Handling OS Jitter on Multicore Multithreaded Systems*

Pradipta De Vijay Mann
IBM India Research Lab
New Delhi
Email: {pradipta.de, vijamann}@in.ibm.com

Umang Mittal†
Indian Institute of Technology,
New Delhi
Email: umang.k.mittal@gmail.com

Abstract

Various studies have shown that OS jitter can degrade parallel program performance considerably at large processor counts. Most sources of system jitter fall broadly into 5 categories - user space processes, kernel threads, interrupts, SMT interference and hypervisor activity. Solutions to OS jitter typically consist of a combination of techniques such as synchronization of jitter across nodes (co-scheduling or gang scheduling) and use of microkernels. Both techniques present several drawbacks. Multicore and Multithreaded systems present opportunities to handle OS jitter. They have multiple cores and threads, some of which can be used for handling OS jitter, while the application threads run on remaining cores and threads. However, they are also prone to risks such as inter-thread cache interference and process migration. In this paper, we present a holistic approach that aims to reduce jitter caused by various sources of jitter by utilizing the additional threads or cores in a system. Our approach handles jitter through reduction of kernel threads, intelligent interrupt handling, and switching of hardware SMT thread priorities. This helps in reducing jitter experienced by application threads in the user space, at the kernel level, and at the hardware level. We make use of existing features available in the Linux kernel and Power Architecture as well make enhancements to the Linux kernel. We demonstrate the efficacy of our techniques by reducing jitter on two different platforms and operating system versions. In the first case our approach helps in reducing periodic jitter that improves both average and worst case performance of a simulated parallel application. In the second case our approach helps in reducing infrequent very large jitter that helps the worst case performance of a real parallel application. Our experimental results show up to 30% reduction in slowdown in the average case at 16K OS images and up to 50% reduction in slowdown in the worst case at 8 OS images using this approach as compared to a baseline configuration.

1. Introduction

OS jitter (henceforth referred to as OS Jitter) refers to the interference experienced by an application due to activities inside an operating system. Petrini et al. showed that operating system interference can cause up to 100% performance degradation at 4096 processors in ASCI Q [1]. Our earlier work [2] showed that most sources of system jitter fall broadly into 3 categories - user space processes, kernel threads and interrupts (refer Figure 1). SMT interference and hypervisor activity are the two new additions to this traditional list of sources of jitter as next generation petascale systems try to exploit the benefits of simultaneous multithreading and virtualization.

Solutions to OS jitter typically consist of a combination of techniques such as synchronization of jitter across nodes (co-scheduling or gang scheduling) and use of microkernels. Both techniques present several drawbacks. Synchronization of all jitter across nodes using periodic priority boosts requires either a global switch clock against which all node clocks should be synchronized or use of software techniques such as NTP. There is also the issue of clock drift with time. Microkernels, on the other hand, limit the use of massively parallel systems as existing applications need to be ported to these kernels.

Multicore and Multithreaded systems present opportunities to handle OS jitter. They have multiple cores and threads, some of which can be used for handling OS jitter, while the application threads run on remaining cores and threads. The technique of leaving one of the CPUs of a n-CPU SMP machine idle has been used by HPC systems to handle jitter and it has proved to be effective [1]. While, it has the disadvantage of losing out on some compute power per node, with larger number of cores per node, this loss is increasingly becoming smaller in percentage terms. Similarly, leaving sister CPU threads idle on a SMT machine is typically the

*This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002

†*This work was done while the author was a summer intern at IBM India Research Laboratory*

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

Jitter Source	Lowest Interruption (us)	Highest Interruption (us)	Total Frequency	Mean (us)	Total Jitter %
timer	1.73	3397.14	164360	3.39	85.483
eth0	4.6	22.55	764	9.04	1.062
events0	1.49	166.4	3148	4.07	1.97
rpciod0	45.08	46.8	1	46.79	0.007
rtasd	22.72	38.23	396	28.21	1.716
pdflush	1.45	79.9	594	6.19	0.565
watchdog	1.31	2.12	1384	1.54	0.326
nfsd4	5.09	6.42	33	5.72	0.029
nfsd	6.45	6.47	8	6.47	0.008
irqbalan	145.01	200.39	297	159.88	7.295
init	1.49	30.85	595	16.82	1.538

Interrupts

WorkQueue Kernel Threads

Other Kernel Threads

User level Processes

Figure 1. Different categories of sources of system jitter

recommended way to run a HPC workload that is known to be sensitive to SMT interference. However, in both the scenarios it is left to the operating system to do a good job of load balancing and scheduling all sources of jitter on the idle threads or cores. While this works well for user level processes, it does not work for other sources of jitter such as interrupts and kernel threads. Moreover, there is also the risk SMT inter-thread cache interference and process migration.

In this paper, we present a holistic approach that aims to reduce jitter caused by various sources of jitter by utilizing the additional threads or cores in a system. Our approach handles jitter through reduction of kernel threads, intelligent interrupt handling, and switching of hardware SMT thread priorities. This helps in reducing jitter experienced by application threads in the user space, at the kernel level, and at the hardware level. We make use of existing features available in the Linux kernel and Power Architecture as well make enhancements to the Linux kernel. We demonstrate the efficacy of our techniques by reducing jitter on two different platforms and operating system versions. In the first case our approach helps in reducing periodic jitter that improves both average and worst case performance of a parallel application. In the second case our approach helps in reducing infrequent very large jitter that helps the worst case performance of a parallel application.

The main contributions of this paper are the following:

1) Design of an overall methodology that tackles system jitter at the user level, within the operating system kernel and at the hardware level.
2) Enhancements to Linux Kernel for handling jitter caused by various sources of jitter - user level processes, kernel threads, interrupts and SMT interference.
3) Experimental validation of the overall approach that indicates up to 30% reduction in average case slowdown at 16K OS images for a simulated parallel application and up to 50% reduction in worst case slowdown at 8 OS images for a real application.

The rest of this paper is organized as follows. Section 2 describes the overall design and implementation of our approach. In Section 3 we describe our experimental methodology. Section 4 presents our experimental results. Section 5 gives on overview of related research. Section 6 summarizes our findings and gives directions for future research.

2. Design and Implementation

In this section we present the overall design and implementation of our approach. Our approach is based on the model that most parallel applications run one thread per physical CPU and the sister SMT threads on that processor can be utilized for handling jitter.

