jitSim: A Simulator for Predicting Scalability of Parallel Applications in Presence of OS Jitter

Pradipta De and Vijay Mann

IBM Research - India, New Delhi

Abstract. Traditionally, Operating system jitter has been a source of performance degradation for parallel applications running on large number of processors. While some large scale HPC systems such as Blue Gene/L and Cray XT4, mitigate jitter by making use of a specialized light-weight operating system on compute nodes, other clusters have attempted using HPC-ready commodity operating systems such as ZeptoOS (based on Linux). However, as large systems continue to be designed to work with commodity OSes, OS jitter still remains an active area of research within the HPC community. While, it is true that some of the specialized commodity OSes like ZeptoOS have relatively low OS jitter levels, there is still a need to have a quick and easy set of tools that can predict the impact of OS jitter at a given configuration and processor number. Such tools are also required to validate and compare any new techniques or OS enhancements that mitigate jitter. Emulating jitter on a large "jitter-free" platform using either synthetic jitter or real traces from commodity OSes has been proposed as one useful mechanism to study scalability behavior under the presence of jitter. However, this requires access to large scale jitter free systems, which are few in number and not so easily accessible. As new systems are built, that should scale up to a million tasks and more, the emulation approach is still limited by the largest jitter free system available. In this paper we present jitSim - a simulation framework for predicting scalability of parallel compute intensive applications in presence of OS jitter using trace driven simulation. The jitter simulation framework can be used to quickly simulate the effects of jitter that is characteristic of a given OS using a given trace. Furthermore, this system can be used to predict scalability up to any arbitrarily large number of task counts. Our methodology comprises of collection of real jitter traces, measurement of network latency, message passing stack latency, and shared memory latency. The simulation framework takes the above as inputs and then simulates multiple parallel tasks starting at randomly chosen points in the jitter trace and executing a compute phase. We validate the simulation results by comparing it with real data and demonstrate the efficacy of the simulation framework by evaluating various jitter mitigation techniques through simulation.

1 Introduction

Operating system jitter (or OS jitter) refers to the interference experienced by an application due to scheduling of background daemon processes and handling

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 117–130, 2010.

of asynchronous events such as interrupts. It has been shown that parallel applications on large clusters suffer considerable degradation in performance due to OS jitter [1,2].

Techniques to mitigate jitter fall broadly under four categories: use of microkernels, OS and hardware level tuning techniques [3], synchronization of jitter [4,5] and enhancements to popular commodity OSes [6]. Several large scale HPC systems, like the Blue Gene/L [7] and Cray XT4 [8], make use of a customized light-weight microkernel at the compute nodes to mitigate OS jitter. These customized kernels typically do not support general purpose multitasking and may not even support interrupts. However, these systems require applications to be modified or ported to their respective platforms. Other systems like the ASCI Purple and the JS21 system at the Barcelona Supercomputing Center [9], which make use of commodity OSes (AIX and RedHat Enterprise Linux respectively), still suffer from OS jitter [10]. These systems rely mainly on OS and hardware level tuning and/or synchronization of jitter across nodes. Synchronization of jitter across all nodes can yield moderate (close to 50% [4]) to very high (close to 300% [5]) performance improvements. Simultaneous multi threaded (SMT) and hyperthreaded processors can also help in mitigating jitter, even though they may have other performance implications. The ZeptoOS project [6] from Argonne National Laboratory has been working on making Linux HPC-ready on compute nodes as well as I/O nodes.

While, it is true that some of the specialized commodity OSes like ZeptoOS have relatively low OS jitter levels, OS jitter still remains an active area of research within the HPC community as researchers continue to explore application sensitivity to jitter levels in their clusters [11]. As large systems continue to be designed to work with commodity OSes [12,13,6], there is a need to have a quick and easy set of tools that can predict the impact of OS jitter at a given configuration and number of processors. Such tools are also required to validate and compare any new techniques or OS enhancements that mitigate jitter. As the effectiveness of any technique to mitigate jitter can only be evaluated in a large cluster with thousands of nodes, one of the biggest hindrances in the development and evaluation of new techniques for handling jitter is that there are a few large clusters running commodity OSes worldwide, which are often unavailable for experimental and validation purposes.

