver (SMI)*.

Methodology Netperf 2.6 [5] was used to measure the throughput between two cluster nodes. Each type of test was performed for varying levels of SMIs, and multiple long SMIs were issued back to back to simulate longer SMIs. Each test ran for 60 seconds and was repeated five times; all reported numbers are the average of these five trials.

Results and Discussion The effects of long SMIs on TCP throughput are shown in Figure 14. The leftmost bar in the chart indicates the remaining CPU time available to the system after subtracting the amount of time spent in SMM. The middle bar shows the throughput for the case where SMIs are enabled only on the server. The right bar shows the throughput for the case where SMIs are enabled only on the client. Our results confirm that the drop in throughput scales with the amount of time spent in SMM. The results also show that in cases where SMIs are enabled on the client but not on the server, the client experiences some additional degradation due to overhead in the form of TCP packet retransmissions. Table III illustrates this effect further.

Table III shows the amount of data that was transferred between two hosts in 60 seconds and the average number of TCP retransmissions, which is virtually zero in the case where the client was not interrupted by SMIs (top rows). This changes if the client is slowed down by SMIs (bottom rows). For example, with one long SMI per second, almost 50,000 retransmissions (800 / sec) occurred. Interestingly, this number decreases with increasing amount of time spent in SMM. For instance, with five long SMIs per second, only half the number of retransmissions occurred. The inherently lower transfer rate at that point explains this phenomenon.

The effects of long SMIs on UDP throughput are shown in Figure 15. Most importantly, we did not see a significant difference between TCP and UDP in terms of their sensitivity to SMM. Like TCP, the drop in throughput scales with the amount of time spent in SMM. Unlike TCP, both cases (SMIs enabled on the sender but not on the receiver, and vice versa) experienced the same level of degradation, which was to be expected since UDP is a stateless protocol. However, the reasons for the drop in throughput are different. If the server is slowed down by SMIs, then the drop in throughput is just a consequence of the server sending at a lower rate. If the client is slowed down by SMIs, then the drop in throughput is caused by the receiver being unable to process all incoming UDP datagrams. So while the throughput between the two cases is the same, the amount of data sent over the network is not. This is further detailed in Table IV, which shows the results for the 60 second UDP transmits in terms of the amount of data (in gigabytes) the server sent and the amount of data the client received.

To determine the effect of short SMIs, throughput was measured for the following five different scheduling intervals: 1, 10, 100, and 500 short SMIs per second. All the individual benchmarks experienced no throughput degradations, neither for TCP nor UDP. Hence, our results are identical to the findings reported by Delgado and Karavanic, who explain their results arguing that ”SMI usages that are able to interleave SMIs with I/O may be able to avoid the full penalty of the SMI.” [1] We concur.

Conclusion and Future Work Our results show that long SMIs can have a significant effect on network throughput. More specifically, the performance penalty is closely linked to the length of the amount of time spent in SMM. Systems at the receiving end of a data connection that are frequently interrupted by long SMIs may experience some additional degradation if they fail to keep up with the incoming data stream to an extent to which the server starts to retransmit data. Additional traffic ultimately affects all systems on the network. By contrast, short SMIs seem to have no significant effect on network throughput.

Netperf can also be used to measure CPU utilization and latency. All future work should include these two important aspects of network performance.

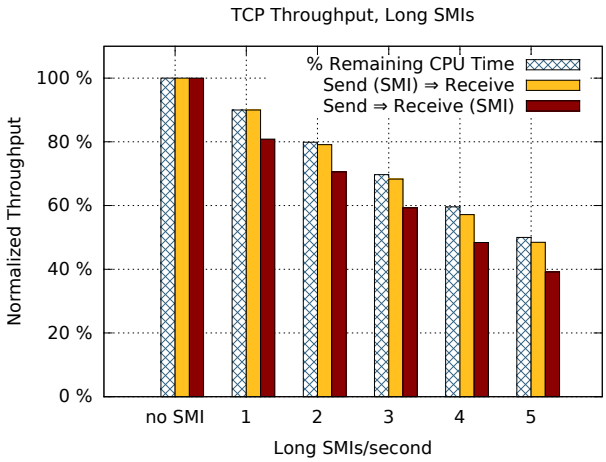

Fig. 14. TCP throughput scaled against a baseline for varying levels of SMIs; Maximum Std. Deviation 1.5%

