System Management Mode's Effect on Pthreads Performance

Chandhni Kannatinavida, Jim Miller, Michael Kirschbaum, Neil Babson
Portland State University

Abstract

Intel introduced System Management Mode (SMM) into its x86 processors to handle operating system independent functions such as Advanced Power Management (APM). All cores enter SMM simultaneously, and the mode cannot be interrupted by privileged software such as kernels and hypervisors. This makes an interrupt to SMM particularly disruptive to application performance, and their frequency and duration should therefore be limited; but very little research has been done on the performance effects of extensive SMM use. Proposals to implement Runtime Integrity Measurement Mechanisms (RIMM) security checks in SMM motivated Brian Delgado and Karen Karavanic to research SMM impact on the kernel as well as on a range of application. They demonstrated that unacceptable side effects were produced as the amount of time that the processor spent in SMM was increased. Our work expands upon these results, focusing particularly on how SMM impacts parallelism using Pthreads library functions. We measured this by writing a group of benchmarks and timing their execution while interrupts to SMM of varying frequency and duration were generated. We show that increasing the amount of time spent in SMM caused a corresponding slowdown for all Pthreads tasks, and that for large numbers of threads this slowdown was up to double that experienced by unthreaded processes.

1. Introduction

System Management Mode (SMM) is an x86 processor mode which was introduced in 1990. It was designed to perform platform specific operations, such as power management [3]. Before entering SMM the processor suspends all normal execution, including that of privileged software such as operating systems and hypervisors [1]. The processor state is saved to a protected memory region reserved for SMM called SMRAM [3]. Entry into the mode is triggered through a System Management Interrupt (SMI) which is the highest level of interrupt, and can occur in any process mode besides the SMM itself. An SMI causes all cores to enter SMM until the SMI handler finishes its work and control is returned.

Since the SMI handler cannot be interrupted, and preempts all processor cores, it can have a disruptive effect on host software performance, especially for real-time applications. Originally SMM was intended to used be for infrequent hardware tasks such as implementing Advanced Power Management (APM) [5], and it was therefore believed that any disruptive side effects would be insignificant as long as individual SMI latencies were kept below an

acceptable limit, which Intel defined as 150 microseconds [1]. The number and duration of SMIs increased when BIOS developers started to implement other applications, such as USB device handlers [5] in SMM. Zou et. al. found that this type of SMI could have a latency of over a millisecond [4], which they deemed unacceptable when running time sensitive programs.

Another expansion in the use of SMIs came in the form of a number of proposals to use SMM to increase security in virtualized environments, either by implementing security checks known as Runtime Integrity Measurement Mechanisms (RIMMS) [1][7][8][9] or by providing hardware support for the isolation of sensitive workloads [6]. For these applications, which monitor the integrity of the hypervisor and enforce isolation between virtual machines, the protection afforded by operating in SMRAM is very tempting. Concern about the performance ramifications of these security implementations, which entail SMIs with latency far in excess of the proposed limits [1], led Delgado et al. to develop a methodology for measuring the impact of SMM use at the system and application levels [1].

Our work builds on that of Delgado et al. to focus on the effects of SMM on parallel Pthreads applications. We study the impact that varying the lengths and frequencies of SMI has on the total time taken for thread create, mutex lock and unlock, conditional signals, and yield benchmark tasks. The rest of this paper is structured as follows. Section two reviews the approach and results provided by Delgado et al. In section three we present our methodology, briefly describing the suite of benchmarks we used, and the SMI driver we adapted from Delgado et al., which was used to generate interrupts. We also wrote a python script to carry out the measurements. Section four contains our experimental results. In section five we provide some analysis of these results, and the final section consists of a summary of our work and concluding remarks.

2. Related Work

Very little data exists on the how the use of SMM affects performance. The growing number of proposed security applications using SMM to protect their base code prompted Delgado and Karavanic to address this deficit in their paper “Performance Implications of System Management Mode” [1]. The authors provide a measurement methodology and developed three different techniques for SMI generation and measuring the time spent in SMM. Investigating both system and application level performance, they were able to show that increasing the amount of time spent in SMM resulted in “warnings, perceptible degradations for latency sensitive applications, throughput impacts, delays, and inaccurate time accounting at the hypervisor, kernel, and applications levels [1].” They concluded that the duration of the SMIs required for the implementations of RIMM in SMM that they surveyed would result in unacceptable performance side effects.

The present work is intended to augment that of Delgado and Karavanic, focusing on how the use of SMM affects the performance of Pthreads parallel applications.

3. Methodology

Delgado et al. were kind enough to provide us with an SMI driver they had developed, which generated short and long SMI's by writing a value to port 0xb2. It could be used to generate single long or short SMIs, or to turn on regularly scheduled interrupts. We modified the driver, adding the ability to specify the number of SMI calls (up to a maximum, for safety), and the length of the interval between calls in the case of repeating scheduled interrupts (down to a minimum). The default for each value is that of the original driver. We also added more consistent and informative print messages from the driver, for ease of use. Throughout testing we periodically checked the processor temperature, since the processor was unable to call on hardware support for temperature control while in SMM. We found that the processor temperature did increase by more than 10 degrees Fahrenheit when it spent a large proportion of its time in SMM, but this was still safely below a dangerous level.

In order to isolate the effect that SMIs have on the performance of individual Pthreads functions we created a suite of simple benchmark programs, following the approach described by Supinski [2]. Each test was designed to intensively exercise one Pthreads library function. The benchmark programs are as follows:

create.c: Usage: ./create [<numThreads>] [<iterations>]
This test makes numThreads calls to pthread_create(). The master thread creates a worker thread which starts in a recursive function that increments a shared counter, and creates another thread. This repeats until numThreads threads have been created. An implicit call to pthread_exit() occurs when each thread leaves the worker function. The master thread waits on a condition variable until all creations have occurred. The second command line argument specifies the number of times the test should repeat, with default equal to one.

yield.c: Usage: ./yield [<numThreads>] [<iterations>]
This test measures the speed of context switches between threads by having each thread repeatedly call sched_yield(). The threads are all pinned to the same core to force them to strictly alternate.

lock.c: Usage: ./lock [<iterations>]
In this test two threads repeatedly lock and unlock a set of four mutex locks in a prescribed order. The master thread starts out holding two of the locks and waits on a condition variable for the worker thread to initialize its two locks. The threads then 'ping-pong' the locks back and forth, with each thread locking and unlocking each lock once in each iteration of the test.

lock2.c: Usage: ./lock2 [<numThreads>] [<iterations>]

This performs a lock contention test, in which multiple threads compete to access a single lock, resulting in heavy usage of the mutex's wait queue.

condition.c: Usage: ./condition [<iterations>]
This is a pthread_cond_wait() and pthread_cond_signal() test. A worker thread repeatedly waits on a condition variable, while the master thread repeatedly signals it to wake up. The worker increments an iterations variable before waiting again.

condition2.c: Usage: ./condition2 [<numThreads>] [<iterations>]
numthreads threads are generated, and they all repeatedly wait and signal on a set of five condition variables.

We also wrote a python script, 'Run_experiment.py', which was used to conduct the experiments by using the SMI driver to generate different frequencies and durations of interrupts, and collecting timing data for the benchmark programs. Each recorded data point is an average of five benchmark program runs.

4. Results

All tests were conducted on a Dell PowerEdge R410 server, which is a node of a high performance research cluster at Portland State University. The nodes run bare metal Fedora.

We measured the total time taken for Pthreads benchmark programs to finish execution for varying SMI intervals. The values are calculated from SMI driver option 7, which produces a count of the number of cpu cycles that have elapsed. All the data values are an average of 5 runs. The data is for a particular jiffy-frequency value at which a long (or short) SMI is produced. A jiffy is an instance of the scheduler interrupting the system to do scheduling. In Linux this happens 1000 times per second, so 1000 jiffies per second. The values of SMI intervals go from 50 to 950. So 250 means that a long SMI was produced every 250 jiffies (1/4 of a second) and 950 means that a long (or short) SMI was scheduled only about once a second, and so we would expect less delay due to interference in that case.

One experiment was run with identical parameters except for the introduction of the "short SMI" as a replacement for the "long SMI". We also ran an experiment to test for the ratio of time taken by the benchmark programs to run alone to the time taken for the programs to execute with varying SMI intervals (both long and short SMI's) minus the average time spent in the SMI mode i.e.

$$\text{Degradation for a run} = \frac{\text{Baseline Pthreads Time}}{\text{SMI interfered time} - (\text{SMI length} * \#\text{SMIs})}$$

In the following sections an SMI refers to a long SMI and short SMIs are specified explicitly.

4.1 Effects on Thread Yielding Parameter

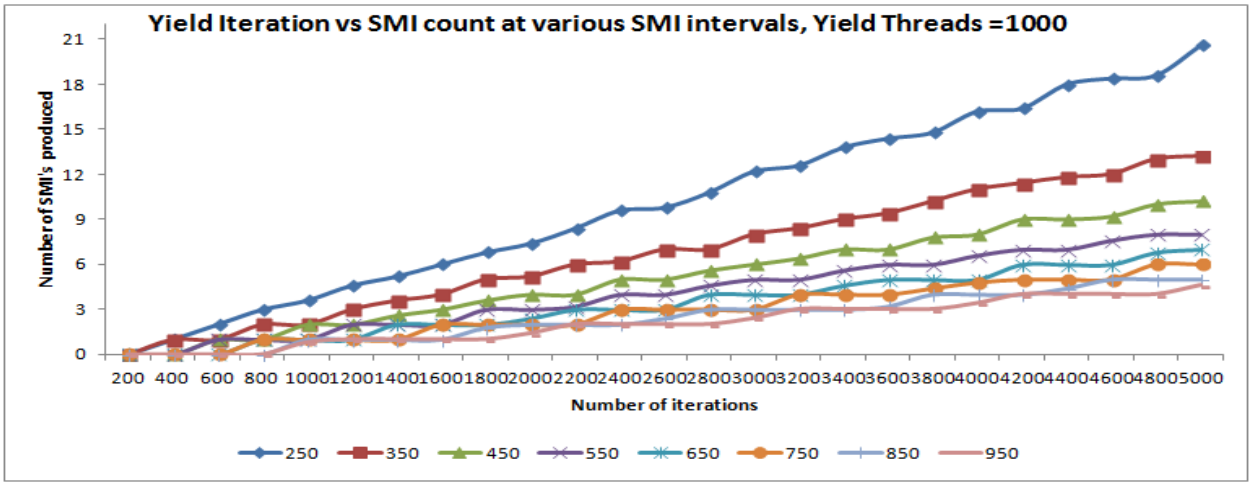

Figure 1

Figure 1 shows the average SMI count i.e. how many SMI's were generated on the runs when the time taken was calculated.