Emulating jitter on a large "jitter-free" platform using either synthetic jitter or real traces from commodity OSes has been proposed as useful mechanism to study scalability behavior under the presence of jitter [11,14,15]. Beckman et al. [15] used a single node benchmark to measure jitter and injected synthetic jitter of varying length and periodicity on a jitter-less platform such as Blue Gene/L to study its impact on scalability of various collective operations. They used purely synthetic jitter rather than collecting traces from real Linux systems. In an earlier work, we improved upon their emulation technique to ensure precise emulation of jitter [14]. Ferreira et al. [11], in their Supercomputing 2008 (SC 2008) paper, described the effect of different kinds of kernel-generated noise on application performance at scale.

All the above approaches to predict system performance require an accurate methodology for precisely emulating jitter. Emulation of jitter on large "jitter free" platforms requires access to large scale jitter free systems. As new systems are built that should scale up to a million tasks and more, the emulation approach is still limited by the largest jitter free system available. In this paper, we present the design and implementation of jitSim - a simulation framework for predicting scalability of parallel compute intensive applications in presence of OS jitter using trace driven simulation. The jitter simulation framework can be used to quickly simulate the effects of jitter that is characteristic of a given OS using a given trace. Furthermore, this system can be used to predict scalability up to any arbitrarily large number of task counts. The methodology is based on collection of real jitter traces, measurement of network, message passing stack, and shared memory latencies and simulation of multiple parallel tasks starting at randomly chosen points in the trace that use the cycles between jitter events in the trace as the available compute cycles to reach a given number of compute phase cycles. We validate the simulation results by comparing it with real data and demonstrate the efficacy of the simulation framework by evaluating various jitter mitigation techniques through simulation.

The rest of this paper is organized as follows. Section 2 describes the challenges involved in simulating OS jitter and gives an overview of our approach. Section 3 gives the details of the jitter simulator framework. Section 4 presents our results. We present an overview of related research in section 5 and finally conclude in section 6.

2 Methodology

In this section we first describe the challenges involved in simulating the effects of OS jitter. We then give an overview of our methodology.

2.1 Challenges

Any methodology for predicting scalability of parallel applications in presence of OS jitter using trace driven simulation faces the following challenges:

- Collection of a trace that is representative of OS Jitter on a real system
- Simulation of the synchronization step in a parallel application
  - Requires knowledge of network topology, system architecture, number of cores on a single chip, number of chips in a node
  - Requires measurement of one hop network latency, latency in the message passing stack, shared memory access latency
  - Requires knowledge of the synchronization algorithm in the message passing layer (e.g. a binary tree, a k-ary tree, an octet tree, etc)
- Different types of applications are likely to be affected differently by OS jitter and will therefore, have a different scaling behavior.

In this paper, we present a methodology that is aware of the above challenges and overcomes them. We collect jitter traces using a microbenchmark that ensures that it does not introduce any additional jitter of its own. We collect information about one hop network latency, latency in the message passing stack and shared memory access latency through various micorexperiments. This work assumes the parallel application to be a compute intensive application. For a real application to fit into this model, we require that the application be modeled as a sequence of compute, communication and jitter phases. The jitter phases will comprise of various application induced latencies such as memory access latency or I/O latency.

2.2 Overall Approach

Our overall approach comprises of the following steps:

1. A large jitter trace is collected from a single core, or a set of cores. The jitter trace is collected using a timestamp reader benchmark, which is explained later.
2. one hop network latency, MPI stack latency, shared memory access latency using SEND/RECV messages of varying sizes between MPI tasks running on same nodes and those running on the same node (but different cores) are measured.
3. Different portions of the trace can be thought of as multiple traces collected on different nodes - can be used to model jitter experienced by multiple tasks.
4. Different MPI tasks start their simulation from an index at any random point in the trace (for baseline) or at any random point that represents the start of a high priority window (for co-scheduler) or at the same randomly chosen point (for perfectly synchronized jitter).
5. Cycles between two jitter events in trace are used as available compute cycles and the total cycles consumed by each task (which include jitter cycles and compute cycles) are calculated to complete a compute phase.
6. A synchronization point is simulated across all tasks at the end of a compute phase by passing send and receive messages (using the latencies measured earlier) across the tasks arranged in a tree.
7. A slowdown is calculated by by comparing the total consumed cycles to the compute cycles.

These steps are described in more detail in the sections below.

3 Implementation Details

In this section we describe in detail the steps involved in our simulation methodology.

