

A Trace-driven Emulation Framework to Predict Scalability of Large Clusters in Presence of OS Jitter*

Pradipta De, Ravi Kothari, Vijay Mann

IBM India Research Laboratory, New Delhi, India

{pradipta.de, rkothari, vijamann}@in.ibm.com

Abstract—Various studies have pointed out the debilitating effects of OS Jitter on the performance of parallel applications on large clusters such as the ASCI Purple and the Mare Nostrum at Barcelona Supercomputing Center. These clusters use commodity OSes such as AIX and Linux respectively. The biggest hindrance in evaluating any technique to mitigate jitter is getting access to such large scale production HPC systems running a commodity OS. An earlier attempt aimed at solving this problem was to emulate the effects of OS jitter on more widely available and jitter-free systems such as BlueGene/L. In this paper, we point out the shortcomings of previous such approaches and present the design and implementation of an emulation framework that helps overcome those shortcomings by using innovative techniques. We collect jitter traces on a commodity OS with a given configuration, under which we want to study the scaling behavior. These traces are then replayed on a jitter-free system to predict scalability in presence of OS jitter. The application of this emulation framework to predict scalability is illustrated through a comparative scalability study of an off-the-shelf Linux distribution with a minimal configuration (runlevel 1) and a highly optimized embedded Linux distribution, running on the IO nodes of BlueGene/L. We validate the results of our emulation both on a single node as well as on a real cluster. Our results indicate that an optimized OS along with a technique to synchronize jitter can reduce the performance degradation due to jitter from 99% (in case of the off-the-shelf Linux without any synchronization) to a much more tolerable level of 6% (in case of highly optimized BlueGene/L IO node Linux with synchronization) at 2048 processors. Furthermore, perfect synchronization can give linear scaling with less than 1% slowdown, regardless of the type of OS used. However, as the jitter at different nodes starts getting desynchronized, even with a minor skew across nodes, the optimized OS starts outperforming the off-the-shelf OS.

I. INTRODUCTION

Operating system jitter (henceforth referred to as OS jitter) refers to the interference experienced by an application due to scheduling of daemon processes and handling of asynchronous events such as interrupts. Various studies [1], [2] have shown that parallel applications on large clusters suffer considerable degradation in performance (up to 100% degradation at 4096 processors [2]) due to OS jitter. Several large scale HPC systems, like the Blue Gene/L [3] and Cray XT4 [4],

avoid OS jitter by making use of a customized light-weight microkernel at the compute nodes. These customized kernels typically do not support general purpose multitasking and may not even support interrupts. Moreover, these systems require applications to be modified or ported for their respective platforms. Other systems like the ASCI Purple and the Mare Nostrum at the Barcelona Supercomputing Center [5] make use of commodity OSes (AIX and RedHat Enterprise Linux respectively) and thereby suffer from OS jitter [6].

With a growing interest in the use of commodity OSes for HPC systems [7] [8] [9], there is a much greater need today than ever before, to develop and to evaluate various techniques for mitigating OS jitter. However, 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 the availability of such large scale production HPC clusters running commodity OSes worldwide is extremely limited for experimentation and validation purposes. An earlier attempt from Beckman et al. [10] aimed at solving this problem by emulating the effects of OS jitter on a jitter-free system such as BlueGene. This approach definitely holds a lot of promise, as BlueGene is one of the most scalable systems today and has the largest footprint in terms of number of nodes. However, Beckman et al. found out that the overhead of the interval timer on BlueGene/L, which they used for injecting synthetic jitter, was 16 μ s and they could not inject any jitter less than 16 μ s. Our analysis reveals that nearly 98% of the jitter encountered on off-the-shelf Linux distributions with minimal configurations is less than 14 μ s. The percentage is even higher on more optimized versions of Linux. In order for any such emulation approach to produce correct results, it should take into account the inherent limitations of the jitter emulation platform (which can be in the same range as the majority of jitter durations - around 10-15 μ s) and take necessary steps to overcome them.