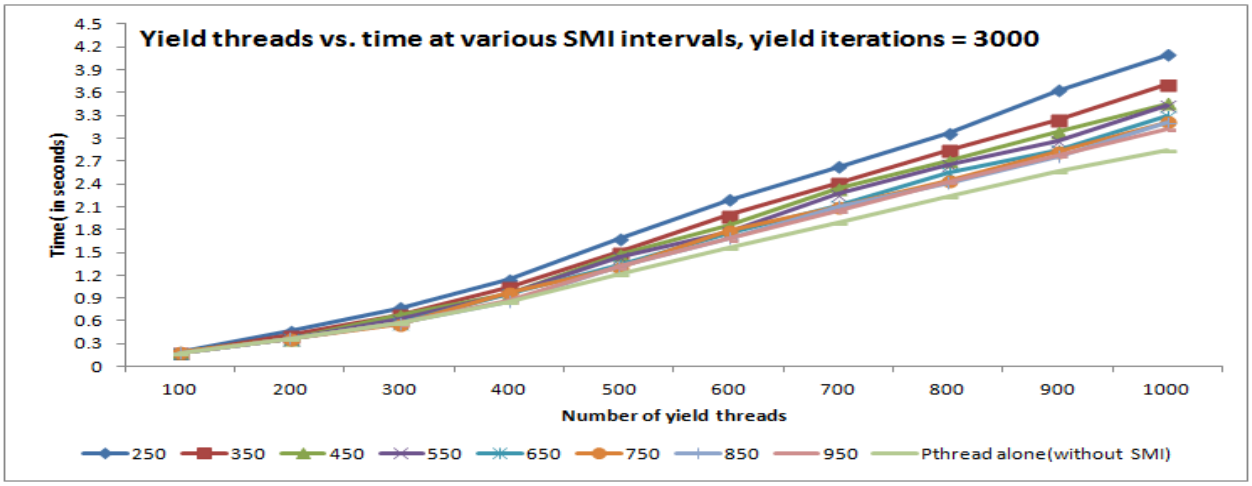

Figure 2

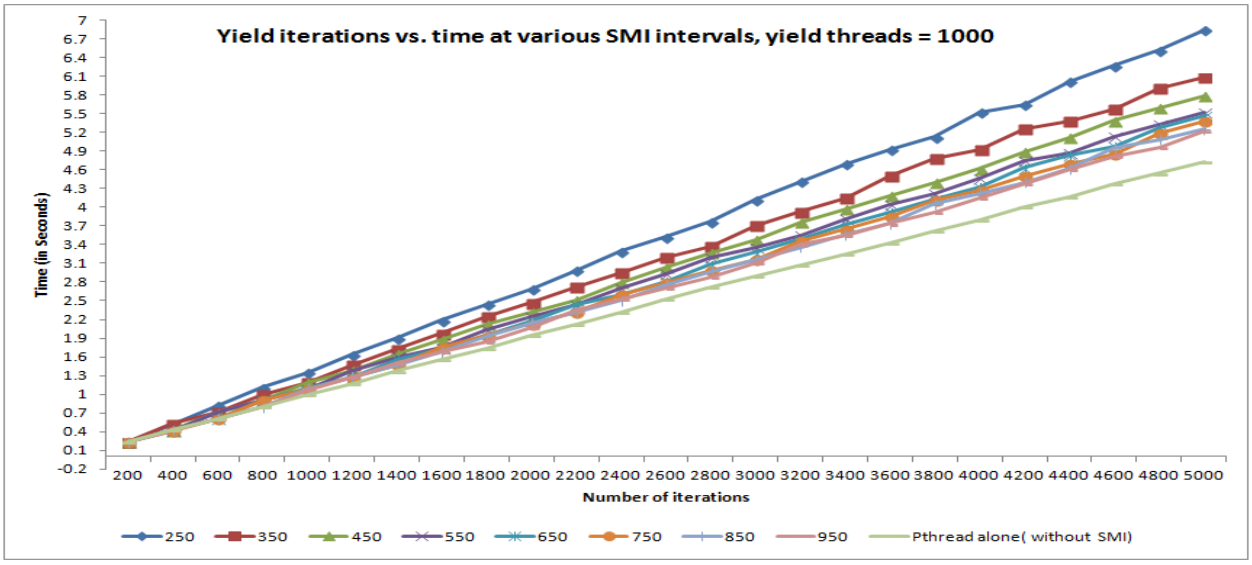

Figure 3

Results for evaluation of thread yielding are shown in Figure 2 and Figure 3. Figure 2 is plotted for varying SMI interval lengths, with the vertical axis referring to the time in seconds for completion of the C program (it is the average time from 5 runs) while the horizontal axis refers to increasing numbers of simultaneous threads that the yield program used (one of its parameters). The number of yield iterations is held constant at 3000. The graph also contains the case when no SMI was triggered; in this case SMI frequency is given as Pthreads alone (without SMI). The results show that the curve is more pronounced when SMIs at shorter intervals are generated. Figure 3 is plotted for varying SMI interval lengths, with the vertical axis referring to time in seconds and horizontal axis refer to increasing number of iterations. The number of yield threads held constant at 1000.

4.2 Effects on Lock-Unlock Parameters

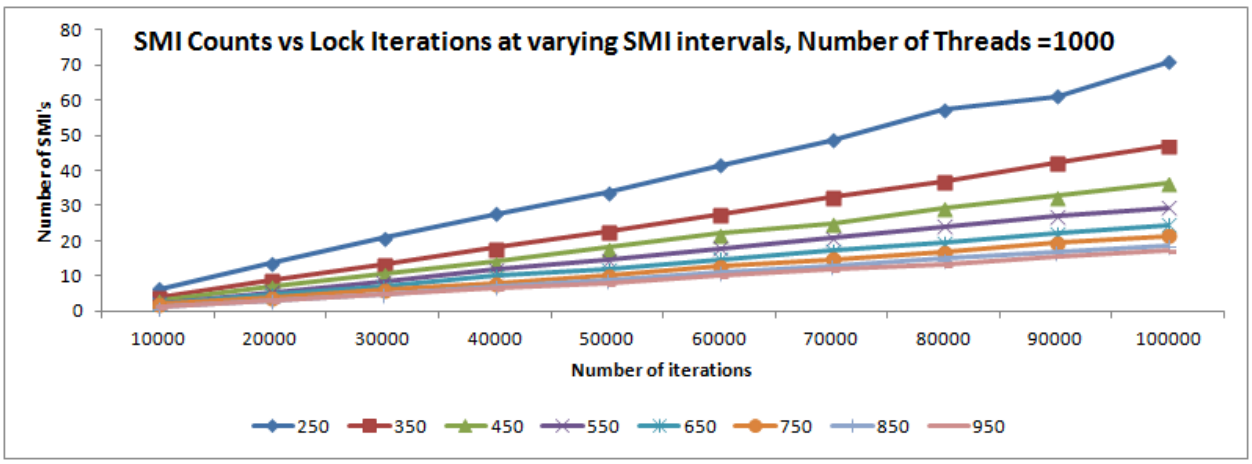

Figure 4

Figure 4 shows the average SMI count i.e. how many SMI's were generated on the runs when the time taken was calculated for the lock-unlock parameters of Pthreads.