As confirmed by our earlier work on identifying sources of jitter, sources of jitter can be classified broadly as user space processes, kernel threads and interrupts. Interference from code running on sister SMT threads and hypervisor activity also becomes important as most modern commodity clusters move to multithreaded processors and virtualization technologies. We do not discuss hypervisor jitter in this paper. We describe these sources of jitter and the solutions to each one of them below. Figure 2 gives an overview of these sources of jitter and solutions to each.[1]

1. Typically, on Power architecture, even numbered CPUs are referred to as “primary SMT threads” and odd numbered CPUs are referred to as “secondary SMT threads”. In this paper, we refer to odd numbered CPUs as ”primary SMT threads” and even numbered CPUs as “secondary SMT threads” due to implementation problems we encountered in switching off even numbered CPUs - a requirement for “Isolated CPUs” configuration.

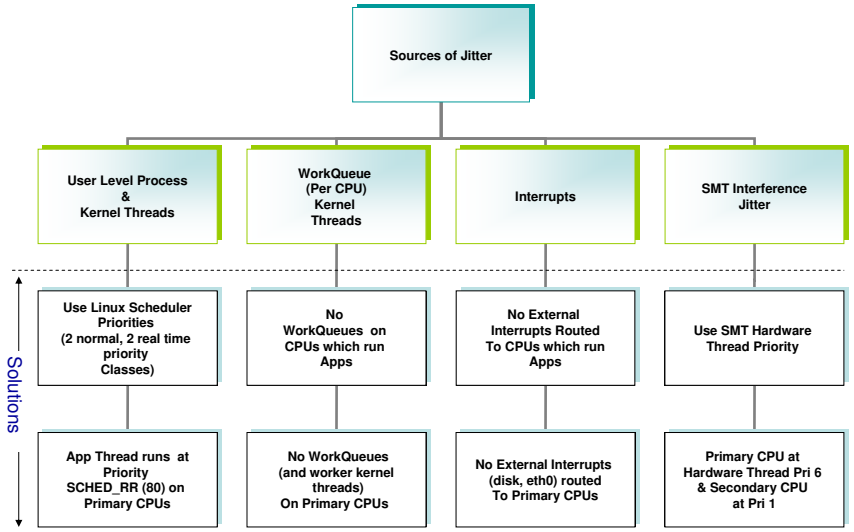

Figure 2. An overview of the various sources of jitter and our solutions to each

2.1. OS Priority: handling user space processes and kernel threads

Most modern commodity operating systems such as Linux, provide various scheduling policies and priorities that can be used by an application program to control the level of interruptions it experiences from other user space processes and kernel threads. Linux provides 4 scheduling policies - SCHED_OTHER (default), SCHED_RR (real time), SCHED_FIFO (real time), and SCHED_BATCH. It also provides scheduling priority levels from 1 to 99 (real time priorities, increasing in priority) and 100 to 139 (normal, decreasing in priority). Under Linux, the scheduling priority and policy of even the kernel threads can be specified from user space. For more details on Linux scheduling policies and priorities, please refer [3].

Kernel threads can again be broadly classified as belonging to two groups - those which are created per CPU and those which are created once for an entire OS image. Per CPU kernel threads are typically worker threads that handle tasks that are enqueued to their respective workqueues. Workqueues are part of generic infrastructure that Linux maintains for completing various tasks. Various Linux subsystems can enqueue their task requests onto workqueues which are then handled by a pool of worker threads. These worker threads are created one per CPU. Linux maintains workqueues for various tasks such as writing to a block device (kblockd), softirq handling (ksoftirqd) and asynchronous IO (aio). There is also a default workqueue (keventd)that can be used by kernel device drivers. The number of worker threads on a large multithreaded system can quickly grow very large. For example, a 32 processor 2 way SMT machine will have 64 worker threads for each of the workqueues mentioned above. Clearly, all worker threads are not required on all CPUs most of the time.

Note that running the application threads at really high priority can starve other kernel threads that are essential for the system. It has been recommended that application threads do not run at priority levels higher than 89 [4]. In our experimentation, the application threads are run at priority 80 with SCHED_RR policy. We call this configuration the “OS Priority” configuration.

Linux scheduling policies and priorities have been designed to maximize average case performance instead of worst case performance. Many of its performance improvement techniques decrease average time by increasing the worst case time [5]. This restricts the use of Linux for large clusters that run fine grained parallel applications. Hence, it is unlikely that scheduling priorities alone can provide a solution.

2.2. Isolated CPUs: removing house keeping - process scheduling and interrupt routing from target CPUs

OS scheduling policies and priorities have no impact on handling of interrupts. While, Linux allows user space modification of a SMP interrupt affinity mask,

which controls what interrupts get handled by what CPUs, it does not control routing of interrupt bottom halves or softirqs.

Furthermore, while elevating an application's scheduling policy and priority will help reduce interruptions from other user space processes and kernel threads, it will not completely remove it. It is desirable to completely remove a CPU from all scheduling decisions and interrupt handling. Such a CPU can be used for scheduling a process or kernel thread only by explicitly binding the process to it.

The above features have been explored by the Linux community in the form of isolated CPUs [6]. It consists of a set of patches some of which are architecture dependent and have been implemented for x86 architecture. While, these features are not part of the mainline kernel and some of them have been replaced by other features such as scheduling domains and cpusets, they continue to be used in some systems. We evaluated the effectiveness of the above features and ported the available kernel patches to POWER architecture. This required implementing some architecture dependent features such as interrupt routing from scratch. We call this configuration the "Isolated CPUs" configuration.

2.3. Hardware Thread Priorities: reducing SMT interference

Simultaneous multithreading (SMT) allows instructions from multiple instructions streams (threads) to be executed simultaneously in one clock cycle. This helps in increasing system throughput as it reduces "pipeline bubbles" that get created due to limited instruction level parallelism. SMT implementation usually involves duplication of some resources and sharing of certain other resources such as processor caches. SMT interference refers to the side-effects of sharing of these resources by the various hardware threads on the processor. Most of the SMT interference comes from sharing of caches. Various studies [7] have shown that a lot of applications can degrade in performance under SMT due to cache pollution.

POWER5 and POWER6 architectures implement 8 different hardware thread priority levels or SMT priorities. The priority difference between the two threads controls how many instructions get fetched from each instruction stream per cycle, which in turn determines the extent of SMT interference an application experiences from the code running on its sister SMT thread. Out of the 8 priorities, two priorities (0 - thread shut down and 7 - single threaded mode) can be invoked only by the hypervisor. Out of the remaining 6, 3 priority levels (1,5 and 6) can only be set from within the kernel while the remaining 3 levels (2,3, and 4) can be set by user applications. The default priority level in both POWER5 and POWER6 is 4. It should be noted, however, that even if a user application sets a particular hardware thread priority, the Linux kernel resets it to the default value upon return from an interrupt context. More information about SMT priorities on POWER architecture can be found in [8].