3.1 Collection of a Jitter Trace

We use a single node timestamp register reader benchmark (which we refer to as the TraceCollector) for collecting a jitter trace. The TraceCollector is run on a single node that is running the operating system with the specific configuration under which we want to predict the cluster scalability. The TraceCollector has a tight loop that reads the timestamp counter register repeatedly and finds out the timestamp deltas between successive readings. It then compares each timestamp delta with the minimum timestamp delta (tmin) observed on that platform to decide whether that timestamp delta constitutes a jitter instance or not. This is used to create a jitter distribution as well as a jitter trace. Each core runs a unique instance of TraceCollector and a trace is generated for each core. The jitter trace is a tuple of the form: $< jitter\_duration,\ cycles\_to\_next\_jitter >$. More details about trace collection can be found in [16].

3.2 Measurement of Message Passing Latencies

When parallel tasks run on the same physical node in a system, they typically communicate over shared memory whereas when they run on different physical nodes, they communicate over the network using the high speed interconnect. In order to simulate a tree based synchronization point (for example, a barrier) across tasks, these latencies need to be measured. Three experiments are conducted to estimate one hop, network latency, MPI Stack Latency and Shared Memory Latency. In each experiment, two MPI tasks are run either on two cores of the same node (for measuring shared memory access latency or the MPI stack latency) or on two different physical nodes (for measuring one hop network latency) and MPI SEND and RECV messages of varying sizes are passed between these two tasks and finally an average is calculated. These experiments are shown in Figure 1. MPI Stack latency is divided by half to get an estimate of SEND and RECV latencies. MPI tasks are allocated sequentially among the cores followed by nodes.

3.3 Simulating Multiple Parallel Tasks Using a Single Jitter Trace

Different portions of the trace can be thought of as multiple traces collected on different nodes and hence can be used to model jitter experienced by multiple tasks.

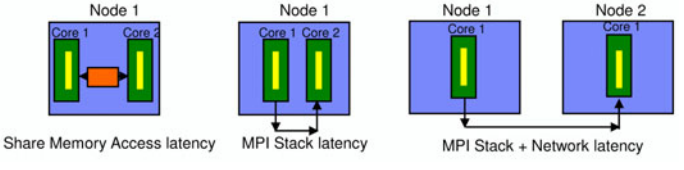

Fig. 1. Measurement of communication latencies

Choosing the point in the jitter trace from where the different tasks start executing is an important decision and it can have interesting ramifications. In a cluster that has unsynchronized jitter, different kinds of jitter activities will hit each node at different points in time. On the other hand, in a cluster that has employed a mechanism for synchronizing jitter across all nodes, jitter activities will hit each node at the same time. In order to emulate the unsynchronized jitter scenario, simulation starts different tasks at different randomly chosen points in the jitter trace. To simulate synchronized jitter, simulation starts different tasks from the same randomly chosen point in the jitter trace. To simulate co-scheduled jitter [5], where a user level co-scheduler daemon puts the favored process into high priority for a given time window (referred to as high priority window) and then into low priority for a short time window (referred to as low priority window), the different parallel tasks start at any random point in the trace that represents the start of a high priority window. This is explained in detail with the help of an example in section 4.

3.4 Simulating Effects of Jitter on a Parallel Application

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 time work quanta into the target compute cycles in each phase by multiplying it by the frequency value. Cycles between two jitter events in trace are used as available compute cycles that can be used by the parallel task. At each task, it keeps adding the available compute cycles till a jitter value is reached or the target compute cycles for a compute phase