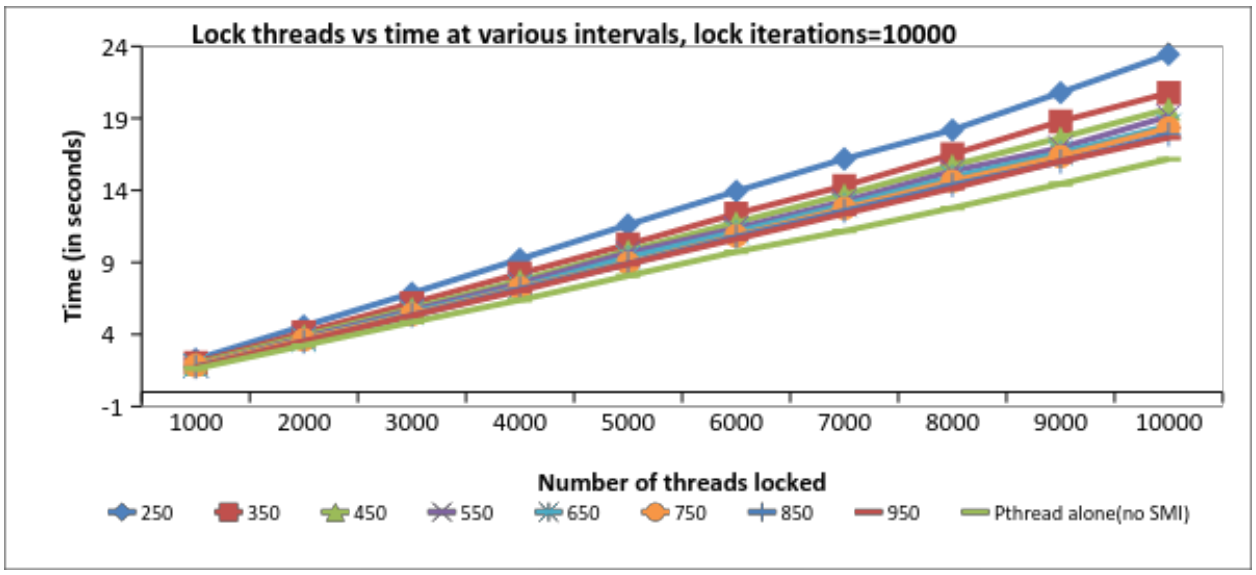

Figure 5

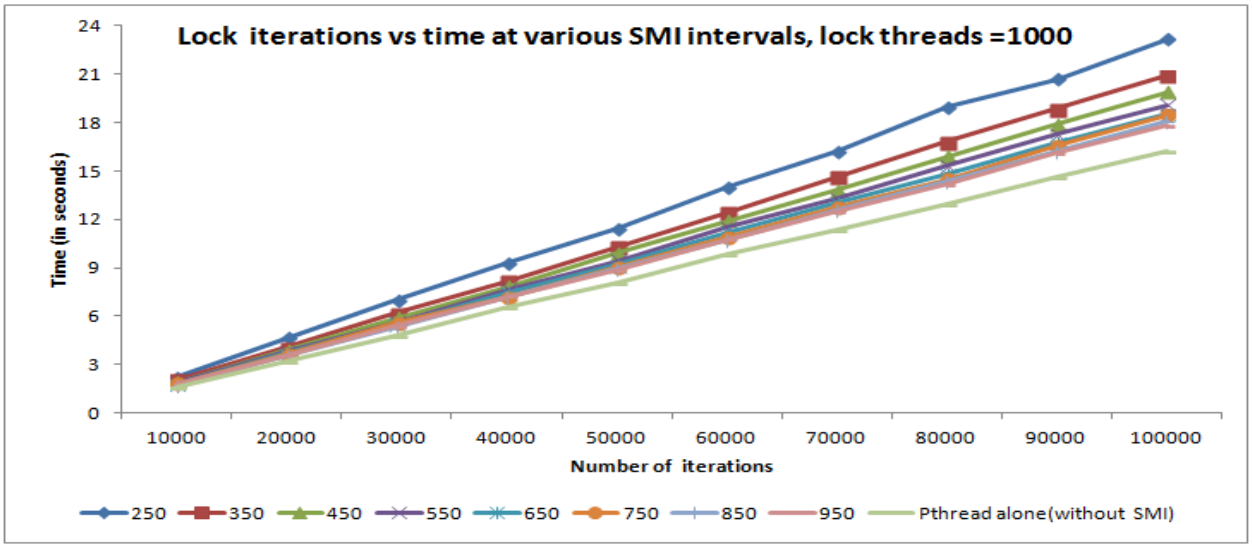

Figure 6

Results for evaluation of mutex locking and unlocking is shown in Figure 5 and Figure 6. Figure 5 is plotted for varying SMI interval lengths, with the horizontal axis referring to the number of threads for lock and the vertical axis to the time in seconds for completion of the C program, again using the average time of five runs. The number of thread iterations in this case is 10000.

Figure 6 is plotted for varying SMI interval lengths, with the horizontal axis referring to the number of thread iterations for locking and the vertical axis referring to the time in seconds

(average of five runs).The number of threads in this case is 1000. The curve is more pronounced when SMIs at shorter intervals are generated; thus shorter-interval SMIs have more effects on the Pthreads application.

4.3 Effects on Condition signals

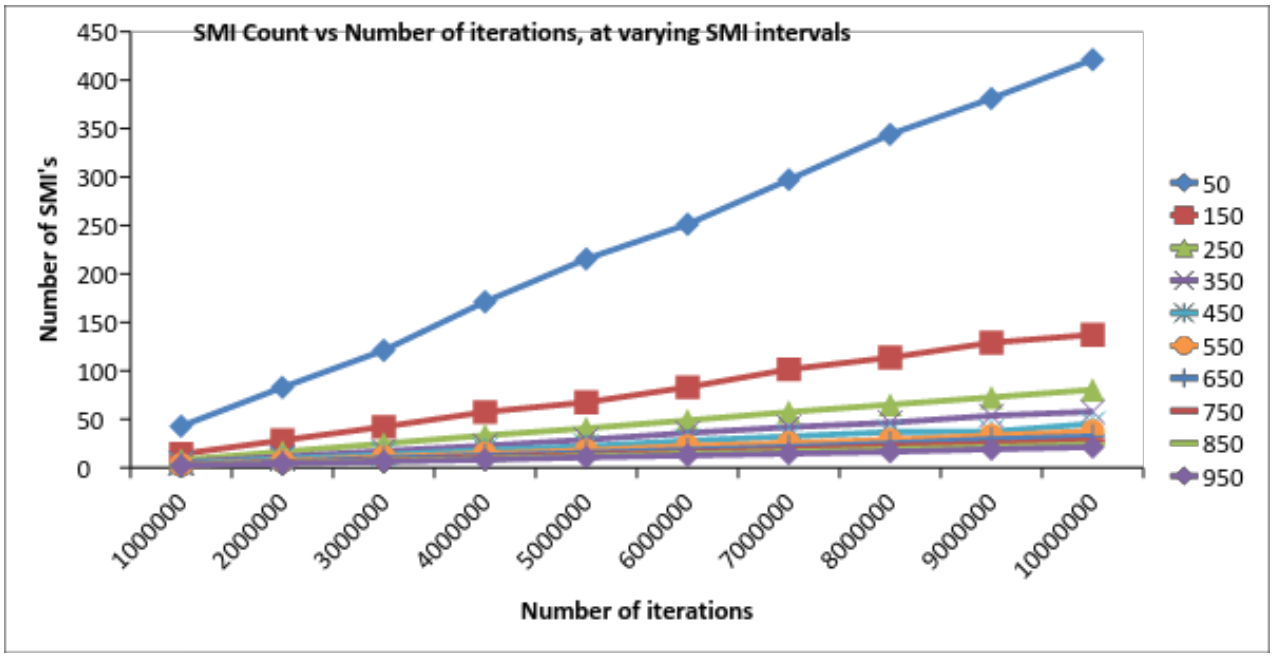

Figure 7

Figure 7 shows the average SMI count, i.e. how many SMI's were generated on the runs when the time taken was calculated for the condition signals parameters of Pthreads.

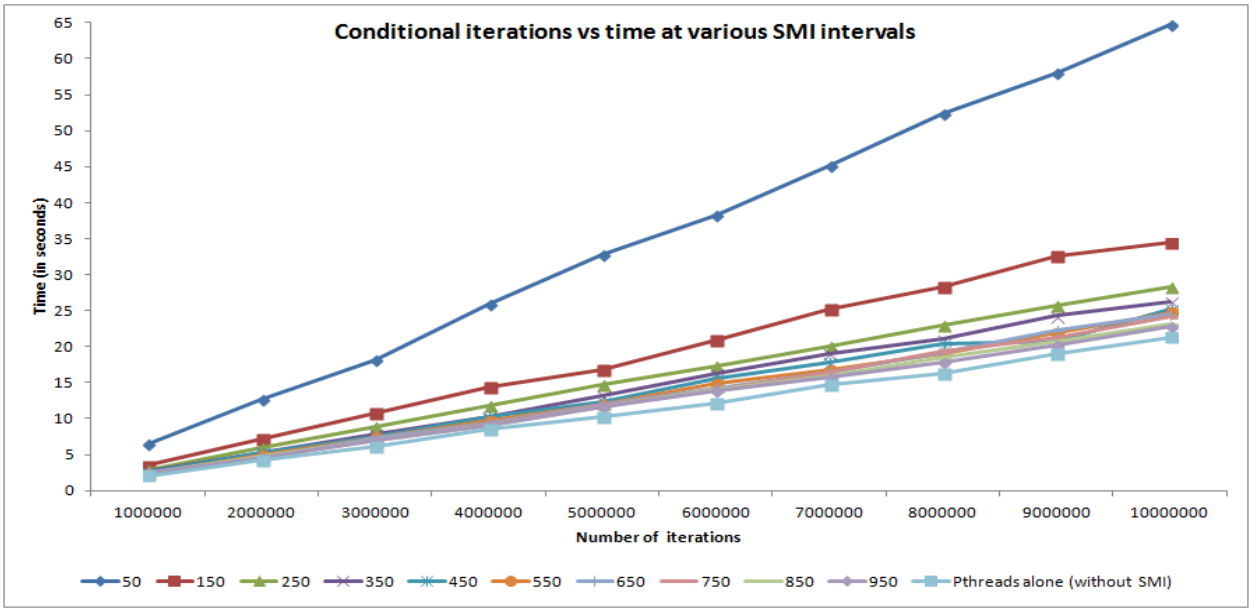

Figure 8

Figure 8 is plotted for varying SMI interval lengths with the vertical axis referring to the time in seconds for completion of the C program (it is the average time from 5 runs) and the horizontal axis refer to number of thread iterations for conditional signals. The results show that curve is more pronounced when shorter-interval SMI's are generated. The graph shows a sharp increase in the curve for shorter SMI intervals than for longer SMI intervals.