In this paper, we point out the several such shortcomings of previous emulation attempts. We have designed and implemented an emulation framework that overcomes these shortcomings by using innovative techniques and can accurately predict the scaling behavior of parallel applications

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

running on large clusters in presence of OS jitter. It builds on some of the tools that we have developed in our earlier work [11] [12]. We collect jitter traces from a commodity OS with a given configuration, under which we want to study the scaling behavior. These traces are then replayed on a jitter-free system, while a typical parallel application or benchmark with repeated compute-barrier phases is executed.

The jitter emulation framework makes use of several innovative techniques to avoid possible pitfalls that may lead to inaccuracies in jitter emulation. This includes techniques such as scaling up all jitter values (and the corresponding runtime of the parallel application that executes while jitter is being introduced) in time domain in order to emulate even the smallest possible jitter, and then scaling down the final results. We validate the accuracy of our jitter emulation on a single node by comparing the jitter distributions from an Intel machine running an off-the-shelf Linux OS with that collected from a Blue Gene/L node with emulated jitter. In order to validate the accuracy of the predicted results from emulation, we compare them against results from a real cluster. A close match between the predicted results and the real data indicates the correctness of our methodology. The application of this emulation framework to study OS jitter is illustrated through a comparative scalability study of an off-the-shelf Linux distribution with a minimal configuration (run level 1) and a highly optimized embedded Linux distribution, running on the IO nodes of BlueGene/L.

The main contributions of this paper are the following:

- 1) A generic methodology for predicting the scaling behavior of a cluster running various OS configurations or employing various jitter handling techniques.
- 2) Implementation of a jitter emulation framework that makes use of several innovative techniques to avoid possible pitfalls that may lead to inaccuracies in jitter emulation.
- 3) A comparative scalability study of an off-the-shelf Linux distribution with a minimal configuration (runlevel 1) and a highly optimized embedded Linux distribution, running on the IO nodes of BlueGene.
- 4) Experimental results that provide the following insights into scalability behavior of large clusters running a commodity OS:
 - An optimized OS along with a technique to synchronize jitter can reduce the performance degradation due to OS jitter from 99% (in the case of the off-the-shelf Linux without any synchronization) to a much more tolerable level of 6% (in the case of the highly optimized BlueGene/L IO node Linux with synchronization) at 2048 processors.
 - Perfect synchronization can give linear scaling with less than 1% slowdown, regardless of the type of OS used. However, as the jitter at different nodes starts getting desynchronized, even with a minor skew across nodes, the optimized OS starts outperforming the off-the-shelf OS.

The rest of the paper is organized as follows. Section II presents an overview of our methodology. We also describe some of the pitfalls that can lead to inaccuracies in jitter emulation. In section III, we present the implementation details of various components of our jitter emulation framework. Section IV describes in detail the innovative techniques employed by the framework to avoid the pitfalls introduced in section II and also presents a validation of those techniques. Section V presents our experimental results. Section VI compares the scalability prediction results from jitter emulation with data from a real cluster. In Section VII, we present an overview of related research. Finally, we conclude and give directions for our future research in section VIII.

II. METHODOLOGY

In this section we outline some of the requirements for emulating OS jitter on a jitter free system and give an overview of our methodology.

A. Requirements

Any methodology for emulating OS jitter by introducing synthetic jitter should take note of the following pitfalls and should incorporate ways to overcome them.

- *Pitfall 1:* Introduction of synthetic jitter on a given platform might have an overhead of its own. This overhead should be measured and accounted for while introducing any synthetic jitter. For example, if one uses a timer to introduce jitter asynchronously, the *settimer* system call has an overhead of its own which should be measured and adjusted for. It is, therefore, possible that on a given system, any jitter which is less than a particular value, cannot be introduced as it is smaller than the overhead itself.
- *Pitfall 2:* There are system specific limitations on resolution of timers and sleep system calls. This implies that there has to be a minimum time gap between any two jitter events that can be introduced. It is possible that jitter events that occur in quick succession of each other can not be introduced with precision because of this limitation. Imprecision in sleep calls due to timer resolution can also introduce a skew across nodes while emulating perfectly synchronized jitter.
- *Pitfall 3:* Realistic emulation of jitter on a large number of nodes using a jitter trace collected from a single node requires that the single node trace be very large and the individual nodes replay only a small portion of that trace. If all nodes replay the entire trace (even if they start at different points in the trace), all nodes will end up observing the same set of jitter samples (albeit at different points of time) - something which is unlikely to happen in a real setup.