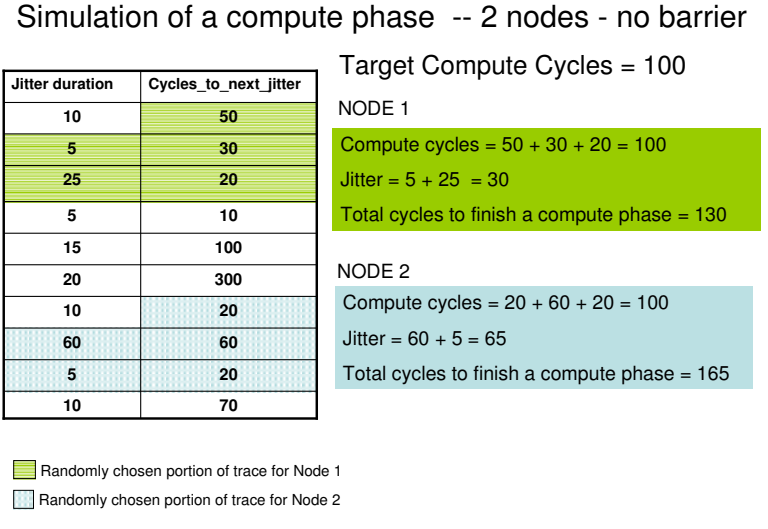

Fig. 2. Simulation of a compute phase with 2 parallel tasks

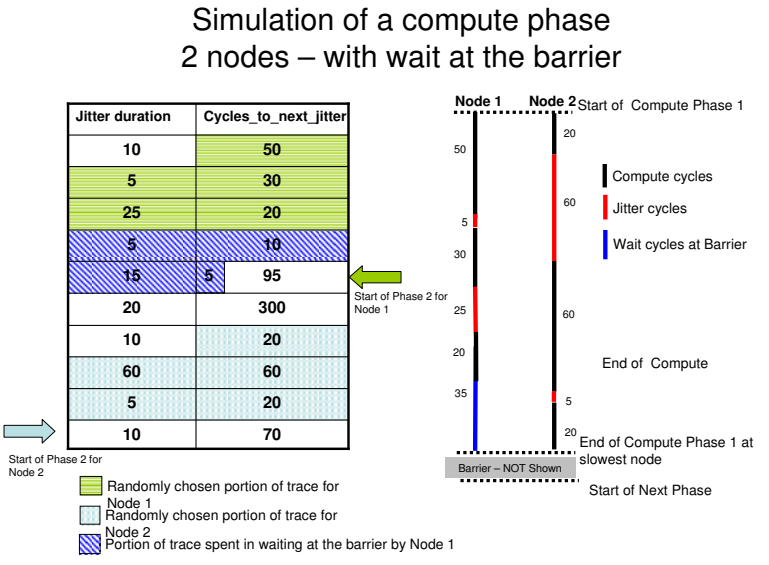

Fig. 3. Simulation of a compute phase with 2 parallel tasks showing wait at the barrier

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. This is shown in Figure 2.

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. This is shown in Figure 3.

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.

3.5 Simulating Effects of Co-scheduled or Synchronized Jitter on a Parallel Application

It has been shown that synchronization of jitter across all nodes can yield moderate (close to 50% [4]) to very high (close to 300% [5]) performance improvements. Jitter can be synchronized across nodes using a coscheduler. A co scheduler makes use of a user level co scheduler daemon puts the favored process into high priority for a given time window (referred to as high priority window) and then into low priority for a short time window (referred to as low priority window). The start of these windows is synchronized across nodes by updating

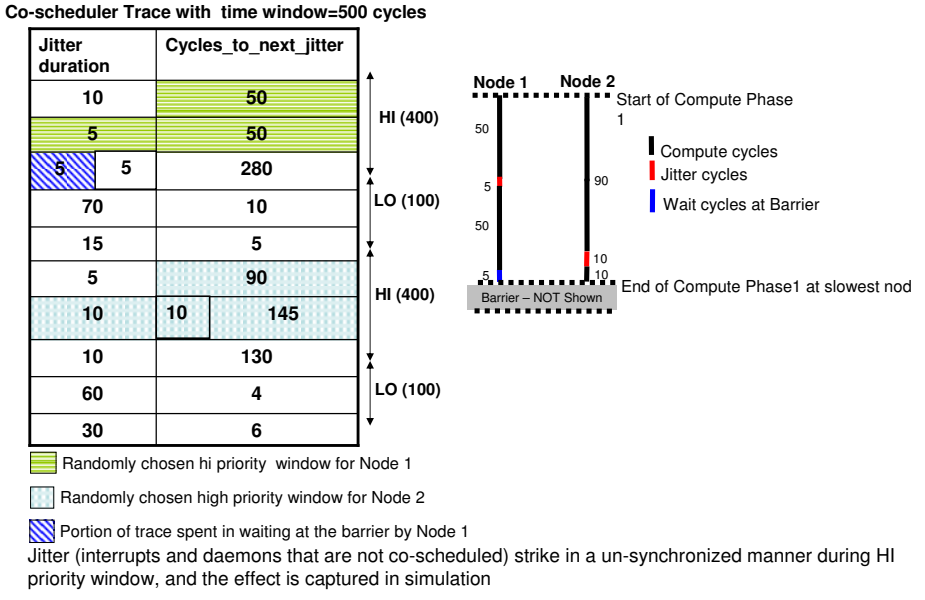

Fig. 4. simulation phase 1: Simulation of co-scheduled jitter on two nodes

the length of the low priority window according to the clock skew. This effectively synchronizes all jitter across the different nodes as all jitter now occurs during the low priority window. To simulate co-scheduled jitter, the different parallel tasks start at any random point in the trace that represents the start of a high priority window.