We made changes to the Linux kernel and developed a patch that contains a set of system calls that can be used by user applications to set any hardware thread priority from 1 to 6. Changes were also made to ensure that these priorities are not reset by the kernel upon return from an interrupt context. Other researchers [8] have investigated the use of hardware thread priorities on Power architecture to enhance the overall parallel program performance which can deteriorate due to load imbalance between the two SMT threads. We explore the effect of hardware thread priorities and reduction in SMT interference by setting the hardware priority of the primary SMT thread to 6 and that of the secondary SMT thread to 1. We call this configuration the "Hardware Priority" configuration.

3. Experimental Methodology

Our experimental methodology consists mainly of the following steps:

1) Collection of jitter traces along with kernel data to identify sources of jitter
2) Simulating slowdown due to jitter using the traces collected to predict slowdown at large processor counts
3) Running a parallel benchmark on a real cluster when available

We now describe each of these steps in detail.

3.1. Jitter Trace Collection

We first collect a jitter trace for all the configurations described in the Section 2 using a trace collector benchmark. The trace collector benchmark executes a tight loop that reads the timestamp register or the cycle counter register on the processor and calculates the difference between successive readings (timestamp deltas). If the timestamp deltas exceed a particular threshold, then it is considered a jitter event. This produces a jitter trace. Trace Collector benchmark has been described in detail in our earlier work [2] [9] and in [10]. While we run the trace collector benchmark, we also collect kernel data using the methodology described in [2]. This helps us identify the sources of jitter in the trace.

In all configurations we keep both SMT threads switched on and the trace collector benchmark is bound to the primary SMT thread while the secondary SMT

thread is kept idle. In all configurations, except the Hardware Priority configuration, both SMT threads run at their default priority of 4. This forms the Baseline SMTON configuration. In the OS Priority configuration, we boost the scheduling priority of the trace collector benchmark on the primary SMT thread to 80 and run it with SCHED_RR real time scheduling policy. In the Hardware Priority configuration, we boost the hardware priority of the primary SMT thread which runs the trace collector benchmark to 6 while lowering the hardware priority of the secondary SMT thread to 1. Finally, in the Isolated CPU configuration, we isolate the primary SMT thread which runs the trace collector benchmark. As a result, no workqueues and corresponding worker kernel threads are created on the primary SMT thread and all interrupts (external interrupts and softirqs or bottom halves) are routed to secondary SMT thread.

3.2. Simulating slowdown due to jitter using the traces collected

The simulation process is based on our earlier work on emulating OS jitter on BlueGene/L using a trace driven approach [9]. Using the same trace driven approach, we conduct a simulation on a smaller cluster, where the barrier is assumed to be instantaneous and the slowdown caused by jitter in only the compute phase is predicted. A parallel application that executes a compute bound loop of 1 ms followed by an instantaneous barrier across all parallel threads is simulated. To simulate unsynchronized jitter, the simulator starts at different randomly chosen indices in the trace. These different indices represent the starting point of parallel tasks (or threads) running on different processors.

The simulator is a cycle accurate simulator that takes as input a jitter trace, work quanta value in time and frequency of the platform from where the trace has been collected. It first converts the work quanta in time into the target compute cycles in each phase by multiplying it by the frequency value. It then computes the difference between successive start timestamps in the jitter trace to calculate the available application runtime which can be used for compute cycles. At each simulated parallel task, it keeps adding the application runtime cycles till a jitter value is reached or the target compute cycles for a phase is reached. If a jitter value is reached, it is added to the total cycles consumed. When the target compute cycles are reached, it represents the end of a compute phase and entry into the barrier phase. Due to jitter being introduced, different tasks will report a different value of the total cycles consumed in each phase, and maximum of all those values is calculated to predict the overall completion time for that phase. The compute time taken by the faster tasks is subtracted from this maximum completion time to calculate the number of cycles each task should wait (wasted cycles) before starting the next compute phase. After the end of N compute phases (N represents end of work or an experiment), an average of these overall completion times for each phase is calculated. Advantage of using a simulator approach is that one can use it to predict scalability at any processor count. Disadvantage is that it does not take into account the actual barrier phase.

For the purposes of this paper, we simulate up to 16K OS images. This corresponds to about half a million parallel tasks on a 32 way SMP node. This is the number of parallel tasks some of the next generation petascale systems will support.

3.3. Running a parallel benchmark on a real cluster

One set of our experiments was conducted on a cluster of 256 processors (8 nodes with 32 cores each). For those experiments, we ran a parallel benchmark on all the 256 cores. Parallel benchmark represents a typical parallel application with repeated compute-barrier phases. Each processor in the cluster runs a MPI task that executes the parallel benchmark kernel.

The main kernel represents a fixed amount of work that is expected to finish in a given time (referred to as quanta). Number of iterations or amount of work required to consume quanta time is pre-calculated using a calibration step prior to executing the kernel. The work done can be any operation. For the experiments in this paper, we have chosen it to be a Linear Congruential Generator (LCG) operation defined by the recurrence relation:

$$x(j+1) = (a * x(j) + b) mod p \tag{1}$$

At the end of the compute phase, there is a barrier call for synchronization. If an OS activity interrupts a MPI task either during the compute phase or the barrier phase, it slows down other MPI tasks in the cluster that have already completed their work and have entered the barrier call.

We measure the time (using cycle accurate timers) spent in the compute phase, referred to as qt, time spent in the barrier call, referred to as bt, and the total time for completing a phase, referred to as qtbt. More details about the parallel benchmark can be found in [9] [11].

4. Experimental Validation

In this section we present our experimental validation of our approach. We conducted two sets of experiments on different platforms, operating system versions and with different number of processors.