4.4 Effect on Thread create parameter

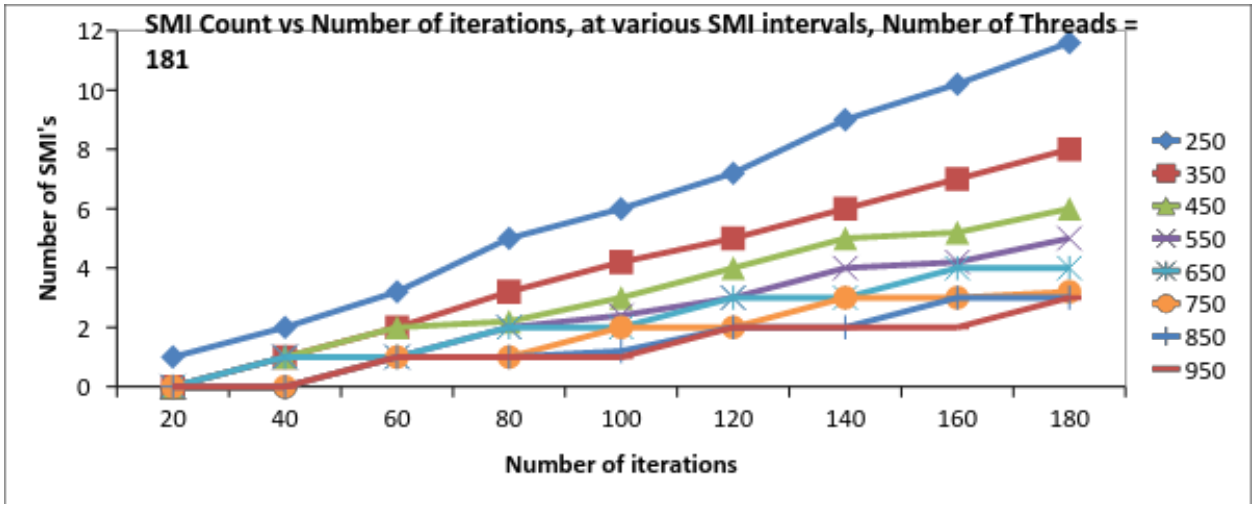

Figure 9

Figure 9 shows the average SMI count i.e. how many SMI's were generated on the runs when the time taken was calculated for the thread create parameters of Pthreads.

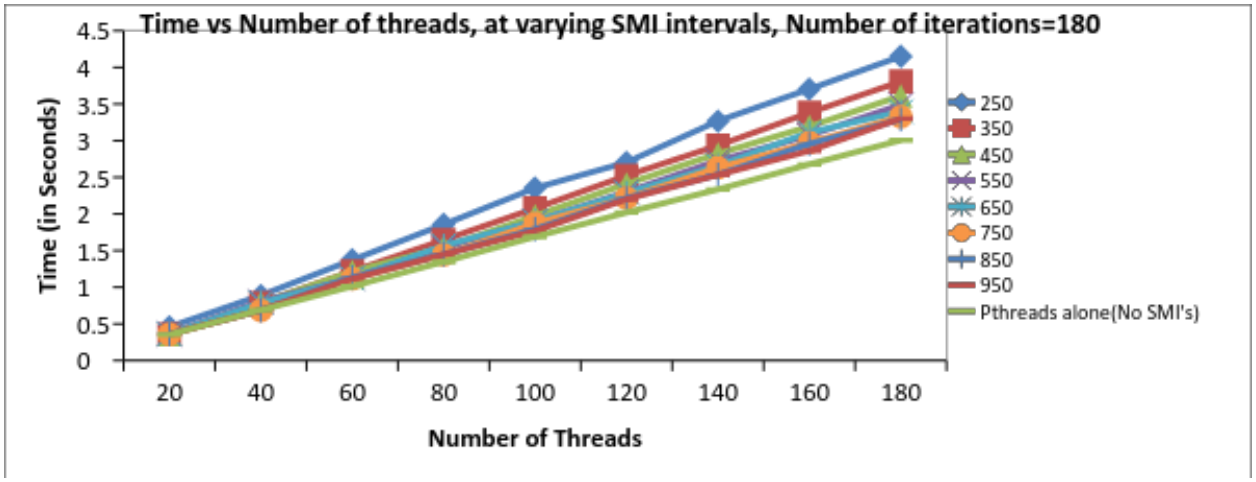

Figure 10

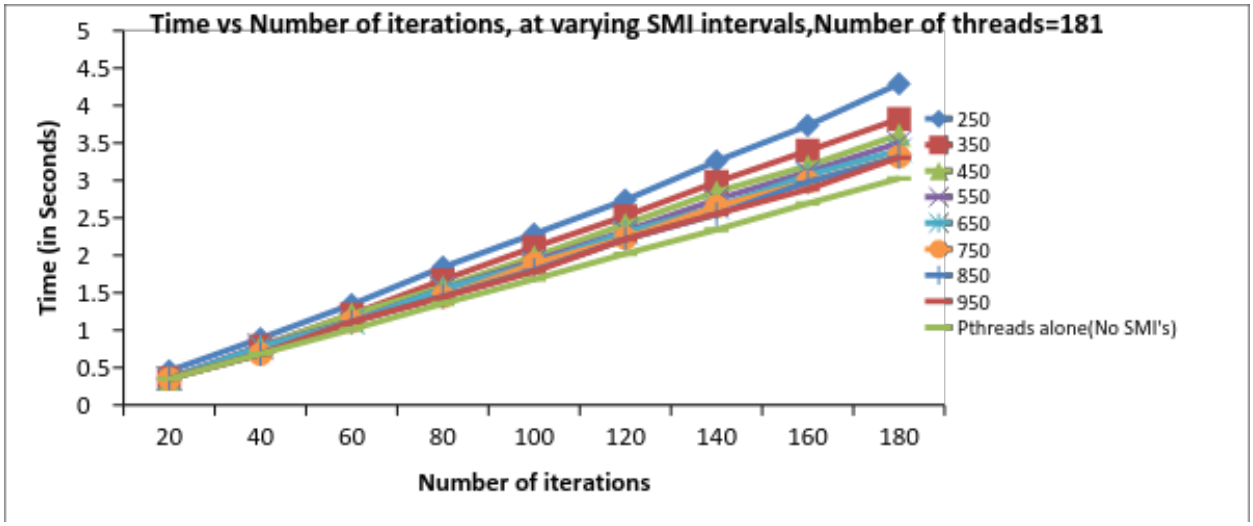

Figure 11

Results for evaluation of thread create is shown in Figure 10 and Figure 11. Figure 10 is plotted for varying SMI interval lengths, with the horizontal axis referring to the number of threads created and the vertical axis to the time in seconds for completion of the C program, again taken as average time in five runs. The number of thread iterations in this case is 180.

Figure 11 is plotted for varying SMI interval lengths, with the horizontal axis referring to the number of iterations and the vertical axis referring to the time in seconds (average of five runs). The number of threads in this case is 181. The curve is more pronounced when SMIs at shorter intervals are generated.

4.5 Comparing long and short SMIs

One experiment was run twice with identical parameters, except for the introduction of the “short SMI” as a replacement for the “long SMI” used in previous experiments. This option is a feature of the Blackbox SMI driver provided by Delgado et al. [1], and further modified by our team for additional flexibility and consistency of reporting information. 5000 instances of the long SMI were generated and each measured by comparing the system TSC values before and after, taking the difference and dividing by the number of CPU ticks per second, to obtain an average long SMI length in seconds. The same was done for the short SMI. It was found that an average long SMI is .1022 seconds, while an average short SMI is much shorter, .00005552 seconds in length.

Therefore, a long SMI on our system lasts about 1/10 sec and a short SMI lasts about 1/20,000 sec. The Linux kernel’s scheduler has a 1/1000 sec interval. This means that a long SMI is considerably longer than a scheduling interval, while a short SMI is considerably shorter, setting up the possibility that empirical measurements of SMI interference with Pthreads may look different in the two cases.