Avoiding pitfall 1 becomes extremely important considering the fact that the overhead of systems calls such as *settimer* can be in the range of 10-15 μ s and that nearly 98% of the jitter encountered on off-the-shelf Linux distributions with minimal configurations is less than 15 μ s in magnitude.

Algorithm 1 The Parallel Benchmark kernel

```
1:  $t_s = \text{gettimeofday}()$ ;  
2: while elapsed_time < period do  
3:   MPI_Barrier  
4:    $t_1 = \text{get\_cycle\_accurate\_time}$   
5:   do_work ( iteration_count )  
6:    $t_2 = \text{get\_cycle\_accurate\_time}$   
7:   MPI_Barrier  
8:    $t_3 = \text{get\_cycle\_accurate\_time}$   
9:   compute time (qt) =  $t_2 - t_1$ , barrier time (bt) =  $t_3 - t_2$ , total  
   completion time (qtb) =  $t_3 - t_1$   
10:   $t_e = \text{gettimeofday}()$ ;  
11:  elapsed_time =  $t_e - t_s$   
12:  store qt, bt, qtb  
13:  MPI_Bcast(elapsed_time)  
14: end while
```

B. Overview

Our overall methodology for predicting scalability of parallel applications running on large clusters in the presence of OS jitter is based on the following steps:

- 1) An application or a parallel benchmark [12] is run on a given number of nodes on a jitter free system and is expected to finish a given amount of work. The time taken to finish this work is calculated - T_{jitter_less} .
- 2) OS jitter trace is collected from a node that is running an operating system with a particular configuration under which we want to study the scalability behavior. This can be done using various tools. We make use of our own single node benchmark [11], which is based on reading the CPU timestamp register in a tight loop. We refer to this as the “TraceCollector”.
- 3) The jitter trace is then replayed by an emulator (referred to as the “JitterEmulator”) that introduces jitter by executing a busy loop for the given period.
- 4) While jitter is being replayed from the trace, the same application or parallel benchmark used in step 1 is run with the same amount of work. The total amount of time taken for that work to complete is calculated - T_{jitter} . This time (T_{jitter}) is then compared with the time taken by the application on that system in the absence of any jitter (T_{jitter_less}) to calculate a percentage slowdown.
- 5) The above steps are repeated for an increasing number of nodes (or processors) to get the scalability behavior of the cluster in the presence of jitter that is characteristic of the OS from which the jitter trace was collected in step 2.

We describe each of the above steps in more detail in the following section.

III. IMPLEMENTATION DETAILS

In this section we present details about the different components of our jitter emulation framework.

A. Parallel Benchmark

The parallel benchmark [12] represents a typical parallel application with repeated compute-barrier phases. The kernel

of parallel benchmark is shown in Algorithm 1. Each processor in the cluster runs a MPI task that executes the parallel benchmark kernel.

The function call *do_work()* 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:

$$x_{j+1} = (a * x_j + b) \bmod p \quad (1)$$

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 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 *qtb*. The broadcast call (in line 13) ensures that the task with rank 0 sends the current elapsed time to all other tasks. This, in turn, ensures that all tasks terminate after executing the same number of iterations of the *do_work* loop. More details about the parallel benchmark can be found in [12].