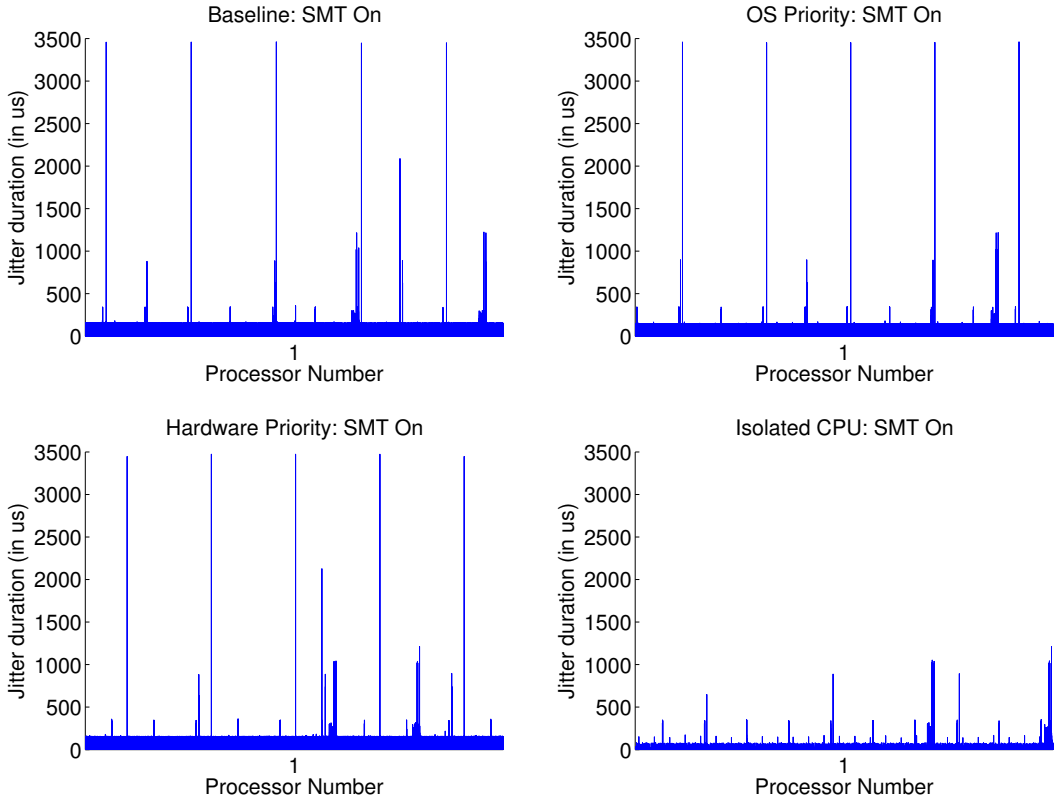

Figure 3. Jitter timeseries for different configurations on a 1-Processor 2-Way SMT Power5 node

4.1. Experiments on single 1-processor 1.5 GHz 2-way SMT POWER5 Node

The first set of experiments was done on a 1 processor, 2 way SMT, POWER5 node running Fedora Core 6 with kernel version 2.6.24.

4.1.1. Jitter trace collection on a single node. We first collected a jitter trace using the trace collector benchmark and then simulated the performance of various configurations for a parallel application that does a barrier after every 1 ms of computation for up to 16K OS images. The trace collector was executed to collect a million jitter events for each of the configurations. This takes approximately 85 minutes for each experiment on the system we used.

The jitter traces for the four configurations are shown in Figure 3, where y-axis represents jitter duration in μs and x-axis represents jitter samples. The jitter trace for Baseline SMTON configuration shows most jitter to be around 200 μs. There are some peaks in the 800-1100 μs region, one peak around 2000 μs and some very high periodic peaks around 3500 μs that occur once approximately every 17 minutes. Use of real time scheduling policy and priority in the OS priority configuration helps reduce some jitter peaks, such as those in the 800-1100 μs and the peak around 2000 μs. However, the very high periodic peaks around 3500 μs remain unaffected. Boosting the hardware priority of the primary SMT thread to 6 (and reducing that of the secondary SMT thread to 1) in the Hardware priority configuration produces a jitter trace that is very similar to the Baseline SMTON configuration.

The Isolated CPU configuration produces the most improved jitter trace. In stark contrast with other configurations, there are no high periodic peaks around 3500μs. Even the overall average jitter level has been reduced from 200μs in the baseline configuration to less than 100 μs.The highest peaks in this configuration are restricted to 1000μs. This indicates the high periodic jitter peaks were being caused either by workqueue kernel threads or by some interrupt. To find out the answer to this question, we looked at the kernel data for all configurations that was collected while the trace collector benchmark was running.

4.1.2. Kernel data analysis. Using the methodology described in our earlier work [2] we created a ranked list of sources of jitter for each configuration. We considered only the processes and interrupts that occur on the primary SMT thread to be a source of jitter. Sources of jitter for all configurations are given in Table

	Jitter Source	Lowest Interruption (us)	Highest Interruption (us)	Frequency	Mean (us)	Std Dev (us)	Total Jitter (%)	Total Jitter (us)
Baseline SMTON	timer	3.18	3459.55	167226	3.84	10.61	94.74	642829.48
	events1	2.21	145.06	3032	5.1	10.44	2.28	15449.56
	rtasd	3.69	67.62	198	44.16	4.75	1.29	8744.2
	ipr	14.91	58.94	143	27.33	11.16	0.58	3908.33
	watchdog	1.71	6.59	1372	2.34	0.26	0.47	3204
	bash	2050.12	2087.35	1	2050.12	0	0.3	2050.12
	migratio	3.69	2087.35	3	688.7	1179.03	0.3	2066.1
	IPI	4.93	9.48	46	6.19	0.75	0.04	284.53
OS Prio	timer	3.11	3459.96	173560	3.78	10.48	98.85	655886.55
	ipr	8.39	82.91	169	28.23	12.3	0.72	4771.12
	watchdog	1.37	2.89	1386	2.03	0.23	0.42	2819.13
	eth0	18.38	18.6	1	18.6	0	0	18.6
	IPI	6.23	7.61	1	7	0	0	7
H/W Prio	timer	2.85	3446.37	172664	3.45	10.55	95.27	594919.4
	events1	1.68	58.1	3040	4.65	6.65	2.26	14125.76
	rtasd	1.83	55.52	200	41.79	7.7	1.34	8358.5
	watchdog	1.55	3.98	1385	2.19	0.2	0.49	3031.01
	bash	2109	2126.23	1	2109	0	0.34	2109
	ipr	12.32	48.85	54	27.64	10.82	0.24	1492.41
	IPI	5.37	41.57	48	8.74	7.56	0.07	419.31
	kblockd	3.86	5.85	2	4.84	1.39	0	9.69
	migratio	4.37	4.38	1	4.37	0	0	4.37
Isolated CPU	***timer***	2.82	***31.56***	173823	3.08	***0.25***	97.46	536166.15
	rtasd	4.93	54.63	397	24.5	18.15	1.77	9725.47
	watchdog	1.58	2.65	1387	1.93	0.1	0.49	2676.52
	migratio	4.93	10.2	199	6.47	0.67	0.23	1288.35
	IPI	5.49	8.2	46	6.52	0.54	0.05	299.97

Table 1. *Sources of jitter for different configurations on a 1-Processor 2-Way SMT Power5 node*

1.