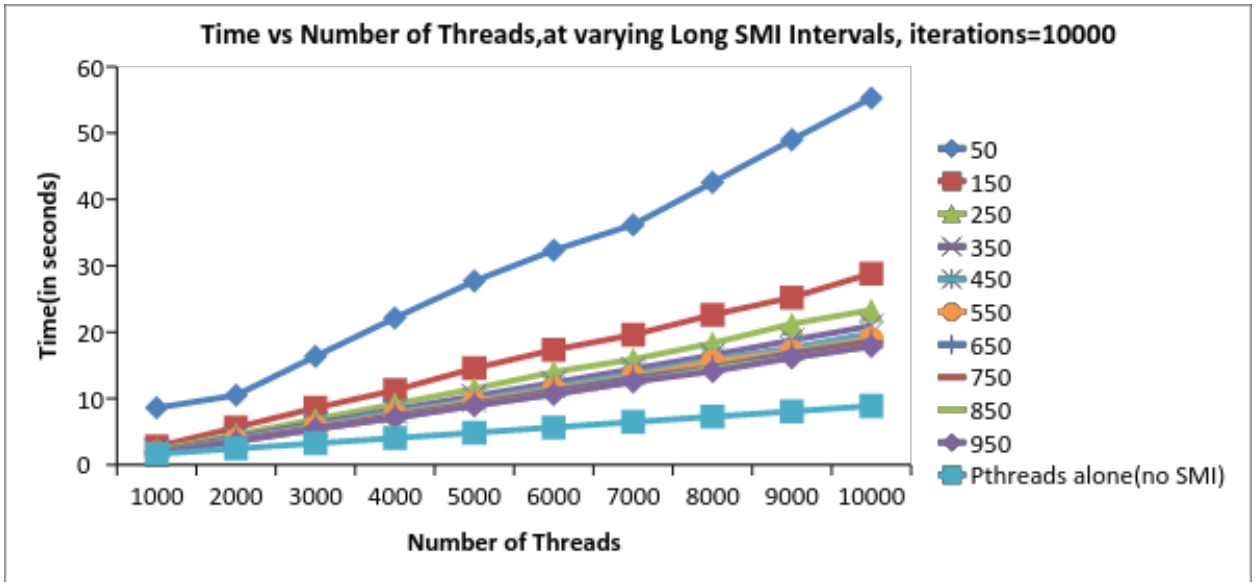

Figure 12

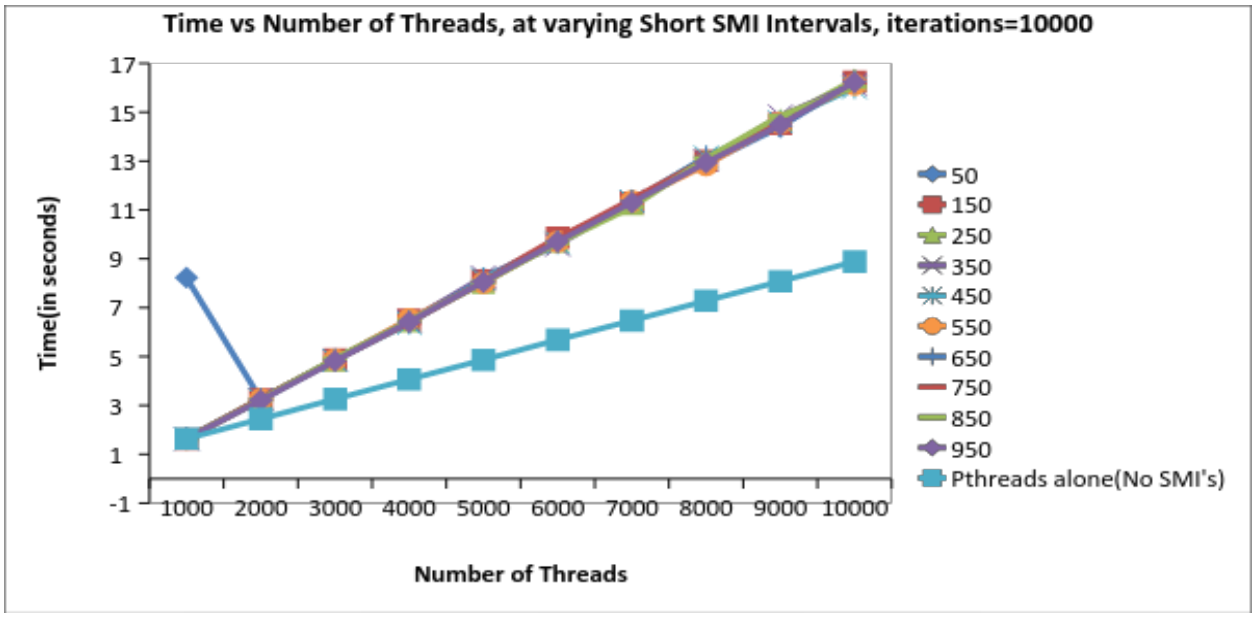

Figure 13

This was indeed found to be the case, but only to a limited extent. As seen in the two figures above, total completion times for both long and short SMIs generally trend steadily upwards, as the number of simultaneous threads is increased in the lock2 experiment. In the lock2 Pthreads program, threads contend to lock and unlock the same mutex, for a parameterized number of iterations. In our case, a fixed 10,000 iterations were used and the number of threads was allowed to vary from 1000 to 10,000, while the SMI interval was allowed to vary from 50 jiffies to 950 jiffies.

4.6 Ratio of Pthreads-only time to non-SMI-sub time for Short SMI's

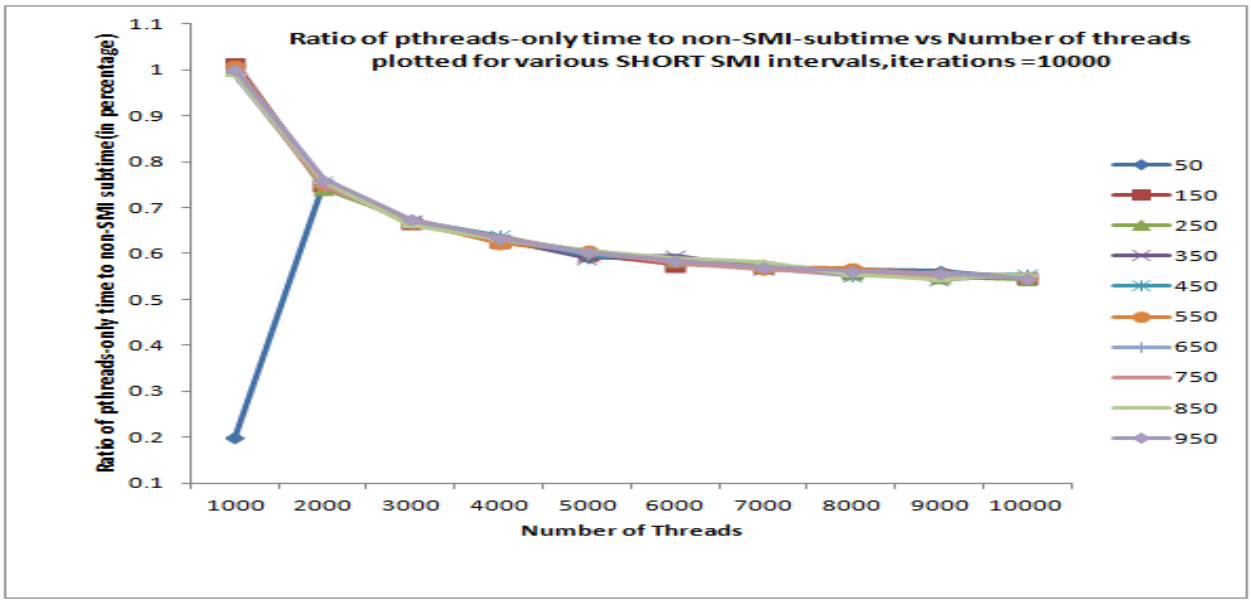

Figure 14

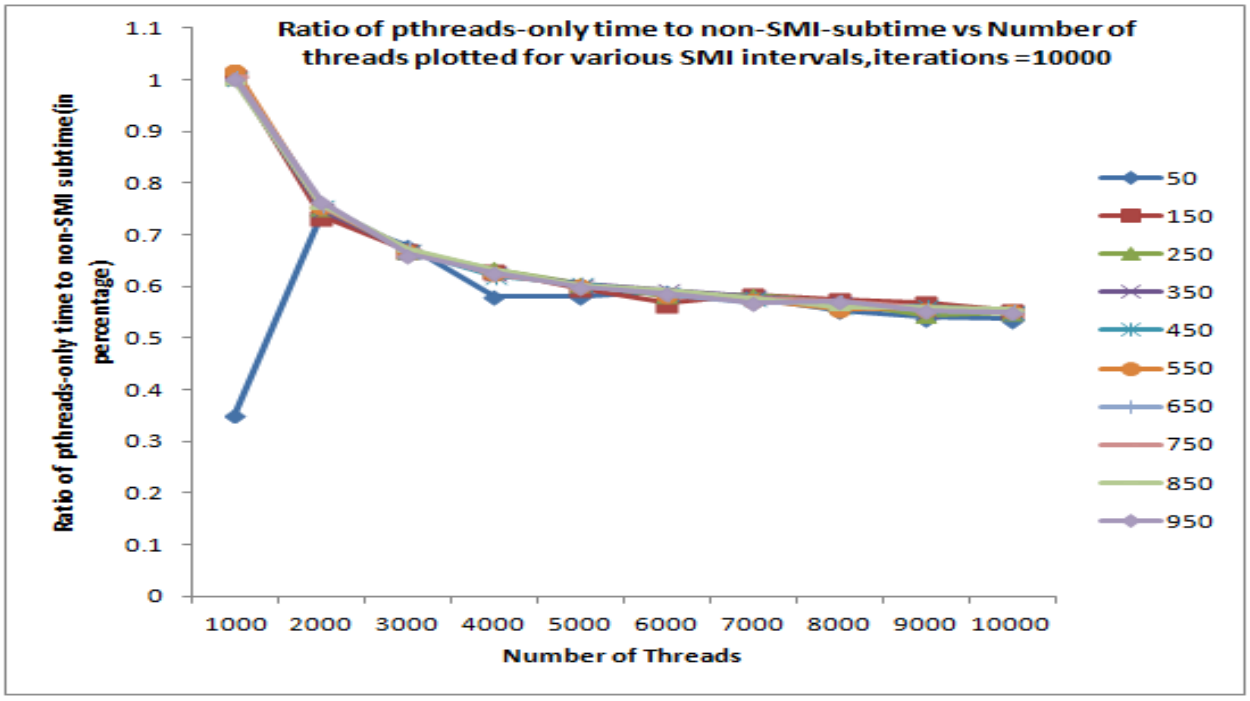

Figure 15

The same experimental runs in the lock2 experiment were also used to calculate a value of interest to us: the ratio of run time required for a baseline data point with no SMI

interference, to the run time required for the same data point interfered by SMIs, after the duration of the SMI itself is removed. In other words, we tried to precisely and quantitatively ascertain whether an SMI-interfered run required more time to complete the portion of the run used for Pthreads program work. This is where we tried to capture the "pure" degradation in system performance, expressing it as a ratio using the formula:

$$\text{Degradation for a run} = \frac{\text{Baseline Pthreads Time}}{\text{SMI interfered time} - (\text{SMI length} * \#\text{SMIs})}$$

The results were very interesting here. As the two graphs above show, this ratio, at almost every parameter for number of threads and SMI interval length, is similar in both the long and short SMI cases. Thus, SMI length seems to have little relevance to the amount of pure system overhead. Another striking result is that the 1000-thread data runs, at all SMI intervals except for 50 jiffies, exhibit essentially no overhead – the ratio is about 1.0. But just as uniformly, the ratio steadily declines in a slope as the number of threads is increased, eventually leveling off at about .55, again for all SMI intervals. This means that at 10,000 simultaneous threads, the influence of the SMIs caused about a doubling of the time required for Pthreads program work.