B. TraceCollector

TraceCollector is a variation of the single node benchmark used by us in our earlier work [11] and by other researchers [10]. 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 kernel is shown in Algorithm 2. It is based on reading the timestamp register on the CPU in a tight loop (lines 7 and 17). First, the minimum time taken to read the timestamp register (t_{min}) is recorded (lines 4-11). After this, the timestamp register is read in a loop and difference between successive timestamp readings are compared against the specified threshold (line 18). We currently set this threshold to be 10 times the minimum difference observed (t_{min}). These timestamp deltas represent the number of cycles required to read the timestamp register. Most of these deltas (around 99% of them) would be very small and these correspond to the actual number of cycles required for the *rdtsc* instruction (t_{min} - it is roughly equal to 88 cycles on Intel Xeon - $0.03\mu s$ on a 2.8 GHz machine and equal to 75 cycles on a BlueGene/L IO Node at 700 MHz - $0.11\mu s$). However, when a daemon process is scheduled or an interrupt is handled or any other system activity occurs that takes the CPU away from the application, the deltas are much higher. If the difference is greater than the specified threshold, the timestamps themselves are recorded (lines 20-21). The timestamps are later stored to create a jitter trace and the timestamp deltas are added to a histogram (referred to as the user-level histogram) (lines 27-29).

Algorithm 2 TraceCollector

```
1: i=0; indexA2=0; threshold=10;
2: N=(1*1024*1024); tmin=LONG_LONG_MAX;
3: /* first find tmin */
4: current_reading = rdtsct();
5: while (i < N) {
6:   prev_reading = current_reading;
7:   current_reading = rdtsct();
8:   if(current_reading - prev_reading < tmin)
9:     tmin = current_reading - prev_reading;
10:  i++;
11: }
12: i=0;
13: /* now compare all timestamps against tmin * threshold */
14: current_reading = rdtsct();
15: while (i < N) {
16:   prev_reading = current_reading;
17:   current_reading = rdtsct();
18:   if(current_reading - prev_reading > tmin * threshold) {
19:     /* collect jitter timestamps */
20:     A2[indexA2++] = prev_reading;
21:     A2[indexA2++] = current_reading;
22:     i++;
23:   }
24: }
25: for (i=0; i < indexA2 - 2 ; i++) {
26:   /* store jitter trace - timestamp, runtime, sleeptime */
27:   store(A2[i], A2[i+1]-A2[i], A2[i+2]-A2[i+1]);
28:   /*add jitter duration - timestamp deltas to user-level-histogram*/
29:   add_to_distribution(A2[i+1]-A2[i]);
30: }
```

The jitter trace is generated by storing the timestamp (*i.e.* the start time of the jitter activity), the duration of the jitter activity, and the time difference between successive jitter events (line 24). We will refer to these as the *start-time*, *run-time*, and *sleep-time* respectively.

Ideally, the tight loop that reads the timestamp register should be as small as possible (with only the *rdtsct* instruction) and all processing of the collected timestamps should be done outside the loop so that there are no overheads incurred while measuring jitter. However, this would require huge amounts of physical memory in order to collect a reasonably long trace. One solution is to collect jitter samples in several rounds and process the collected data after each round. This has the drawback that certain periodic jitter events might be missed and that the collected trace will be discontinuous. The overhead introduced by the additional assignment and comparison statements in the loop presented in Algorithm 2 is quite small (much smaller than the 0.3 μ s threshold used for measuring jitter).

1) *Jitter Trace Collection*: For experiments in this paper, we run the TraceCollector to collect jitter traces from two different Linux configurations (an off-the-shelf Linux distribution and a highly optimized embedded version of Linux) running on two different platforms (Intel and PowerPC).

Fedora Core 5 in runlevel 1 with networking support running on Intel: This configuration has a very basic minimal set of daemon processes and drivers and can be thought of as