Kernel data clearly shows the effect of various configurations. The Baseline SMTON and Hardware Priority configurations have nearly identical list of sources of jitter. This is expected as the operating system is unaware of the modified hardware thread priorities and continues to schedule processes and interrupts as usual. The OS Priority configuration reduces the list of sources of jitter with respect to the Baseline SMTON configuration. It can be seen that it is the timer interrupt that causes the highest peaks of around 3500μs in the 3 configurations - Baseline SMTON, OS Priority and Hardware Priority. In all these 3 configurations, jitter caused by the timer interrupt has a wide range from 2.8μs to 3460μs. This is also reflected in a high standard deviation of close to 10.5 μs.

The Isolated CPU configuration corrects this behavior by removing the really high periodic peaks around 3500μs and reducing the range of jitter caused by timer interrupts to 2.8-32μs. The standard deviation also comes down drastically to 0.25μs. As described in Section 2, the Isolated CPU configuration removes the workqueue kernel threads as well as external interrupts and softirqs (or bottom halves) from primary SMT threads. While timer interrupt top halves continue to execute on primary SMT threads, it is evident that most of the large jitter due to timer interrupts was caused by large amount of processing in the bottom halves or the softirqs. Isolated CPU configuration outperforms other configurations by moving these high jitter softirqs to secondary SMT threads.

Note that our kernel instrumentation provides data only about process scheduler and interrupt handling. There are still some jitter peaks in the 100-1000μs region that remain unidentified. There are 450 such events out of the total 1 million events (2.5% of total jitter). We expect these jitter events to have been caused by the Power Hypervisor.

4.1.3. Trace driven simulation. In order to quantify this effect of reduction in jitter, we performed a simulation using the collected jitter traces. A parallel application with a compute loop of 1 ms is simulated. Simulator executes around 300K iterations of the compute loop. Number of tasks are increased from 64 till 16K. In this configuration, each task runs on a single processor node and hence the number of tasks are equivalent to the number of OS images. The results from simulation are given in Figure 4. The percentage slowdown in the average case is reduced from 44.2% in the Baseline

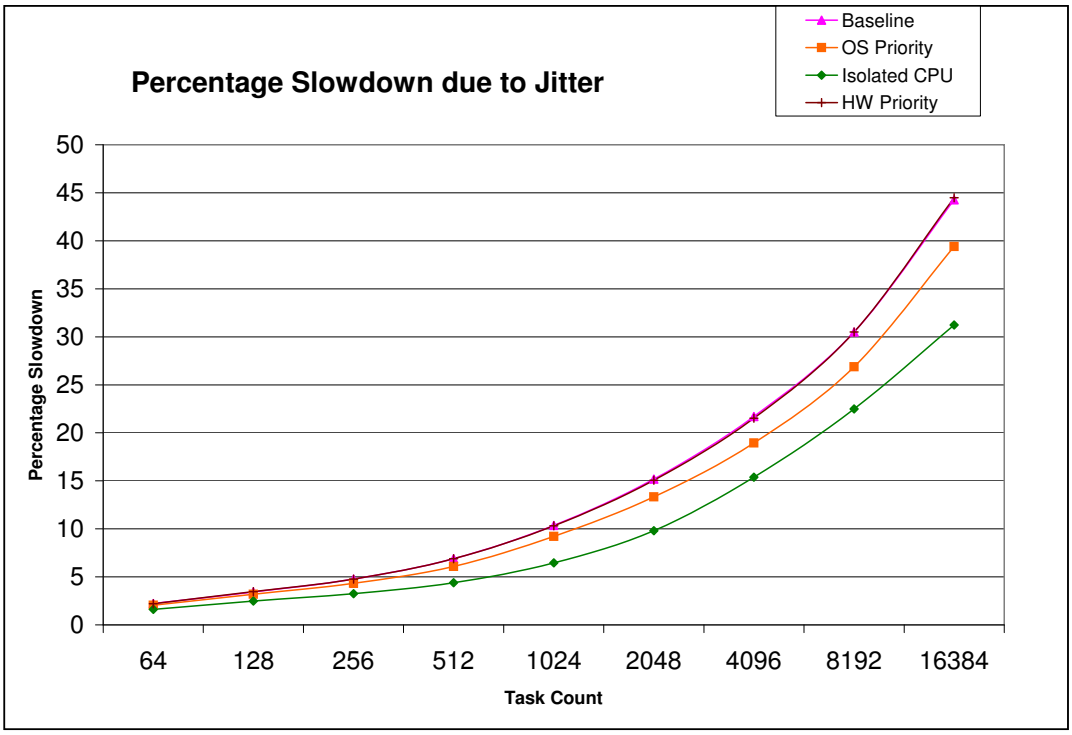

Figure 4. Simulated percentage slowdown due to jitter

SMTON configuration to about 31.2% in the Isolated CPU configuration at 16K tasks - a 30% reduction in slowdown.

4.2. Experiments on 8 32-processor 4.7 GHz 2-way SMT POWER6 Nodes

Our second set of experiments was done on 8 node cluster with each node comprising of 32, 2 way SMT, POWER6 processors running RedHat Enterprise Linux 5.2 with kernel version 2.6.27-rc5.

4.2.1. Jitter trace collection on a single node. Here again, we first collected a jitter trace for all 32 cores on a single node. In these experiments, a trace collector instance ran on each core and collected approximately half a million jitter events. This takes roughly 23 minutes on the systems we used.

The jitter traces for the four configurations are shown in Figure 5, where y-axis represents jitter duration in μs and x-axis represents jitter samples for all the 32 processors (processor number represents the trace collector task id). The first striking difference from the earlier set of experiments is the absence of really high periodic peaks in the Baseline SMTON configuration. All configurations show most jitter to be around 100 μs and some jitter events in the region 100-200 μs and a couple of jitter peaks around the 500-550 μs region. Baseline SMTON configuration has 4 jitter peaks around the 500 μs region, whereas OS Priority configuration reduces the number of these large peaks to 2. Isolated CPU and Hardware Priority configurations each reduce the number of large peaks around 500μs to 1. Note that the difference between various configurations is less stark here as compared to the uniprocessor runs.