5. Analysis

As we have seen from the Results section, experimental runs interrupted by long SMIs are considerably lengthened, especially at short SMI intervals, and the series all differentiate themselves. Here, the 50-jiffy series trends dramatically upwards, likely due to the fact that a long SMI itself is 1/10 sec in length, leaving little room for normal system workload completion between such frequent SMIs.

The analogous figure for short SMIs reveals another interesting trend: all of the run lengths track upward at almost exactly the same values, except for the bottom series – a baseline series with no SMI interference - which tracks upward less steeply. The probable explanation is that, as number of threads increases, their combined influence on the system amounts to a strong signal which is not easily perturbed by SMIs which are too short in length, while long SMIs do perturb this signal because of their significant length.

Since this experiment involved only the Pthreads mutex mechanism, and since Linux "fast user space mutexes" rely on system calls rather than on the regular scheduler tick, we do not believe that lengthening of run completion times is induced solely by SMI resonance with the timer interrupt, as found in the Delgado et al. discussion of scheduler jitter [1]. Rather, the lengthening is induced by the completion time of the SMIs themselves, supplemented by some other kind of disturbance to the regularity of the OS caused by the SMIs. Certainly, any disruption to the scheduler tick will degrade system performance in many kinds of far-reaching ways. But timely handling of system calls is the central issue with our highly contentious mutexes.

The experiments run to compare pthreads-only time to non-SMI-subtime shows that SMIs are having some kind of impact on the OS's ability to handle system call interrupts for mutexes. The accuracy of the calculation of the average length of SMI can affect the results.

6. Conclusion

Various proposals have been made to repurpose System Management Mode to implement security checks and other functionalities. Due to the disruptive nature of System Management Interrupts these approaches may introduce unacceptable side effects, as shown by Delgado et al. [1]. The contribution of this paper is to add to the available data on the performance effects of using SMM, specifically on Pthreads library functions. Here we present some of our results on effects of SMM on parallel Pthreads applications, specifically for the parameters thread create, yield, mutex lock-unlock and conditional signals.

The results are as expected from the studies conducted by Delgado et al. [1]. SMM has a delaying effect on Pthreads applications, for the time taken to create threads, lock and unlock mutexes, etc. The effects are more pronounced for SMIs generated at shorter intervals. The results lead us to conclude that SMIs are have an impact on the OS's ability to handle system call interrupts for mutexes .The graphs shows that the for the SMI interval between 350 and 950 the time taken does not vary much. We do not know the specific reason for this but this presents scope for future works.

Acknowledgements

We would like to thank our professor, Dr. Karen Karavanic for guidance and assistance, our TA, H. Forrest Alexander for providing advice and technical support, and Brian Delgado for allowing us to use his SMI driver.

References

[1] Delgado, D., Karavanic, K. "Performance Implications of System Management Mode," In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, September 2013.

[2] de Supinski, B. R. "Benchmarking Pthreads Performance," In *PDPTA*, pp. 1985-1991. 1999.

[3] Collins, R. "Intel's System Management Mode," *http://www.rcollins.org/ddj/Jan97/Jan97.html*, accessed June 4, 2015.

[4] Zuo, B., Chen, K., Liang, A., Guan, H., Zhang, J., Ma, R., Yang, H. "Performance tuning

towards a KVM-based low latency virtualization system," In *2010 2nd International Conference on Information Engineering and Computer Science*. 2010. pp. 1-4.

[5] "SMIs Are EEEEVIL (Part2)," http://www.blogs.msdn.com/b/carmencr/archive/2005/09/01/459194.aspx, accessed June 4, 2015.

[6] Azab, A. M., Ning, P., Zhang, X., "Sice: a Hardware-Level Strongly Isolated Computing Environment for x86 Multi-Core Platforms," In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM (2011), pp. 375-388.

[7] Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N. C. "Hypersentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity," In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM (2010), pp. 38-49.

[8] Zhang, F., Leach, K., Sun, K., Satvrou, A. "Spectre: A Dependable Introspection Framework Via System Management Mode," In *Dependable Systems and Networks (DSN), 2013 43 Annual IEEE/IFIP International Conference on*. IEEE (2013), pp. 1-12.

[9] Wang, J., Stavrou, A., Ghosh, A. "Hypercheck: A Hardware Assisted Integrity Monitor," In *Recent Advances in Intrusion Detection*, pp. 158-177. Springer Berlin Heidelberg, 2010.