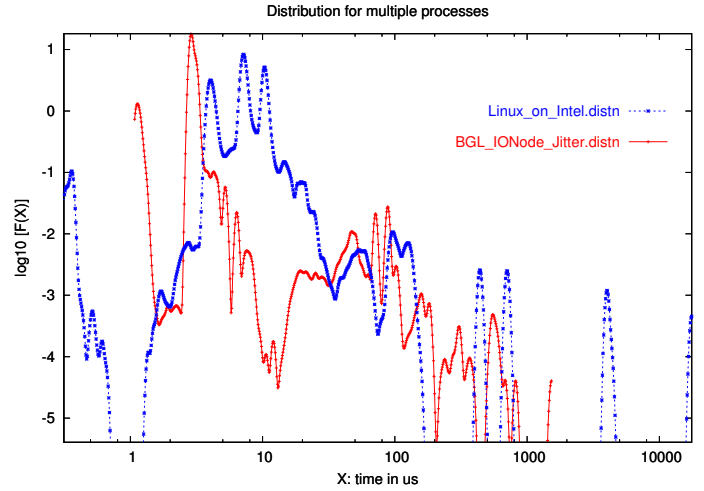


Fig. 1. Distribution of jitter from off-the-shelf Linux running on Intel in runlevel 1 and optimized Linux running on BlueGene/L IO Nodes

an off-the-shelf configuration optimized for HPC applications. The machine had a single Intel Xeon (TM) 2.8 GHz CPU with a cache size of 512 KB Cache and 1GB RAM. The timer interrupt interval was configured to be the default of 4 milliseconds (*i.e.* 250Hz).

Highly optimized embedded version of Linux running on BlueGene/L IO Nodes: The BlueGene/L IO nodes (PowerPC 440, 700 MHz) run a highly optimized version of Linux (kernel version 2.6.5-348). This Linux distribution is referred to as embedded [13] as it does not use any swap space, has an in-memory root file system, uses little memory, and lacks the majority of daemons and services found in off-the-shelf distributions.

The above traces were collected by running the TraceCollector to collect 1 million jitter samples (that takes approximately an hour in our experiments) using the methodology described above. Probability distributions of jitter durations in both the jitter traces have been plotted using the Parzen window density estimation technique [14] in Figure 1. The y-axis is a logarithm of the probability that a particular jitter occurs and the x-axis is the jitter duration in μ s. An analysis of off-the-shelf Linux runlevel 1 jitter trace reveals that nearly 98% of jitter values are less than 14 μ s. For the highly optimized BlueGene/L IO node Linux, nearly 99.5% of the jitter values are less than 5 μ s. It does not have a large number of points in the 10-50 μ s range like the runlevel 1 trace.

C. JitterEmulator

In this section, we first give reasons for using Blue Gene/L as the emulation platform followed by some details of the *JitterEmulator* component.

1) *Blue Gene/L as the jitter emulation platform*: We use Blue Gene/L as the jitter emulation platform. Blue Gene/L uses a specialized light-weight microkernel on the compute nodes that does not allow multitasking and therefore does not have any daemon processes or a process scheduler. This creates an operating environment that is free of any system induced jitter. This has been illustrated in earlier studies [6]. We verified it again by running the TraceCollector on a Blue

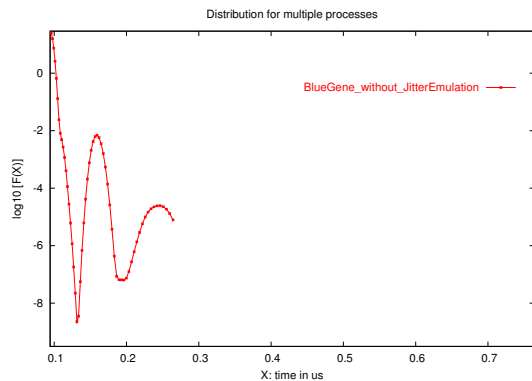


Fig. 2. Distribution of jitter on Blue Gene/L : output from the TraceCollector

Gene/L node. The output of the TraceCollector is shown in Figure 2.

We chose Blue Gene/L as the jitter emulation platform as it provides us with the scale at which we want study the impact of jitter. Even though, the experiments for this paper were conducted on a single rack of Blue Gene/L (*i.e.* 1024 nodes or 2048 processors), one can use existing installations of Blue Gene/L that have 65,536 dual-processor compute nodes [15].

2) *JitterEmulator Details*: JitterEmulator is a component that resides on each processor where a MPI task runs. Jitter emulator reads the jitter trace file generated by the TraceCollector into an in-memory data structure where each record consists of of a *start-time* (timestamp for the jitter activity), a *run-time* (duration of the jitter activity), and *sleep-time* (time gap between the current and the next jitter activity). It then introduces the jitter corresponding to a particular record by executing a busy loop for the given run-time, followed by executing a sleep for the given sleep-time.