4.2.2. Parallel benchmark runs on a real cluster. Since we had access to a 8 node (256 cores) cluster with identical Power6 nodes, we ran the parallel benchmark for this set of experiments. The nodes were connected via Infiniband. As described earlier, the parallel benchmark is a typical parallel application that executes repeated compute bound loops of 1 ms - each followed by a barrier. We could not do an OS Priority run as boosting the OS priority of the parallel benchmark to 80 with SCHED_RR policy seemed to halt the progress of the program and it could never finish. It is likely that some of the essential kernel threads (probably those related to Infiniband) are not getting a chance to run. We are still investigating the exact cause for this behavior.

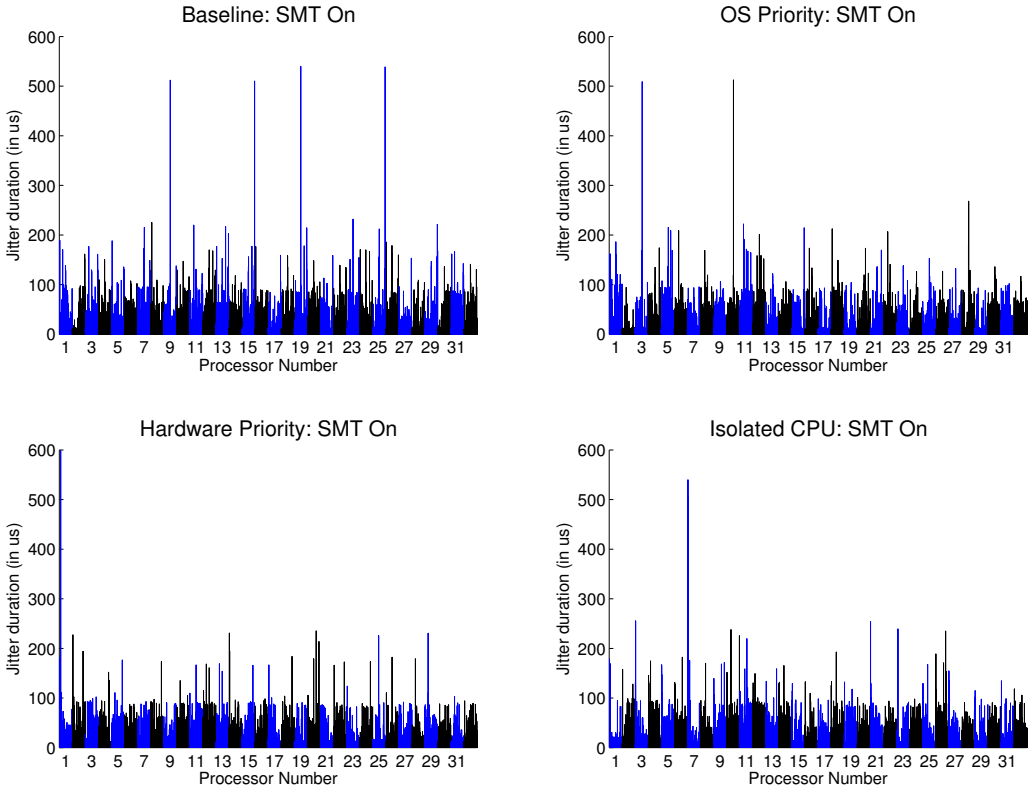

Figure 5. Jitter timeseries for different configurations on a 32-Processor 2-Way SMT Power6 node

In place of the OS priority configuration, we evaluated a "Leave 1 CPU Idle" configuration, in which 1 physical CPU is left idle so that all system jitter can be handled by that CPU. This technique of leaving one of the CPUs of a n-CPU SMP machine idle has been used by HPC systems to handle jitter and it has proved to be effective [1]. As 1 CPU is dedicated for handling all jitter, we switch off secondary SMT threads on all the remaining CPUs to reduce any SMT interference. This makes it different from all other configurations which have the secondary SMT thread switched on. This also helps us match the configurations that have been traditionally tried to reduce jitter. For this experiment, we left 1 CPU idle on each node (248 cores).

We collected the maximum compute time and maximum total (compute and barrier) time across all 256 cores in each iteration. The parallel benchmark executes roughly half a million compute and barrier iterations. The timeseries plots for maximum compute time and maximum total time for all four configurations - Baseline SMTON, Isolated CPU, Hardware Priority and Leave 1 CPU Idle are shown in Figure 6 and Figure 7, respectively. X-axis represents time and y-axis represents the maximum compute time or maximum total time in milliseconds.

The maximum compute time plot (Figure 6) shows that both the Baseline SMTON and Isolated CPU configurations exhibit some periodic high peaks which are not visible in the Hardware Priority configuration. The maximum peak in Baseline SMTON is around 1560μs, in Isolated CPU configurations it is around 1360 μs and in the Hardware priority configuration it is around 2500 μs. These numbers match with the single node jitter traces obtained and plotted above - as the maximum compute time equals jitter free compute time (1 ms or 1000 μs) + maximum jitter observed. The "Leave 1 CPU Idle" configuration has exceptionally high peaks with the highest one being around 18.6 ms. This shows that this configuration experiences a lot of jitter even during the compute phase. Analysis of kernel data for this configuration shows the process "pdflush" consuming up to 15 ms. This process is used by Linux to write out data out of its page cache to disk. This indicates that there is some disk writing activity by some system processes that causes "pdflush" to execute. The kernel data for none of the other configurations shows "pdflush" taking this long.

The maximum total time plot (Figure 7) for all configurations except "Leave 1 CPU Idle" is very different from maximum compute time plot. It shows 5 very high peaks in the Baseline SMTON which are in the 40-350 ms range. In the Isolated CPU configuration the