TABLE III
60 SEC TCP DATA TRANSFER, LONG SMIS; MAX STD-DEV IS 102 MB

# SMIs / Second	GB Transferred via TCP Send (SMI) ⇒ Recv	# Retransmissions
0	5.60	0
1	5.05	0
2	4.43	0
3	3.82	3
4	3.21	2
5	2.71	7

# SMIs / Second	GB Transferred via TCP Send ⇒ Recv (SMI)	# Retransmissions
0	5.60	0
1	4.53	49865
2	3.95	45418
3	3.32	37518
4	2.71	28775
5	2.20	24722

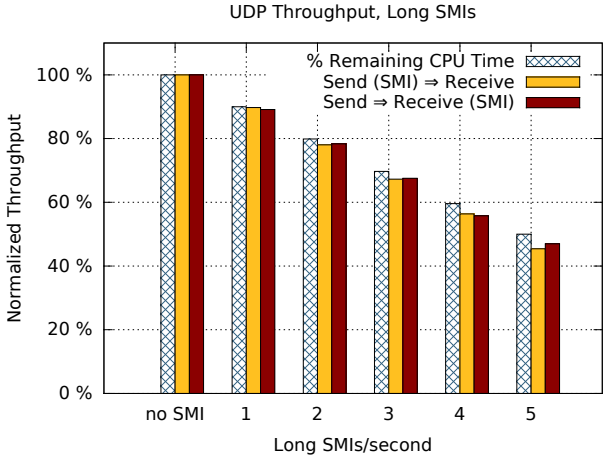

Fig. 15. UDP throughput scaled against a baseline for varying levels of SMIs; Maximum Std. Deviation 1.2%

TABLE IV
UDP DATA TRANSFER, LONG SMIS; GB SENT AND GB RECEIVED IN 60 SEC; MAX STD-DEV IS 102 MB

# SMIs / Second	GB Transferred via UDP Send (SMI) ⇒ Recv	GB Transferred via UDP Send ⇒ Recv (SMI)
0	5.71 ⇒ 5.71	5.71 ⇒ 5.71
1	5.13 ⇒ 5.13	5.71 ⇒ 5.09
2	4.46 ⇒ 4.46	5.71 ⇒ 4.48
3	3.84 ⇒ 3.84	5.71 ⇒ 3.86
4	3.22 ⇒ 3.22	5.71 ⇒ 3.19
5	2.60 ⇒ 2.60	5.71 ⇒ 2.68

H. LMBench

Background Lmbench is a tool for performance analysis for applications. Application tasks are divided into threads and then executed as individual units but these threads may or may not be dependent upon each other for input or data. For data passing, they need to communicate, and how much time they spend to establish connections and transfer data will directly affect performance. Lmbench is a simple suite of portable benchmarks which run a number of benchmarks such as ones which test bandwidth, latency, and system call overhead for different file sizes and memory sizes. Lmbench provides a suite of benchmarks that attempt to measure the most commonly found performance bottlenecks in a range of applications by measuring overhead in interprocess communication, data transfer rate and context switching [6].

Methodology Lmbench provides many bandwidth and latency benchmarks, therefore we will only analyze a few of them to determine the effect of SMIs on application performance. The LMbench tool runs applications by dividing them into several threads and then checks the time to establish connection between processes. In the User Data Protocol (UDP), no acknowledgement of data transfer occurs. Performance is measured by finding UDP latency. If UDP latency between two processes is prolonged upon enabling of SMIs, this indicates that the SMIs are increasing communication time, indicating that they are directly adding time for application execution. In contrast to UDP, the Transmission Control Protocol (TCP) provides acknowledgement for connection establishment between processes. A comparison is also made for TCP latency when SMIs are enabled with short and long durations and when they are not enabled. Additionally, pipe bandwidth, which is where output of one process is transferred to another process as input, is measured in Lmbench. Lastly, the time difference for creation, termination and execution of processes is measured in comparison to SMI enabling.