The JitterEmulator can either run within the context of the application process or within its own process context. Running the JitterEmulator within the application process context has the disadvantage that the application has to be recompiled with a call to the JitterEmulator. In this case, the JitterEmulator introduces jitter through an interval timer mechanism by periodically stopping the application and executing a busy loop for the specified period. Running the JitterEmulator within its own separate context, on the other hand, has the advantage of not requiring a recompilation of the application. However, the target platform must support multiple processes or multitasking for this option and one must ensure that the JitterEmulator process has a much higher priority than the application process (so that it always gets a chance to run). Since Blue Gene/L does not support multiple processes on the compute nodes, we make use of the first option and run the JitterEmulator within the application process context. The Parallel Benchmark is recompiled with a call to the JitterEmulator.

Choosing the point in the jitter trace from where the JitterEmulators at all nodes start introducing jitter is an important decision and it can have interesting ramifications. In a cluster that has unsynchronized jitter, different kinds of jitter events 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 events will hit each node at the same time. In order to emulate the unsynchronized jitter scenario, the JitterEmulators at all nodes start introducing jitter from different randomly chosen points in the jitter trace. To emulate synchronized jitter, the JitterEmulators at all nodes start introducing jitter from the same randomly chosen point in the jitter trace.

The JitterEmulator makes use of the interval timer mechanism (using the *setitimer* system call) to transfer control to itself. It sets up a signal handler for the SIGALRM signal. The *setitimer* system call sets up a timer, which on expiry delivers the SIGALRM signal to the parallel benchmark application process. The signal handler (timer handler) routine executes a busy loop for the jitter duration (*i.e.* the *run-time*) and then sets the next timer after an interval equal to the *sleep-time* in the current jitter trace record. It then increments the index in the jitter trace so that it picks up the next jitter record in the trace when the timer expires again.

IV. SAFEGUARDING AGAINST THE PITFALLS

The JitterEmulator makes use of several innovative techniques to *safeguard against the pitfalls* discussed earlier in section II.

A. Safeguard against Pitfall 1

We measured the overhead of introducing jitter on Blue Gene/L to be in the range 14-16 μ s. Out of this, the overhead of the *setitimer* system call is about 10 μ s (measured by having an empty timer handler routine). The remaining 4-6 μ s are spent in the various steps in the timer handler routine. In order to offset this overhead, we reduce the run-time of all jitter values by 14 μ s. This implies that any jitter value less than 14 μ s can not be introduced during emulation. This is unacceptable as most timer interrupt activity is less than 14 μ s. As mentioned earlier, nearly 98% of jitter values in the off-the-shelf Linux runlevel 1 jitter trace are less than 14 μ s. For the highly optimized BlueGene/L IO node Linux, this number is even higher and almost 99.5% of the jitter values are less than 14 μ s. The earlier work from Beckman et al. [10] on jitter emulation suffers from this pitfall and they are unable to introduce any jitter which is less than 14 μ s and as we know, most of the jitter encountered on an off-the-shelf commodity OS with a minimal configuration or an optimized OS is in this range.

To overcome this, we scale all the jitter values (run-time and sleep-time) as well as the *quanta* time and the period of the Parallel Benchmark by a constant factor. At the end of the experiment, all values (competition times (*qibt*), and barrier times (*bt*)) are scaled down by the same factor. Ideally, we should use a scaling factor of 14 to ensure that all jitter values in the trace can be emulated. However, this would result in running the Parallel Benchmark for a period 14 times longer than the intended period. Analysis of off-the-shelf Linux runlevel 1 jitter trace revealed that 77% of the jitter values are greater than or equal to 5 μ s. Hence, with a