The overall simulation of co scheduled jitter for two nodes is shown in Fig 4 and Fig 5.

3.6 Simulating Effect of Jitter at Synchronization Phase(Barrier)

The above discussion does not mention the effect of OS jitter due to communication once the slowest task reaches a synchronization point such as barrier. Usually, a synchronization point for Messaging Passing Interface (MPI) applications is implemented by arranging the tasks in a tree like structure. This ensures that each task only communicates with its neighbors in the tree while performing a synchronization operation.

The barrier operation comprises of two stages: a pre-barrier stage succeeded by a post-barrier stage. We assume that these stages are implemented using message passing along a complete binary tree, as shown in Figure 6. The basic structure does not change for any implementation based on a fixed tree of bounded degree, such as a k-ary tree. A process is associated to each node of the binary tree. A special process, called the root (Task 1 in Figure 6) initiates the post-barrier stage and concludes the pre-barrier stage. The pre-barrier stage is entered at

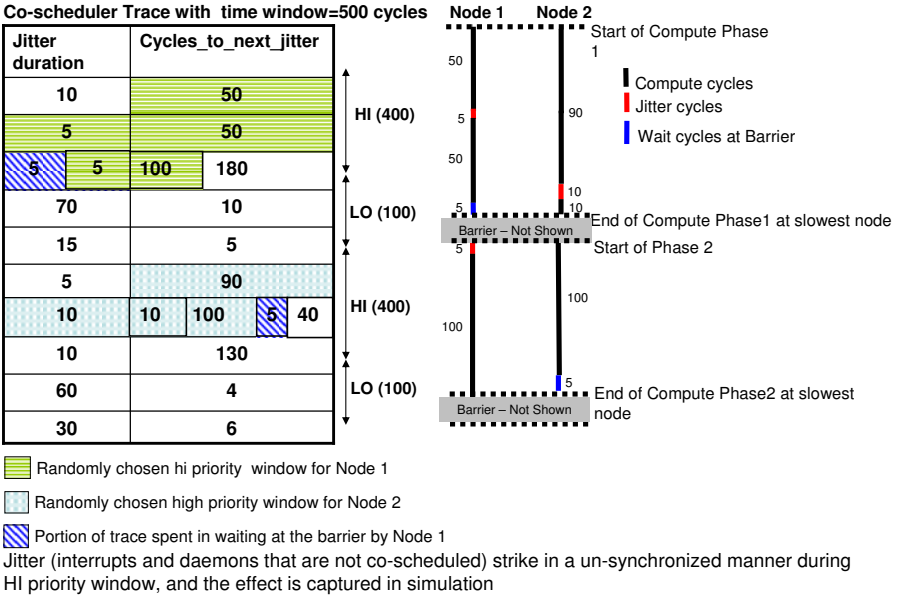

Fig. 5. simulation phase 2: Simulation of co-scheduled jitter on two nodes

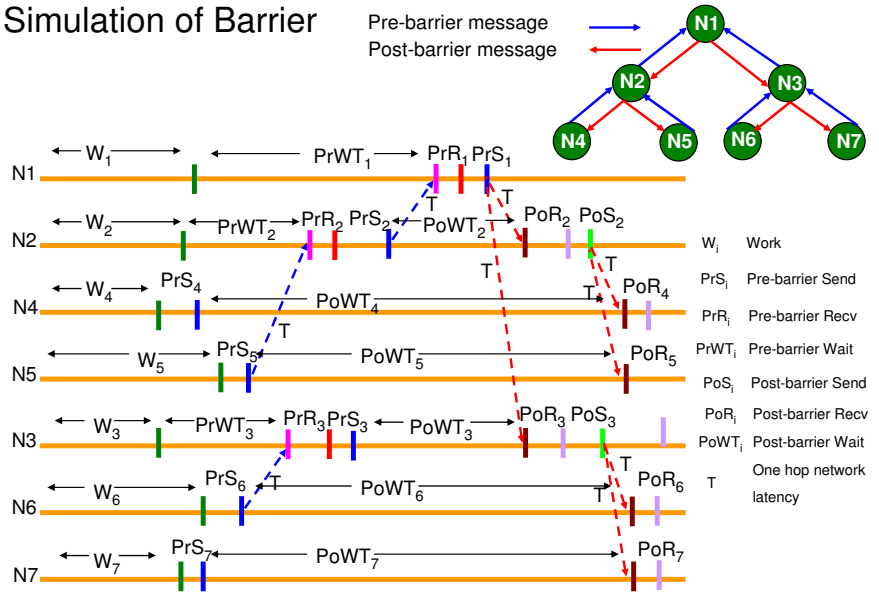

Fig. 6. Simulation of barrier for 7 tasks arranged in a complete binary tree

the end of computation phase, when each task notifies its parents of completion of work. This stage ends when the root finishes its computation and receives a message from both its children indicating the same. This is followed by a post-barrier stage, when a start-compute-phase message initiated by root is percolated to all leaf nodes. A more complete description of the communication phase can be found in [17].

Every time a message is sent from a task to its parent, or receives a message from its children, certain number of cycles are allocated to that operation (and counted as part of the compute cycles). These latencies are measured from the experiments mentioned earlier. As the simulator traverses the trace, it looks for the given number of send or receive cycles. If it encounters a jitter event during this time, the jitter cycles are added to the total cycles consumed. Simulation of a barrier for 7 tasks arranged in a binary tree is shown in Figure 6.

4 Experimental Results

We conducted 3 main sets of experiments to prove the effectiveness of our jitter simulation methodology.

1. Validation of simulation results against real data
2. Effect of length of trace on simulation