A. APPENDIX

Run_experiment.py

```
#!/usr/bin/python
#
# CS533, Spring 2015
# Project group B4, Pthreads application effects of SMM
# Master script for generation of experimental data:  total time, SMI count, and ratio of
Pthreads-only time to non-SMI-subtime, given a supplied Pthreads program and our
customized smidriver.
# Author:  Jim Miller (JGM)
# 6/6/15
#
# Usage:  Edit appropriate global variables and program logic, usually at the places marked
"EDITABLE", and run 'run_experiment.py'
```

```
import subprocess

COMMAND_EXTRACT_CPUFREQ = "lscpu | grep MHz | sed 's/^CPU MHz: *//;s/\.//'"
COMMAND_EXTRACT_SMIS = "dmesg | grep smidriver | tail -1 | sed 's/^.*SMI count
//;s/\.//'"
COMMAND_EXTRACT_TSC = "dmesg | grep smidriver | tail -1 | sed 's/^.*TSC //;s/\.//'"
COMMAND_SENSORS = "sensors | grep -v coretemp | grep -v Adapter"

#EDITABLE
COMMAND_PTHREADS_ARG = " 10000 "
#EDITABLE NEXT 3
COMMAND_PTHREADS_ARG_LOWERLIMIT = 1000
COMMAND_PTHREADS_ARG_UPPERLIMIT = 11000
COMMAND_PTHREADS_ARG_STRIDE = 1000
#EDITABLE
COMMAND_PTHREADS_PREFIX = "/project/ptests/lock2 "

#EDITABLE NEXT 7
COMMAND_SMI_1_ARG = ""
COMMAND_SMI_2_ARG = ""
COMMAND_SMI_3_ARG = ""
COMMAND_SMI_4_ARG = ""
COMMAND_SMI_5_ARG = ""
COMMAND_SMI_6_ARG = ""
COMMAND_SMI_7_ARG = ""
#EDITABLE NEXT 3
COMMAND_SMI_ARG_LOWERLIMIT = 50
COMMAND_SMI_ARG_UPPERLIMIT = 1050
COMMAND_SMI_ARG_STRIDE = 100
COMMAND_SMI_1_PREFIX = "echo 1 "
COMMAND_SMI_2_PREFIX = "echo 2 "
COMMAND_SMI_3_PREFIX = "echo 3 "
COMMAND_SMI_4_PREFIX = "echo 4 "
COMMAND_SMI_5_PREFIX = "echo 5 "
COMMAND_SMI_6_PREFIX = "echo 6 "
COMMAND_SMI_7_PREFIX = "echo 7 "
#EDITABLE
COMMAND_SMI_PREFIX = COMMAND_SMI_4_PREFIX
COMMAND_SMI_SUFFIX = " > /proc/smidriver"
COMMAND_SMI_1 = COMMAND_SMI_1_PREFIX + COMMAND_SMI_1_ARG +
COMMAND_SMI_SUFFIX
COMMAND_SMI_2 = COMMAND_SMI_2_PREFIX + COMMAND_SMI_2_ARG +
COMMAND_SMI_SUFFIX
COMMAND_SMI_3 = COMMAND_SMI_3_PREFIX + COMMAND_SMI_3_ARG +
COMMAND_SMI_SUFFIX
COMMAND_SMI_4 = COMMAND_SMI_4_PREFIX + COMMAND_SMI_4_ARG +
COMMAND_SMI_SUFFIX
```

```
COMMAND_SMI_5 = COMMAND_SMI_5_PREFIX + COMMAND_SMI_5_ARG +
COMMAND_SMI_SUFFIX
COMMAND_SMI_6 = COMMAND_SMI_6_PREFIX + COMMAND_SMI_6_ARG +
COMMAND_SMI_SUFFIX
COMMAND_SMI_7 = COMMAND_SMI_7_PREFIX + COMMAND_SMI_7_ARG +
COMMAND_SMI_SUFFIX

DELIMITER = ","
#EDITABLE
EXPERIMENT_FILENAME =
"EXPANDED_LONG_SMIS_smi_frequency_vs_lock2_threads,lock2_iterations=10000.txt"
RESULTS_SAVE_DIR = "/project/results/"
SMIS_FILENAME_PREFIX = "smis-"
TIMINGS_FILENAME_PREFIX = "timings-"
SMIPERCENTAGES_FILENAME_PREFIX = "smi_percentages-"
EXPERIMENT_SMIS_RESULTS_PATH = RESULTS_SAVE_DIR +
SMIS_FILENAME_PREFIX + EXPERIMENT_FILENAME
EXPERIMENT_TIMINGS_RESULTS_PATH = RESULTS_SAVE_DIR +
TIMINGS_FILENAME_PREFIX + EXPERIMENT_FILENAME
EXPERIMENT_SMIPERCENTAGES_RESULTS_PATH = RESULTS_SAVE_DIR +
SMIPERCENTAGES_FILENAME_PREFIX + EXPERIMENT_FILENAME

#EDITABLE - Choices are "AVG", "MIN", "MAX"
REDUCTION_TYPE = "AVG"
REDUCTION_AVG = "(float(sum(REP_VALUES)) / float(len(REP_VALUES)))"
REDUCTION_MIN = "(min(REP_VALUES))"
REDUCTION_MAX = "(max(REP_VALUES))"

#EDITABLE
REPETITIONS = 5

# Based on experiment:  average of 5000 long SMIs generated separately (seconds)
AVG_SHORT_SMI_LENGTH = 0.000055518260
AVG_LONG_SMI_LENGTH = 0.10223856
#EDITABLE
AVG_SMI_LENGTH = AVG_LONG_SMI_LENGTH

repTimings = []
repSMIs = []
repPthreadsAloneTimings = []
SMICounts = []
TSCCounts = []
reducedSMI = 0.00000000
reducedTiming = 0.00000000
cpufreq = 0.00000000
SMIsResultsFile = open(EXPERIMENT_SMIS_RESULTS_PATH, 'a')
timingsResultsFile = open(EXPERIMENT_TIMINGS_RESULTS_PATH, 'a')
```

```
SMIPercentagesResultsFile =
open(EXPERIMENT_SMIPERCENTAGES_RESULTS_PATH, 'a')
measurementNumber = 0
cnt = 0

def runPthreadsCommandsAlone():
  global TSCCounts
  global repPthreadsAloneTimings
  global measurementNumber

  for i in xrange(0, REPETITIONS):
    subprocess.call(COMMAND_SMI_7, shell = True)
    proc = subprocess.Popen(COMMAND_EXTRACT_TSC, stdout = subprocess.PIPE, shell
= True)
    proc.wait()
    TSCCounts.append(proc.stdout.read())

    subprocess.call(COMMAND_PTHREADS_EXPERIMENT, shell = True)

    subprocess.call(COMMAND_SMI_7, shell = True)
    proc = subprocess.Popen(COMMAND_EXTRACT_TSC, stdout = subprocess.PIPE, shell
= True)
    proc.wait()
    TSCCounts.append(proc.stdout.read())

    repTimings.append(float((float(TSCCounts[1]) - float(TSCCounts[0])) / (cpufreq *
1000.00000000)))
    TSCCounts = []

  if REDUCTION_TYPE == "AVG":
    reducedTiming = (float(sum(repTimings)) / float(len(repTimings)))
  else:
    if REDUCTION_TYPE == "MIN":
      reducedTiming = (min(repTimings))
    else:
      if REDUCTION_TYPE == "MAX":
        reducedTiming = (max(repTimings))
  repPthreadsAloneTimings.append(reducedTiming)

  proc = subprocess.Popen(COMMAND_SENSORS, stdout = subprocess.PIPE, shell =
True)
  proc.wait()
  measurementNumber += 1
  print "Measurement:  " + str(measurementNumber) + '\n' + proc.stdout.read()

def runPthreadsCommandsWithSMIInterference():
  global TSCCounts
```

```
  global SMICounts
  global repTimings
  global repSMIs
  global measurementNumber
  global reducedSMI
  global reducedTiming

  for i in xrange(0, REPETITIONS):
    subprocess.call(COMMAND_SMI_6, shell = True)
    proc = subprocess.Popen(COMMAND_EXTRACT_SMIS, stdout = subprocess.PIPE,
shell = True)
    proc.wait()
    SMICounts.append(proc.stdout.read())

    subprocess.call(COMMAND_SMI_7, shell = True)
    proc = subprocess.Popen(COMMAND_EXTRACT_TSC, stdout = subprocess.PIPE, shell
= True)
    proc.wait()
    TSCCounts.append(proc.stdout.read())

    subprocess.call(COMMAND_SMI_EXPERIMENT, shell = True)

    subprocess.call(COMMAND_PTHREADS_EXPERIMENT, shell = True)

    subprocess.call(COMMAND_SMI_5, shell = True)

    subprocess.call(COMMAND_SMI_7, shell = True)
    proc = subprocess.Popen(COMMAND_EXTRACT_TSC, stdout = subprocess.PIPE, shell
= True)
    proc.wait()
    TSCCounts.append(proc.stdout.read())

    subprocess.call(COMMAND_SMI_6, shell = True)
    proc = subprocess.Popen(COMMAND_EXTRACT_SMIS, stdout = subprocess.PIPE,
shell = True)
    proc.wait()
    SMICounts.append(proc.stdout.read())

    repTimings.append(float((float(TSCCounts[1]) - float(TSCCounts[0])) / (cpufreq *
1000.00000000)))
    repSMIs.append(float((float(SMICounts[1]) - float(SMICounts[0]))))
    TSCCounts = []
    SMICounts = []

  if REDUCTION_TYPE == "AVG":
    reducedTiming = (float(sum(repTimings)) / float(len(repTimings)))
    reducedSMI = (float(sum(repSMIs)) / float(len(repSMIs)))
```

```
  else:
    if REDUCTION_TYPE == "MIN":
      reducedTiming = (min(repTimings))
      reducedSMI = (min(repSMIs))
    else:
      if REDUCTION_TYPE == "MAX":
        reducedTiming = (max(repTimings))
        reducedSMI = (max(repSMIs))
  timingsResultsFile.write("%.8f" % reducedTiming)
  SMIsResultsFile.write("%.8f" % reducedSMI)

  proc = subprocess.Popen(COMMAND_SENSORS, stdout = subprocess.PIPE, shell =
True)
  proc.wait()
  measurementNumber += 1
  print "Measurement:  " + str(measurementNumber) + '\n' + proc.stdout.read()

  repTimings = []
  repSMIs = []

proc = subprocess.Popen(COMMAND_EXTRACT_CPUFREQ, stdout = subprocess.PIPE,
shell = True)
proc.wait()
cpufreq = float(proc.stdout.read())

timingsResultsFile.write(DELIMITER)
SMIsResultsFile.write(DELIMITER)
SMIPercentagesResultsFile.write(DELIMITER)
for i in xrange(COMMAND_PTHREADS_ARG_LOWERLIMIT,
COMMAND_PTHREADS_ARG_UPPERLIMIT, COMMAND_PTHREADS_ARG_STRIDE):
  if i != COMMAND_PTHREADS_ARG_UPPERLIMIT -
COMMAND_PTHREADS_ARG_STRIDE:
    timingsResultsFile.write(str(i) + DELIMITER)
    SMIsResultsFile.write(str(i) + DELIMITER)
    SMIPercentagesResultsFile.write(str(i) + DELIMITER)
  else:
    timingsResultsFile.write(str(i) + '\n')
    SMIsResultsFile.write(str(i) + '\n')
    SMIPercentagesResultsFile.write(str(i) + '\n')

for i in xrange(COMMAND_PTHREADS_ARG_LOWERLIMIT,
COMMAND_PTHREADS_ARG_UPPERLIMIT, COMMAND_PTHREADS_ARG_STRIDE):
  #EDITABLE
  COMMAND_PTHREADS_EXPERIMENT = COMMAND_PTHREADS_PREFIX + str(i) +
COMMAND_PTHREADS_ARG
  #COMMAND_PTHREADS_EXPERIMENT = COMMAND_PTHREADS_PREFIX +
COMMAND_PTHREADS_ARG + str(i)
```

```
  runPthreadsCommandsAlone()

for i in xrange(COMMAND_SMI_ARG_LOWERLIMIT,
COMMAND_SMI_ARG_UPPERLIMIT, COMMAND_SMI_ARG_STRIDE):
  timingsResultsFile.write(str(i) + DELIMITER)
  SMIsResultsFile.write(str(i) + DELIMITER)
  SMIPercentagesResultsFile.write(str(i) + DELIMITER)
  #EDITABLE
  COMMAND_SMI_EXPERIMENT = COMMAND_SMI_PREFIX + str(i) +
COMMAND_SMI_SUFFIX
  cnt = 0
  for j in xrange(COMMAND_PTHREADS_ARG_LOWERLIMIT,
COMMAND_PTHREADS_ARG_UPPERLIMIT, COMMAND_PTHREADS_ARG_STRIDE):
    #EDITABLE
    COMMAND_PTHREADS_EXPERIMENT = COMMAND_PTHREADS_PREFIX + str(j) +
COMMAND_PTHREADS_ARG
    #COMMAND_PTHREADS_EXPERIMENT = COMMAND_PTHREADS_PREFIX +
COMMAND_PTHREADS_ARG + str(j)
    runPthreadsCommandsWithSMIInterference()
    SMIPercentagesResultsFile.write("%.8f" % (repPthreadsAloneTimings[cnt] /
(reducedTiming - AVG_SMI_LENGTH * reducedSMI)))
    cnt += 1
    if j != COMMAND_PTHREADS_ARG_UPPERLIMIT -
COMMAND_PTHREADS_ARG_STRIDE:
      timingsResultsFile.write(DELIMITER)
      SMIsResultsFile.write(DELIMITER)
      SMIPercentagesResultsFile.write(DELIMITER)
    else:
      timingsResultsFile.write('\n')
      SMIsResultsFile.write('\n')
      SMIPercentagesResultsFile.write('\n')

timingsResultsFile.write(DELIMITER)
cnt = 0
for i in xrange(COMMAND_PTHREADS_ARG_LOWERLIMIT,
COMMAND_PTHREADS_ARG_UPPERLIMIT, COMMAND_PTHREADS_ARG_STRIDE):
  timingsResultsFile.write("%.8f" % repPthreadsAloneTimings[cnt])
  cnt += 1
  if i != COMMAND_PTHREADS_ARG_UPPERLIMIT -
COMMAND_PTHREADS_ARG_STRIDE:
    timingsResultsFile.write(DELIMITER)
  else:
    timingsResultsFile.write('\n')

SMIsResultsFile.close()
timingsResultsFile.close()
SMIPercentagesResultsFile.close()
```