Results and Discussion UDP latency, TCP latency, and TCP and UDP latency for Remote Procedure Call (RPC) with short SMIs yield run times of almost same amount as runs with no SMIs enabled, showing very little difference. There is considerable difference between these latencies when duration of SMIs is increased (when long SMIs are enabled). We can see results in Graph **??** that there is quite a large difference even between short SMI and no SMI for times taken to establish TCP/IP connection.

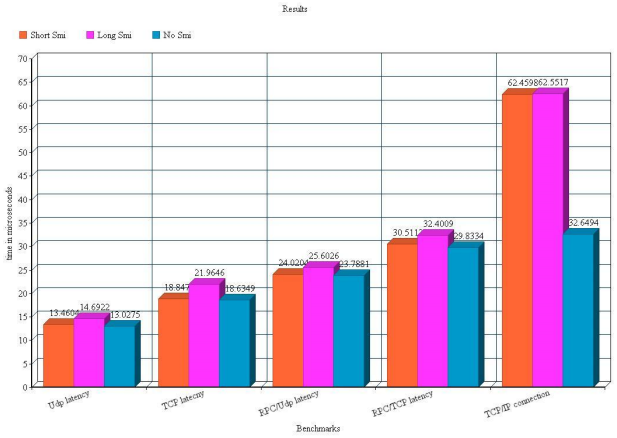

Fig. 16.

We also measured the rate of data transfer when output of one process acts as input to another process. We can see the results listed below. The bandwidth is maximum when SMIs are disabled. Only a 13 Mb/sec difference exists when short SMIs are enabled but a difference of 233 Mb/sec occurs when long SMIs are enabled.

TABLE V

Short SMI	Long SMI	No SMI
1970 Mb/sec	1750 Mb/sec	1983 Mb/sec

Below, the results for time in creating a process and terminating it are displayed. Also is shown the time for creating a process and executing it, as well as for creating a process and executing it in system shell. In these, little difference exists between times of runs with short SMIs and runs with no SMIs, but there is considerable difference when long SMIs are enabled.

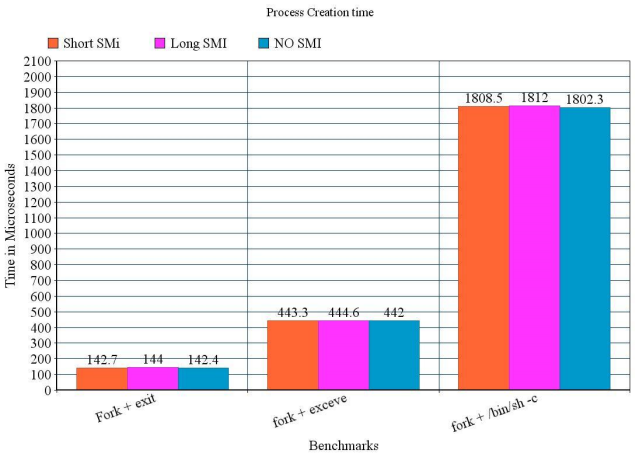

Fig. 17.

Conclusion and Future Work After running Lmbench tool on the provided machines, the results clearly indicate that SMIs are not a bottleneck to performance when their durations are short, but they are when the duration of SMIs is increased. There is very small difference in performance of applications when short SMIs are enabled and SMIs are disabled, so future work would be to find SMI duration where application performance is the exactly same as when SMIs are disabled. This would serve the purpose of protecting system from issues like power throttling without any performance overhead.

I. NBench

Background The Nbench benchmark measures the performance of the computation power of the CPU by calculating integer computation index, floating point index and memory access index (combined sequential and non sequential memory access). Its measurement can give an idea of how well a system will perform. It runs a set of C programs to calculate the performance measurement indices. Nbench uses the runtime statistics to calculate the indices. The set of programs are divided into two major sets: 1) Integer computation and memory access and 2) Floating point computations. The test suite consists of the following set of programs.