3. Use of simulation results to evaluate various jitter mitigation techniques

All results in this section assume a 1 to 1 correspondence between Processors and MPI tasks. For example, a result at 16K Processor implies that there are 16K MPI tasks that are running on 16K Processors, each with its own OS image.

4.1 Comparing Simulator Results with Real Runs

A comparison of scalability prediction from simulation with real runs on Linux running in run level 3 in the baseline (unsynchronized jitter) version is given in Figure 7. The real runs were done on a cluster with 13 identical nodes and each node had 32 Power6 cores running SLES 10. For simulation, a trace was collected from each of the 32 cores on a node. Scalability prediction from simulation matches the real results closely. In this case, simulation used only the average maximum compute times (i.e. the compute phase and the wait after the compute phase due to synchronization point) and did not include the slowdown in the communication phase.

4.2 Effect of Trace Coverage by Each Process during Simulation

One of the critical factors that controls the accuracy of the simulation process is the overall length of the trace used for simulation and what percentage of it gets covered by each MPI task. We found that a jitter trace of about an hour was sufficient to capture most periodic daemons. The percentage of the trace covered by each MPI task is a more difficult parameter to decide. If the

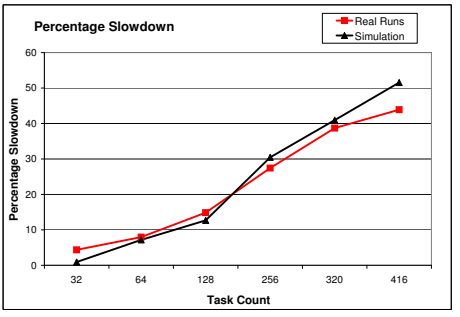

Fig. 7. Comparison of real runs with simulation results

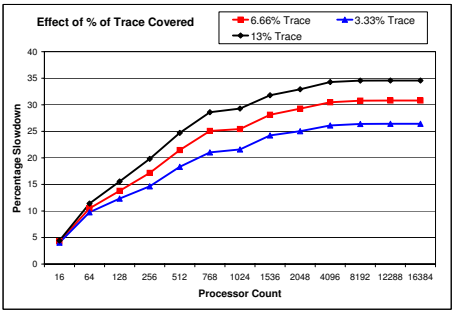

Fig. 8. Effect of length of trace covered by each process during simulation

percentage is too high, all MPI tasks will end up observing similar jitter and hence the overall slowdown will not change as you add more MPI tasks to the simulation. If the percentage is too low, then the simulation results are in the danger of "not being representative enough". We conducted an experiment by varying the percentage of the trace covered by each MPI task. The simulation is done using traces collected from a node with a single Power 5, 1.5 GHz processor running Fedora Core 6, kernel version 2.6.24 (with the CFS scheduler) and with the timer interrupt interval set to 10 ms (100 Hz).The results are shown in Figure 8. While the percentage slowdown keeps increasing as the percentage of trace covered is increased, the overall trend remains the same. More importantly, the number of processors at which the trend flattens (the percentage slowdown becomes constant) is almost the same (4096 processors).