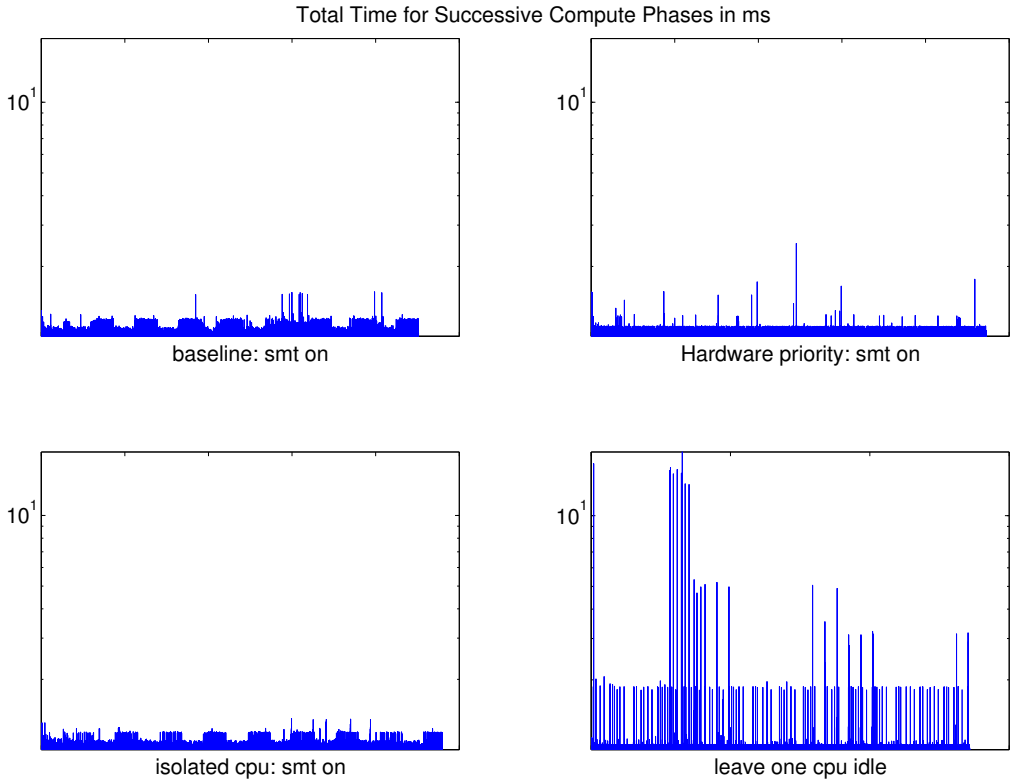

Figure 6. Maximum compute time across all threads in each iteration for a 256 core parallel benchmark run

amplitude of these very high peaks has been reduced to 20-40 ms. The amplitude of periodic high jitter events which were visible in the maximum compute time plot in both these configurations, has gone up. The Hardware Priority configuration, on the other hand does not show very high peaks and it has only one high peak at 6.5 ms. These very high peaks appear only 4 or 5 times out of the half a million iterations executed by the parallel benchmark. Since these very high peaks appear only in the maximum total time plot and not in the maximum compute time plot, it is evident that they are being caused during the barrier phase. This hints to some problem either with the Infiniband fabric or with the Infiniband driver. The fact that the high peaks get reduced in the Hardware priority configuration indicate that it could be due to a faulty Infiniband driver that runs on the secondary SMT thread and causes long delays. When the priority of the secondary SMT thread is reduced, such delays are also reduced. Our initial analysis of the kernel data confirms that these large jitter are not caused by any process or interrupt. Kernel data shows that the idle process ("swapper") runs for equally long duration on the primary SMT thread. This indicates that all nodes just wait for the messages to arrive. We are further investigating this issue.

Leave 1 CPU Idle configuration, which has SMT switched off, has the highest total time peak around 18.8 ms. However, this configuration had the highest compute time peak around 18.6 ms. In this configuration, the high jitter experienced during the compute phase overlaps with the high jitter experienced due to Infiniband issues in the barrier phase.

We also analyzed the average case and worst case performance of the parallel benchmark for the above 4 configurations using both the compute time and total time. For calculating the worst case performance we used the largest 1% of the values for compute time and total time. These results are shown in Figure 8.

It can be observed that Leave 1 CPU Idle configuration performs the worst in all cases due to some disk writing activity that occurs in the compute phase. It also suffers from the disadvantage that there is no secondary SMT thread to handle jitter that occurs due to interrupts or processes that are bound to a given CPU. The average case performance is similar for all the other 3 configurations for both compute time and total time with the Isolated CPU configuration being the best. The worst case performance for total time varies significantly for all 4 configurations. Isolated CPU and Hardware Priority configurations perform much better than the Baseline SMTON in all cases. The difference is largest in the worst case total time performance

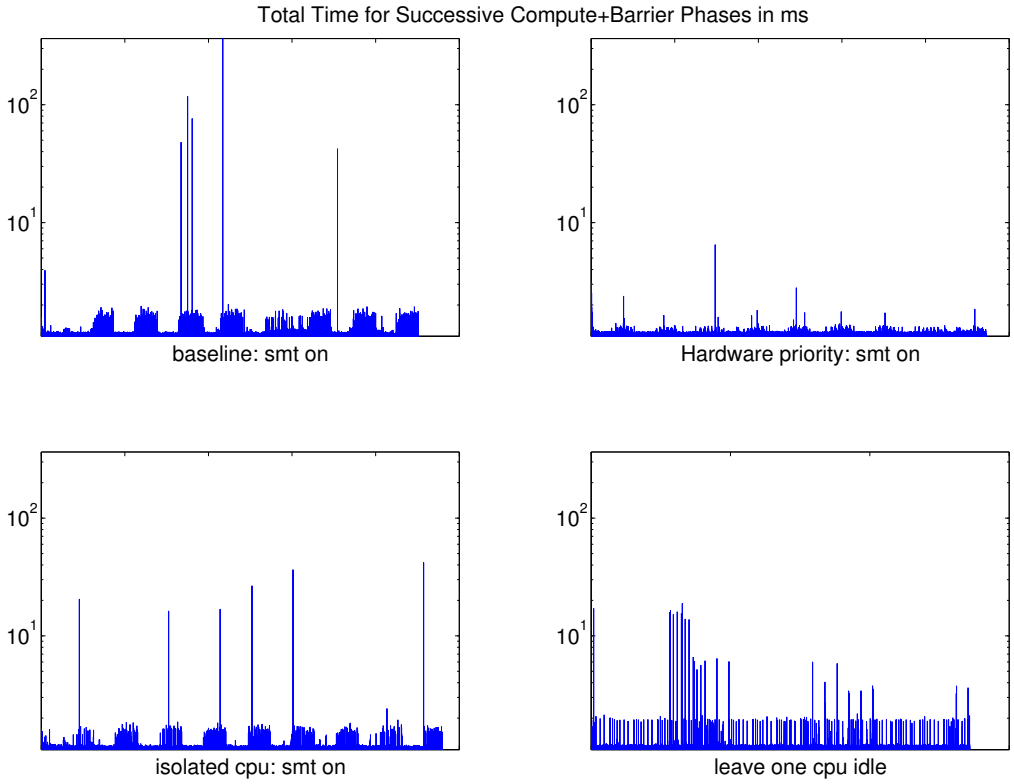

Figure 7. Maximum total time across all threads in each iteration for a 256 core parallel benchmark run

due to the few very high barrier times. Percentage slowdown in the worst case total time for Hardware Priority run is 19%, as compared to 38% in the Baseline SMTON run (50% reduction in worst case slowdown). Percentage slowdown is 24% in the Isolated CPU run (37% reduction in worst case slowdown).