create.c :

```
// Pthreads test suite
// Thread creation test
// Usage: ./create [<numThreads>] [<iterations>]
// This test measures pthread_create time.  The main thread calls the recursive worker
// function which increments a counter and creates another thread.  An implicit call
// to pthread_exit() occurs when each thread leaves the worker function.
// The master thread waits on a condition variable until all the creations have occured.
// The second command line argument specifies the number of times the test should
// repeat; default = 1.

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int numThreads = 10;
int iterations = 1;
int counter = 0;
pthread_mutex_t condLock;
pthread_cond_t done;

void worker() {
  pthread_t thread;

  ++counter;
  //printf("Thread %d\n", counter);
  if (counter < numThreads)
    pthread_create(&thread, NULL, (void*)worker, NULL);
  else
    pthread_cond_signal(&done);
  pthread_exit(NULL);
}


int main(int argc, char **argv) {
  if (argc > 1)
    numThreads = atoi(argv[1]);
  if (argc > 2)
    iterations = atoi(argv[2]);

  //printf("\nnumThreads: %d\niterations: %d\n", numThreads, iterations);
```

```
  pthread_t thread;
  pthread_mutex_init(&condLock, NULL);
  pthread_cond_init(&done, NULL);
  for (int i = 0; i < iterations; ++i) {
    pthread_create(&thread, NULL, (void*)worker, NULL);
    pthread_cond_wait(&done, &condLock);
    counter = 0;
  }
}
```

yield.c :

```
// Pthreads test suite
// Thread yield test
// Usage: ./yield [<numThreads>] [<iterations>]
// This test attempts to measure the speed of context shifts
// between threads by having each thread repeatedly call sched_yield.
// The worker function pins all the threads to the same core in an
// (unsuccessful) attempt to force them to strictly alternate.
// POSIX dosn't define the behavior of sched_yield(), and a call
// to sched_yield() doesn't necessarily force the thread to yield.

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <unistd.h>

int iterations = 100;
int numThreads = 2;

void worker() {
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(0, &cpuset);
  pthread_t current_thread = pthread_self();
  pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &cpuset);
  for (int i = 0; i < iterations; ++i) {
```

```
            //printf("Yield %d\n", i);
            sched_yield();
    }
}


int main(int argc, char **argv) {
    if (argc > 1)
            numThreads = atoi(argv[1]);
    if (argc > 2)
            iterations = atoi(argv[2]);


    pthread_t thread[numThreads];
    for (int i = 0; i < numThreads; ++i)
            pthread_create(&thread[i], NULL, (void*)worker, NULL);
    for (int i = 0; i < numThreads; ++i)
            pthread_join(thread[i], NULL);
}
```

lock.c :

```
// Pthreads test suite
// Lock 'ping-pong' test
// Usage: ./lock [<iterations>]
// This mutex measures the time of two
// threads repeatedly locking and
// unlocking a set of four locks in a prescribed
// order.  The main thread starts out holding two of
// the locks, and waits on a condition variable for the
// worker thread to initialize its locks.  The threads
// then 'ping-pong' the locks back and forth.

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int iterations = 100;
int counter = 0;
pthread_mutex_t testLock[4];
```

```
pthread_mutex_t condLock;
pthread_cond_t ready;


void worker() {
  pthread_mutex_lock(&testLock[1]);
  pthread_mutex_lock(&testLock[3]);
  pthread_mutex_lock(&condLock);
  pthread_cond_signal(&ready);
  pthread_mutex_unlock(&condLock);
  for (int i = 0; i < iterations; ++i) {
    pthread_mutex_lock(&testLock[0]);
    //printf("Locked 0\n");
    pthread_mutex_unlock(&testLock[1]);
    //printf("Unlocked 1\n");
    pthread_mutex_lock(&testLock[2]);
    //printf("Locked 2\n");
    pthread_mutex_unlock(&testLock[3]);
    //printf("Unlocked 3\n");
    pthread_mutex_lock(&testLock[1]);
    //printf("Locked 1\n");
    pthread_mutex_unlock(&testLock[0]);
    //printf("Unlocked 0\n");
    pthread_mutex_lock(&testLock[3]);
    //printf("Locked 3\n");
    pthread_mutex_unlock(&testLock[2]);
    //printf("Unlocked 2\n");
  }
}


int main(int argc, char **argv) {
  if (argc > 1)
    iterations = atoi(argv[1]);

  pthread_mutex_init(&condLock, NULL);
  pthread_cond_init(&ready, NULL);
  pthread_t thread;
  for (int i = 0; i < 4; ++i)
    pthread_mutex_init(&testLock[i], NULL);
  pthread_mutex_lock(&testLock[0]);
  pthread_mutex_lock(&testLock[2]);
  pthread_create(&thread, NULL, (void*)worker, NULL);
  pthread_mutex_lock(&condLock);
  pthread_cond_wait(&ready, &condLock);
  pthread_mutex_unlock(&condLock);
  for (int i = 0; i < iterations; ++i) {
```

```
    pthread_mutex_unlock(&testLock[0]);
    pthread_mutex_lock(&testLock[1]);
    pthread_mutex_unlock(&testLock[2]);
    pthread_mutex_lock(&testLock[3]);
    pthread_mutex_unlock(&testLock[1]);
    pthread_mutex_lock(&testLock[0]);
    pthread_mutex_unlock(&testLock[3]);
    pthread_mutex_lock(&testLock[2]);
  }
  pthread_join(thread, NULL);
}
```

lock2.c :

```
// Pthreads test suite
// Lock contention test
// Usage: ./lock2 [<numThreads>] [<iterations>]
// This test measures the time for multiple threads competing
// to access a single lock.

// Warning: on wyeast08 this test is seg faulting on inputs of
// numThreads > 30000, iterations = 1
// and numThreads > 4000, iterations > 1
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int numThreads = 10;
int iterations = 1;
int counter = 0;
pthread_mutex_t testLock;

void worker() {
  pthread_t thread;

  for (int i = 0; i < iterations; ++i)
  {
    pthread_mutex_lock(&testLock);
    //printf("Locking\n");
    pthread_mutex_unlock(&testLock);
    //printf("Unlocking\n");

  }
```

```
}


int main(int argc, char **argv) {
  if (argc > 1)
    numThreads = atoi(argv[1]);
  if (argc > 2)
    iterations = atoi(argv[2]);

  pthread_t thread[numThreads];
  pthread_mutex_init(&testLock, NULL);
  for (int i = 0; i < numThreads; ++i)
    pthread_create(&thread[i], NULL, (void*)worker, NULL);
  for (int i = 0; i < numThreads; ++i)
    pthread_join(thread[i], NULL);
}
```

condition.c :

```
// Pthreads test suite
// Condition variable wait and signal test
// Usage: ./condition [<iterations>]
//
// One worker thread repeatedly waits on a condition
// variable, while the main thread repeatedly signals
// it to wake up.  The worker increments a variable
// (iterations times) before waiting again.

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int iterations = 1000;
pthread_cond_t testCond;
pthread_mutex_t testLock;
int count = 0;

void worker() {
  do {

    pthread_mutex_lock(&testLock);
    pthread_cond_wait(&testCond, &testLock);
    ++count;
```

```
    printf("Count: %d\n", count);
    pthread_mutex_unlock(&testLock);
  }
  while (count < iterations);

}



int main(int argc, char **argv) {
  if (argc > 1)
    iterations = atoi(argv[1]);

  pthread_t thread;
  pthread_mutex_init(&testLock, NULL);
  pthread_cond_init(&testCond, NULL);
  pthread_create(&thread, NULL, (void*)worker, NULL);
  do {

    pthread_mutex_lock(&testLock);
    pthread_cond_signal(&testCond);
    pthread_mutex_unlock(&testLock);

  }
  while (count < iterations);
}
```

condition2.c :

```
// Pthreads test suite
// Multi-thread signal/wait test
// Usage: ./condition2 [<numThreads>] [<iterations>]

// For some unknown reason this freezes for thread numbers above
// approximately 31880 (in linux.cs.pdx.edu)
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int numThreads = 500;
int iterations = 1;
int count = 0;
pthread_mutex_t countLock;
```

```
pthread_mutex_t condLock[5];
pthread_cond_t cond[5];


void worker() {
  int toWait, toSignal;

  for (int i = 0; i < iterations; ++i)
  {
        for (int j = 0; j < 5; ++j)
        {
    toWait = (i + j) % 5;
    pthread_mutex_lock(&condLock[toWait]);
    //printf("Waiting on condition  %d\n", toWait);
    pthread_cond_wait(&cond[toWait],  &condLock[toWait]);
    pthread_mutex_unlock(&condLock[toWait]);
        toSignal = (toWait + 1) % 5;
    pthread_mutex_lock(&condLock[toSignal]);
    //printf("Signaling condition %d\n", toSignal);
    pthread_cond_signal(&cond[toSignal]);
    pthread_mutex_unlock(&condLock[toSignal]);
        }
  }
  pthread_mutex_lock(&countLock);
  ++count;
  printf("Thread %d done\n", count);
  pthread_mutex_unlock(&countLock);
}




int main(int argc, char **argv) {
  if (argc > 1)
        numThreads = atoi(argv[1]);
  if (argc > 2)
        iterations = atoi(argv[2]);

  pthread_mutex_init(&countLock, NULL);
  pthread_t thread[numThreads];
```

```
  for (int i = 0; i < 4; ++i)
  {
        pthread_mutex_init(&condLock[i], NULL);
        pthread_cond_init(&cond[i], NULL);
  }
  for (int i = 0; i < numThreads; ++i)
        pthread_create(&thread[i], NULL, (void*)worker, NULL);
  while (count < numThreads )
  {
        for (int i = 0; i < 5; ++i)
        {
        pthread_mutex_lock(&condLock[i]);
        pthread_cond_signal(&cond[i]);
        pthread_mutex_unlock(&condLock[i]);
        }
  }
  for (int i = 0; i < numThreads; ++i)
        pthread_join(thread[i], NULL);

}
```