4.3 Comparing Jitter Mitigation Techniques Using Simulation

In this section, we show the effectiveness of the simulation methodology by evaluating various jitter mitigation techniques.

Unsynchronized jitter vs synchronized (co-scheduled) jitter: We first compare unsynchronized (baseline) and synchronized (co-scheduled) jitter. The simulation is done using the traces collected above in section 4.2

Simulation results are shown in Figure 9. It can be observed that at lower processor counts (till about 512 processors) the synchronized jitter performs poorly as compared to the Baseline configuration due to the high jitter introduced by the co-scheduler user daemon and ntpdate. It starts providing performance benefits only after 512 processors and then the slowdown flattens at around 4K processor to a level of 10.8% slowdown and remains the same till about 16K processor counts . The Baseline curve continues to increase without showing any signs of a plateau effect.

OS and hardware tuning techniques to mitigate jitter: In order to demonstrate the effectiveness of the simulation methodology, we reproduce here,

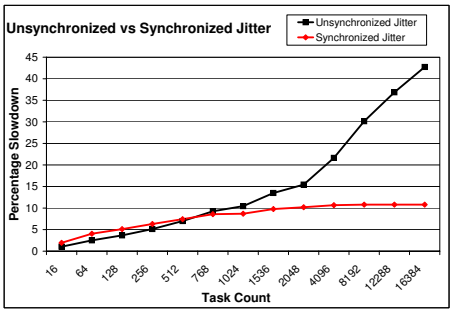

Fig. 9. Using simulation to compare unsynchronized (baseline) and synchronized (co-scheduled) jitter

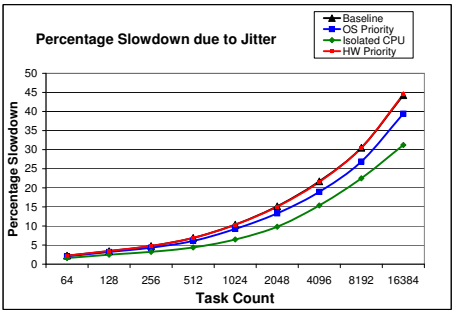

Fig. 10. Using simulation to compare OS and hardware level tuning techniques

a result from one of our earlier works that explored various OS and hardware level tuning techniques to mitigate jitter [3]. The simulation was done using traces collected from a node with a single Power 5, 1.5 GHz processor running Fedora Core 6, kernel version 2.6.24 (with the CFS scheduler) in SMT-ON mode (Baseline). We compared the effectiveness of following techniques compared to the baseline configuration:

- Boosting the OS scheduling priority - SCHED_RR 80 (OS Priority)
- Isolating the primary SMT thread along with IRQ rerouting (Isolated CPU)
- Boosting primary SMT thread hardware priority to 6, while lowering the other thread's priority to 1 (HW Priority)

The results from simulation are given in Figure 10. The percentage slowdown in the average case is reduced from 44.2% in the Baseline SMTON configuration to about 31.2% in the Isolated CPU configuration at 16K tasks - a 30% reduction in slowdown. This result indicates that most of the jitter in these configurations came from interrupts and some from user level and kernel processes. While boosting the OS scheduling priority reduced the jitter due to user level and kernel processes, it did nothing to the jitter due to interrupts. Isolating the primary SMT thread along with IRQ rerouting (Isolated CPU) handled both these sources of jitter and hence performed the best. Boosting the hardware SMT thread priority had limited effect as both the sources of jitter kept getting scheduled on the primary thread. It is interesting to note that the difference between various techniques became apparent only after 128 tasks. This clearly demonstrates the effectiveness of our simulation methodology in comparing various jitter mitigation techniques.

5 Related Research

Application performance modeling and prediction is a well established area in HPC and there are various toolkits available. In [18], WARPP simulator was

used to model the industrial strength Chimaera benchmark and to simulate the effect of machine noise on the runtime variance of the code. A jitter distribution was first generated through the use of P-SNAP benchmark [19] and randomized jitter samples from this distribution were then injected into the simulation during execution. While the WARPP approach preserves the statistical properties of OS jitter on a given node, jitSim attempts to preserve both the statistical as well as time domain behavior of jitter across nodes through trace driven simulation. This helps us simulate both synchronized and unsynchronized (randomized) jitter accurately.