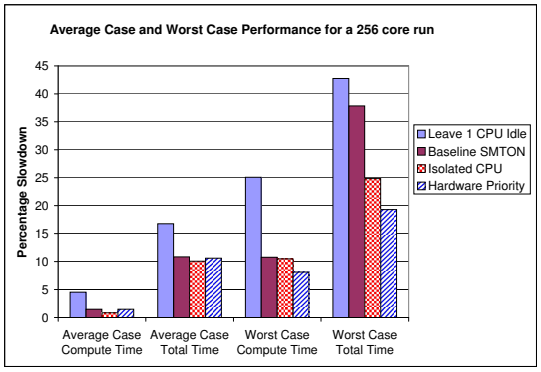

Figure 8. Average case and worst case performance of the parallel benchmark for the 256 core run

5. Related Work

Impact of operating system jitter on parallel program performance and solutions to mitigate this impact have been studied by various researchers. Petrini et al. showed that operating system interference can cause up to 100% performance degradation at 4096 processors in ASCI Q [1]. Beckman et al. observed that only reasonably long jitter (greater than 50μs) occurring frequently enough (every 1 ms) caused any significant performance degradation [12].

Solutions to OS jitter typically consist of a combination of techniques such as synchronization of jitter across nodes (co-scheduling or gang scheduling) [13], [14] and use of microkernels such as the one used on the compute nodes of BlueGene/L [15]. Both these approaches have drawbacks. Synchronization of jitter requires all nodes clocks to be synchronized. This requires either a global switch clock or use of some other software technique such as NTP. Use of microkernel limits widespread use as it requires all existing applications to be ported to those kernels. Another technique that has been widely adopted is leaving out 1 CPU idle out of a n-CPU node [1]. This has proved to be effective earlier in scenarios where most of the jitter was caused by user or kernel processes and not by interrupts and

other kernel activity.

The approach presented in this paper attempts to either remove a source of jitter altogether or reduce its impact by offloading it to an idle SMT thread. Synchronization of jitter can be considered orthogonal to this approach and can still be applied on the residual jitter. Our approach makes use of and builds on several features provided by the Linux operating system and the POWER architecture. Our approach uses several benchmarks proposed in earlier research to measure jitter [2], [10], identify sources of jitter [2] and predict its impact on scalability at large number of processors [9].

6. Conclusions and Future Work

In this paper, we presented a holistic approach that aims to reduce jitter caused by various sources of jitter by utilizing the additional threads or cores in a system. Our approach handled jitter through reduction of kernel threads, intelligent interrupt handling, and switching of hardware SMT thread priorities. This helped in reducing jitter experienced by application threads in the user space, at the kernel level, and at the hardware level. We used several existing features available in the Linux kernel and Power Architecture as well made enhancements to the Linux kernel. Experimental validation of the overall approach indicated up to 30% reduction in average case slowdown at 16K OS images for a simulated parallel application and up to 50% reduction in worst case slowdown at 8 OS images for a real application.

We are currently looking at combining the solutions presented in this paper into one single unified approach. From our experimentation, we infer that isolating a CPU along with modifying the hardware priority of the SMT threads should give us the best gains. However, this presents a trade-off which need to be evaluated carefully. Routing of interrupts to the secondary SMT thread assumes that they can be executed with some level of guarantee on the secondary SMT thread. Reducing the hardware priority of the secondary SMT thread, on the other hand, slows down the code running on that SMT thread considerably and this can have severe performance impact in case of essential interrupts. For example, a parallel application that executes barriers will observe a huge drop in its network performance as all networking interrupts get routed to the throttled secondary SMT thread. We are currently working on a solution to this problem. We are also working on evaluating our approach using real applications.

7. Acknowledgements

We would like to thank Nishanth Aravamudan from IBM Linux Technology Center (Beaverton) , IBM HPC performance team at Poughkeepsie and Liana Fong and Seetharami Seelam from IBM T.J. Watson Research Center for their useful inputs and discussions. We would also like to thank Bill Buros and Peter Wong from IBM Linux Technology Center, Austin for providing us access to a real cluster.

References

[1] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI Q," in *ACM Supercomputing*, 2003.

[2] P. De, R. Kothari, and V. Mann, "Identifying Sources of Operating System Jitter Through Fine-Grained Kernel Instrumentation," in *IEEE Cluster*, 2007.

[3] *Understanding the Linux Kernel, 3rd Ed.* Oreilly, 2005.

[4] "Setting realtime scheduler priorities." [Online]. Available: http://rt.et.redhat.com/page/RHEL-RT_SchedPrioHowto

[5] "sched_setscheduler - Linux man page." [Online]. Available: http://linux.die.net/man/2/sched_setscheduler

[6] "Isolated CPUs on Linux Kernel Mailing List (LKML)." [Online]. Available: http://lkml.org/lkml/2008/2/5/492

[7] J. Kihm, A. Settle, A. Janiszewski, and D. Connors, "Understanding the Impact of Inter-Thread Cache Interference on ILP in Modern SMT Processors," *Journal of Instruction-Level Parallelism*, vol. 7, Jun 2005.

[8] "IBM Redbook: PowerVM Virtualization on IBM System p: Introduction and Configuration Fourth Edition." [Online]. Available: http://www.redbooks.ibm.com/abstracts/sg247940.html

[9] P. De, R. Kothari, and V. Mann, "A Trace-driven Emulation Framework to Predict Scalability of Large Clusters in Presence of OS Jitter," in *IEEE Cluster*, 2008.

[10] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "ACM SIGOPS Operating Systems Review," in *Operating System Issues for Petascale Systems*, 2006.

[11] P. De and R. Garg, "The Impact of Noise on the Scaling of Collectives: An Empirical Evaluation," in *High Performance Computing (HiPC)*, 2006.

[12] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the Effects of Operating System Interference on Extreme-Scale Parallel Machines," Tech. Rep., Jan 2007.

[13] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *ACM Supercomputing*, 2003.

[14] P. Terry, A. Shan, and P. Huttunen, "Improving application performance on HPC systems with process synchronization," *Linux Journal*, no. 127, pp. 68–73, Nov 2004.

[15] T. B. G. Team, "An Overview of the Blue Gene/L Supercomputer," in *ACM Supercomputing*, 2002.