1. Integer Computation and Memory Access

- Numeric Sort - Sort an array of 32 bit integers
- String Sort - Sort an array of strings of arbitrary lengths
- Assignment - Task allocation algorithm
- Idea Encryption - Block cipher encryption algorithm
- Huffman - Text and Graphics compression algorithm

2. Floating-point Computation

- Fourier Coefficients - Series approximation of waveforms
- Lu Decomposition - Algorithm for solving linear equations
- Neural Net - Functional back-propagation network simulator

Methodology The Nbench test suite is based upon dynamic workload. It performs timing using a stopwatch paradigm. The method StartStopwatch() begins timing, and StopStopwatch() ends timing and reports the elapsed time in clock ticks. The benchmark has a global variable called GLOBALMINTICKS. This variable is the minimum number of clock ticks that the benchmark runs for a particular program. It only allows this minimum clock ticks to report to StopStopwatch(). For example, if one runs numeric sort, the program will construct an array filled with random numbers, call the StartStopwatch(), sort the array and then call StopStopwatch(). If the reported time in StopStopwatch() is less than GLOBALMINTICKS, the benchmark will build two arrays and will try again to sort them. The process is repeated until the StopStopwatch() routine reports the minimum time ticks, being GLOBALMINTICKS. This principle is applied to all the tests in the benchmark. [7]

Results and Discussions All the tests were ran for 10 trials upon the same machine (smitest2), and the average was obtained for these trials. During benchmark runs, it was verified that no other process ran in the system to avoid affecting the running time computation of the benchmark.

For both short and long SMI's the average iterations/sec were affected and reduced. For example, for numeric sort

the average iterations/sec were reduced from 1172 to 1123 and 1079, for short and long SMI runs, respectively. For long SMIs, similar behavior was observed for the rest of the programs for integer computation and memory access. However there were anomalous results for the short SMI case. In 4 out of 10 runs, the iterations/sec increased slightly for all the programs of this category. The following graph depicts this behavior for all the programs in integer computation and memory access.

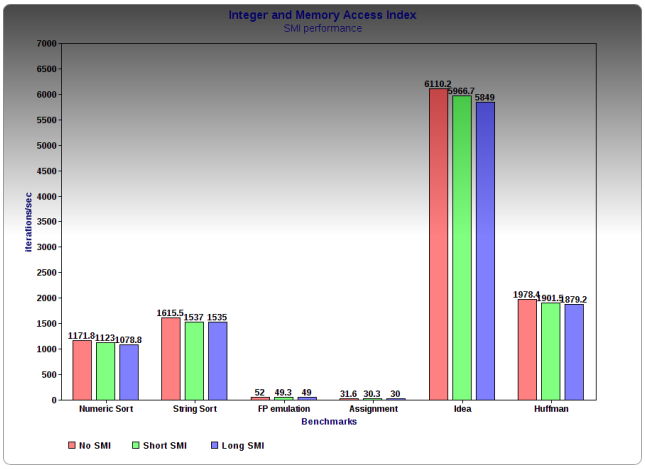

Fig. 18.

Similar behavior occurred for floating-point computation programs. Four out of 10 test runs showed performance improvement in short SMI runs, whereas long SMIs always degraded the computation capacity of the programs. The following graph depicts this behavior.

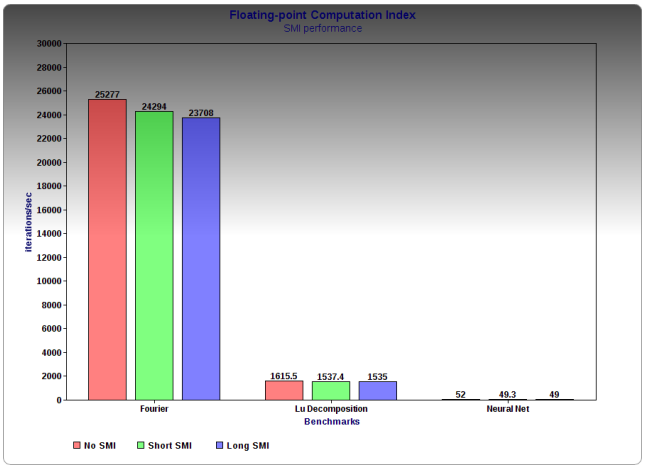

Fig. 19.