Dimemas [20] is a performance prediction tool for MPI programs that uses trace driven simulation. There are other toolkits such as the PACE toolkit [21] that simulate each individual program instruction directly. While all these toolkits can simulate runtime variance indirectly, they do not have explicit support for modeling OS jitter. The framework presented in this paper can be integrated with any of these tools to predict the overall performance of a given application.

Sottile et. al [22] presented an analysis methodology driven by message-passing traces and discussed how operating system and interconnect parameters can be generated and integrated into their methodology. They modeled the application as a message passing graph similar to barrier simulation presented in this paper.

6 Conclusions and Future Work

In this paper, we presented the design, implementation and evaluation of jitSim - a simulation framework for predicting scalability of parallel applications in presence of OS jitter. We validated the simulation results by comparing it with real data. The results presented the slowdown in the computation phase. Slowdown in the communication phase is closely tied to the underlying communication network and the barrier algorithm used (which is dependent on the MPI library). Predicting communication phase slowdown will be an extension to this work.

There are some inherent limitations of the simulation framework that we would like to address in our future work. Currently, it only models a computation phase and a communication phase characteristic of a MPI barrier. One obvious extension is to include other forms of MPI synchronization primitives (such as all-reduce, scatter, gather, etc). Furthermore, the application itself may not be strictly compute intensive and may be sensitive to the memory hierarchy layout. In such cases memory latencies also need to be added to the model.

References

1. Beckman, P., et al.: ACM SIGOPS Operating Systems Review. Operating System Issues for Petascale Systems (2006)
2. Petrini, F., et al.: The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI. In: ACM Supercomputing (2003)
3. De, P., Mann, V., Mittal, U.: Handling OS Jitter on Multicore Multithreaded Systems. In: IEEE IPDPS (2009)

4. Terry, P., Shan, A., Huttunen, P.: Improving application performance on HPC systems with process synchronization. Linux Journal (127), 68–73 (November 2004)
5. Jones, T., et al.: Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In: ACM Supercomputing (2003)
6. ZeptoOS: The small linux for big computers, http://www-unix.mcs.anl.gov/zeptoos/
7. Team, T.B.G.: An Overview of the Blue Gene/L Supercomputer. In: ACM Supercomputing (2002)
8. CrayXT5: Cray XT5 Supercomputer, http://www.cray.com/Products/XT/Systems/XT5.aspx
9. MareNostrum: Barcelona Supercomputing Center, http://www.bsc.es/
10. Hoisie, A., et al.: A Performance Comparison through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/L, Red Storm, and Purple. In: ACM Supercomputing (2006)
11. Ferreira, K., Bridges, P., Brightwell, R.: Characterizing application sensitivity to OS interference using kernel-level noise injection. In: ACM Supercomputing (2008)
12. Right-weight Linux Kernel Project: Los Alamos National Laboratory, http://public.lanl.gov/cluster/projects/index.html
13. Kaplan, L.S.: Lightweight Linux for High-Performance Computing. Linux-World.com (December 2006)
14. De, P., Kothari, R., Mann, V.: A Trace-driven Emulation Framework to Predict Scalability of Large Clusters in Presence of OS Jitter. In: IEEE Cluster (2008)
15. Beckman, P., et al.: The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In: IEEE Cluster Computing (2006)
16. De, P., Kothari, R., Mann, V.: Identifying Sources of Operating System Jitter Through Fine-Grained Kernel Instrumentation. In: IEEE Cluster (2007)
17. De, P., Garg, R.: The Impact of Noise on the Scaling of Collectives: An Empirical Evaluation. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2006. LNCS, vol. 4297. Springer, Heidelberg (2006)
18. Hammond, S., et al.: WARPP: a toolkit for simulating high-performance parallel scientific codes. In: International Conference on Simulation Tools and Techniques (2009)
19. P-SNAP Benchmark: P-SNAP Benchmark, http://www.ccs3.lanl.gov/pal/software/psnap/
20. Girona, S., Labarta, J.: Sensitivity of Performance Prediction of Message Passing Programs. In: International Conference on Parallel and Distributed Processing Techniques and Applications (1999)
21. Nudd, et al: PACE: A Toolset for the Performance Prediction of Parallel and Distributed Systems. The International Journal of High Performance Computing (2000)
22. Sottile, M., Chandu, V., Bader, D.: Performance analysis of parallel programs via message-passing graph traversal. In: IEEE IPDPS (2006)