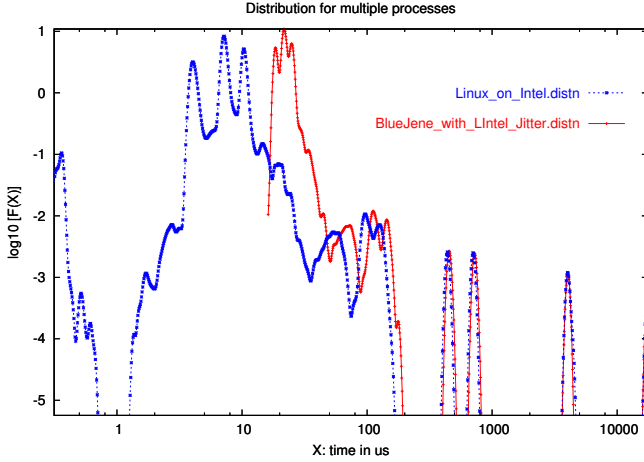


Fig. 3. Verification of jitter emulation on a single node without applying any scaling factor

scaling factor of 3 we were able to introduce all jitter values that are greater than or equal to $5 \mu\text{s}$ and emulate 77% of the jitter samples in the trace. However, in the BlueGene/L IO node trace, even with a scale factor of 8, we could emulate only 31% of the trace. This implies that nearly 70% of the jitter values in this trace are less than $1.75 \mu\text{s}$. However, with a scale factor of 9, we were able to emulate nearly 92% of the trace, which indicates that there are a large number of jitter values between $1.55 \mu\text{s}$ and $1.75 \mu\text{s}$.

B. Safeguard against Pitfall 2

Our measurements indicated that the smallest value below which the *setitimer* system call stops working on Blue Gene/L is nearly $5 \mu\text{s}$. This implies that it is not possible to set a timer that wakes up in less than $5 \mu\text{s}$. Hence, if there is a sleep-time that is less than $5 \mu\text{s}$ we replaced it with 0, effectively resulting in merging of two jitter values into one. Furthermore, the resolution of the timer on BlueGene/L is approximately $130 \mu\text{s}$ on an average and has a standard deviation of about $20 \mu\text{s}$. This has two interesting ramifications.

First, any *sleep-time* value in the jitter trace that is less than $130 \mu\text{s}$ is likely to result in sleep times longer than the specified value. However, this should have only a minimal effect on our emulation results as only 3% of the sleep times are less than $130 \mu\text{s}$ in the off-the-shelf Linux runlevel 1 jitter trace.

Second, the gap between any two jitter instances is likely to have some inaccuracy because of the resolution of the sleep time being $130 \mu\text{s}$. This is compensated by adjusting all the sleep times for the timer resolution. However, the fact that the sleep times have standard deviation of about $20 \mu\text{s}$, implies that even if all the nodes start jitter emulation at the same index in the trace (to emulate synchronized jitter), they will get out of sync with time because of the variance in sleep time. This causes problems in emulating perfectly synchronized jitter. The earlier work from Beckman et al. [10] does not take into account the variance in sleep calls and hence their emulation of synchronized jitter does not represent perfect synchronization. We overcome this limitation by resetting the trace index to the

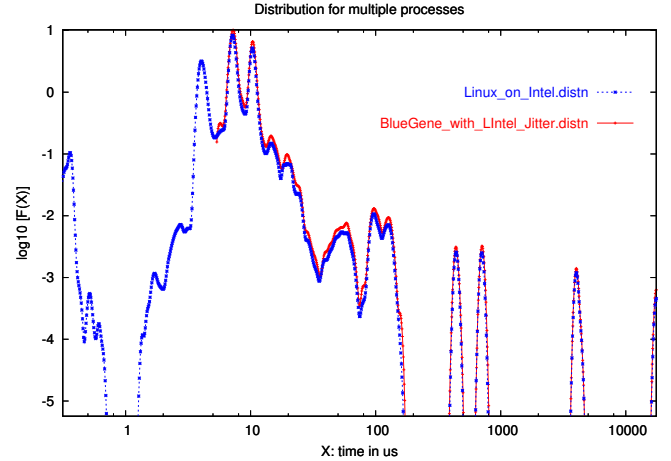


Fig. 4. Verification of jitter emulation after applying a scaling factor of 3

same randomly chosen value across all nodes in each compute phase instead of doing it just once at startup. This helps us emulate perfectly synchronized jitter.