The main idea of the Nbench benchmark is to generate a set of indices which reflect the overall performance of the system. In this benchmark, we get an index score for integer computations, memory access and floating-point computations. Three set of index scores were calculated to measure the performance of the system in no SMI mode, short SMI mode and long SMI mode. The following graph shows that as an average of 10 runs the index score is less for both short SMI and long SMI. However the change in short SMI mode is comparatively less in short SMI than in long SMI.

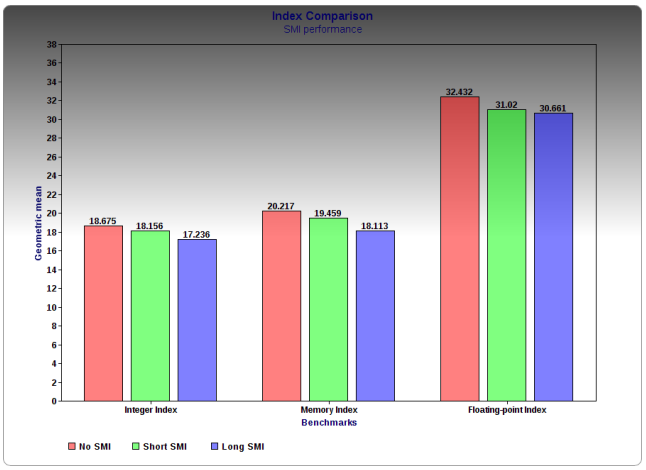

Fig. 20.

The percent change for these indices in given in the following table.

TABLE VI

Index	Short SMI	Long SMI
Integer Computation	2.7	7.7
Memory Access	3.7	10.4
Floating-point Computation	4.3	5.4

Conclusion and Future Work The results of the Nbench benchmark clearly indicate that SMIs interrupt the execution of the application computation. It affects the running time and iterations/sec for all of the test suite program. Long SMIs greatly affects the system performance and care should be taken to ensure that the system spends the least possible time in System Management Mode. Short SMIs do not significantly affect the running time of the application computation. For a noticeable number of cases the iterations/sec were increased slightly for all the programs of the test suite. The performance degradation was only visible when calculating the average of 10 runs. For the future work, especially to arrive at a definite conclusion for benchmark runs with short SMIs, a larger trial set size should be used. Investigation should be done to definitively determine the length of the SMIs, and their performance impact should be studied separately so that a clear definition for long SMI and short SMi can be established. Finally, these Nbench benchmarks should be run in a multiprocess environment where each process is doing some meaningful work. Performance impact of SMIs in such cases should be studied to understand its impact.

J. Power

Background An ever present performance concern, power continues to be an issue for developing technologies, including those revolving around the SMM abstraction layer. New tools such as HyperSentry [8] and Hypercheck [9] execute runtime

checks on hypervisors by way of SMM. Depending on the length frequency of these checks systems may encounter significant power drains to which underlying abstraction layers may be completely unaware of.

Methodology CPU power states were measured using Turbostat, a tool for gathering data on the time spent in various CPU power states. While running, Turbostat reports on an interval the percentage time spent in power states C_0 (Full power), C_1, C_3, and C_6 (deepest sleep state). A control report was first created by running Turbostat on an idle machine for 5 minutes. While never completely idle, Turbostat reported that the majority of time was spent in C_0. Next, Turbostat was run with a stream of short (0.05ms) SMIs occurring every second for 5 minutes. Lastly, Turbostat was run with a stream of long (100ms - 110ms) SMIs occurring every second for 5 minutes.

Results and Discussion While short SMIs generated a relatively negligible effect on power consumption, long SMIs produced significant reductions in percentage time spent in deep sleep states. While running long SMIs, Turbostat reported almost a 25% decrease in time spent in C_6, the deepest CPU sleep state. In addition, CPU states C_0 and C_3 both saw an increase of about 10% in percentage time spent in the states as illustrated in fig. bellow. While programs such as HyperSentry and HyperCheck only generate 35ms SMIs, clearly sustained SMI generation has the potential to cause significant reductions in power efficiency.

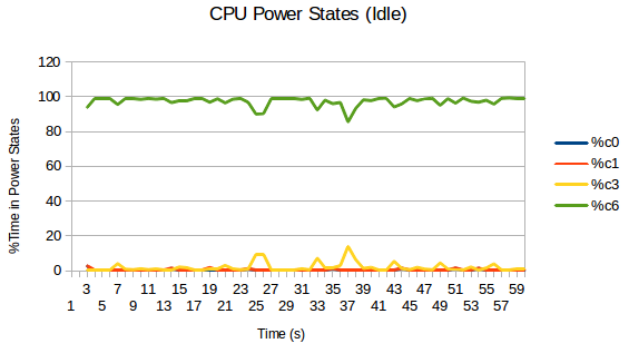

Fig. 21.

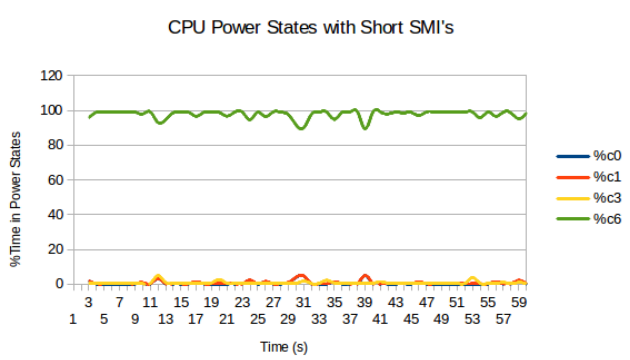

Fig. 22.

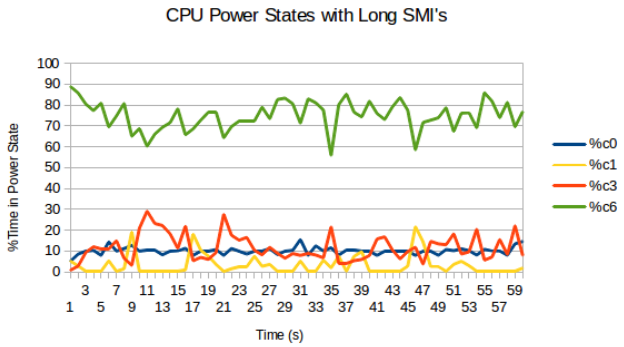

Fig. 23.

Conclusion Short SMIs do not seem to change power states from that produced when no SMIs are used. In contrast, when long SMIs are enabled, much more time is spent in higher powered states. This matches observation of Delgado and Karavanic.

III. General Conclusion

In this work, we investigated the effects of SMIs on system performance and power management. The used benchmarks, which included Linux kernel compilation, NBench, Netperf, LMBench, and custom built MPI benchmarks, allowed us to report on CPU bound computation, network throughput, network latency, memory access time, and MPI functions. We compared the effects of "long" (100 ms) and "short" (1 ms) SMIs against the baseline, and our results show that long duration SMIs may have a significant impact upon system performance. In fact, our findings reconfirm that performance degradation scales with the latency of the SMI. On the other hand, short SMIs seem to have no noticeable effect, particularly if they are interleaved with I/O. Long SMIs can be detrimental upon performance for multi-machine communication. Hyperthreading is an interesting parameter for measuring performance effects of SMIs. Short SMIs do not seem to change power states from that produced when no SMIs are used. In contrast, when long SMIs are enabled, much more time is spent in higher powered states. This matches observation of Delgado and Karavanic.

References

[1] K. L. Karavanic and B. Delgado, *Performance Implication of System Management Mode*. Portland, OR: IEEE, 2013.

[2] Wikipedia, *http://en.wikipedia.org/wiki/Message_Passing_Interface*. wiki, 2013.

[3] *http://www.mpich.org/static/docs/v3.1/www3/*.

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

[5] N. official website, *http://www.netperf.org*. web.

[6] L. McVoy, *lmbench: Portable tools for performance analysis*. Silicon Graphics, Inc , Carl Staelin Hewlett-Packard Laboratories.

[7] *http://www.tux.org/~mayer/linux/byte/bdoc.pdf*. TUX.com.

[8] P. N. A. Azab, *HyperSentry: enabling stealthy in-context measurement of hypervisor integrity*. CCS. Chicago, IL, 2010.

[9] a. A. J. Wang, A. Stavrou, *HyperCheck: a hardware-assisted integrity monitor*. Lect. Notes Comput. Sci. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6307LNCS: 158-177, 2010.