C. Safeguard against Pitfall 3

In order to ensure that all nodes do not end up observing all set of jitter values (*i.e.* the entire trace) in a given experiment and that each node replays only a reasonably small portion of the entire trace, the JitterEmulator framework stops the experiment as soon as any one node finishes a fixed percentage of the total trace. For all experiments in this paper, we introduce only one third of the trace in each run.

D. Validating the Safeguards on a Single Node

We validate the accuracy of our jitter emulation and the effectiveness of the innovative techniques mentioned above on a single node by running the TraceCollector on single processor on BlueGene/L while jitter was being replayed by the JitterEmulator. This step required a recompilation of the TraceCollector with a call to the JitterEmulator. We compared the output of the TraceCollector (frequency distribution of jitter durations) from an Intel machine running Linux (Fedora Core 5, 2.6.20.7 kernel, runlevel 1) with that of a Blue Gene/L node with emulated jitter using a trace collected from the same Intel machine running Linux and a scale factor of 3. In order to validate that the jitter emulation framework has successfully emulated jitter on Blue Gene/L that is representative of the jitter encountered on an Intel machine running Linux (Fedora Core 5, 2.6.20.7 kernel, runlevel 1), the two distributions should match.

Figures 3 and 4 plot the jitter duration for the two cases mentioned above as probability distributions using Parzen window density estimation technique [14]. Figure 3 shows the distribution without applying any scaling factor (safeguard against pitfall 2 above). The emulated jitter curve starts only after $14 \mu\text{s}$. Although nature of the two distributions is similar in the region greater than $14 \mu\text{s}$, the emulated jitter curve is shifted to the right in the middle of the graph (region between 14 and $125 \mu\text{s}$). This is because all jitter values include an

additional 14 μs due to the overhead (because of the pitfall 2 mentioned above - overhead of jitter introduction itself being 14 μs). This effect is not visible in the peaks around and beyond 500 μs as the overhead of 14 μs becomes a small percentage of the actual jitter value.

Figure 4 shows the distribution after scaling all jitter values (runtimes and sleeptimes) by a factor of 3. The usefulness of this technique can be observed immediately. In this case, we were able to introduce jitter values greater than 5 μs and the two distributions exhibit close matching after the 5 μs point on the x-axis.

We also use a quantitative measure called KL divergence [16] to measure the difference between two probability distributions. Smaller the KL divergence score for two distributions, the more closely they match. For two identical distributions, the KL divergence score is 0. The KL divergence score for the two distributions in the no-scaling graph (Figure 3) is 18117, whereas the KL divergence score for the two distributions after applying a scale factor of 3 (Figure 4) is 873 (improvement by a factor of close to 20).

V. EXPERIMENTAL RESULTS

We conducted three sets of experiments to study the effect of OS jitter on the performance of fine grained parallel applications as number of processors in the cluster were increased from 16 to 2048 with 1 MPI task per processor. One full rack of Blue Gene/L (*i.e.* 1024 nodes) was used in the “Virtual Node” mode, which meant that both the processors on a node were used for executing MPI tasks. We also conducted experiments in the “CoProcessor” mode, where only one processor was used for computation, while the second processor was used for communication tasks. Since, our results for “CoProcessor” mode closely match the results for “Virtual Node” mode, we do not present them in this paper.

The three sets of experiments emulated jitter in various ways to study application performance under:

- 1) jitter present in off-the-shelf Linux distribution in runlevel 1 and in a highly optimized embedded Linux distribution;
- 2) jitter that has been synchronized once at start up across all nodes;
- 3) jitter that is synchronized across all nodes in each compute phase;

For all the experiments, the number of processors were increased from 16 to 2048. Each processor had an instance of the Parallel Benchmark (the MPI task) executing the compute-barrier loops and an instance of the JitterEmulator (running in the same process context as the MPI task) introducing jitter from the collected jitter traces. For a given processor count, first a maximum completion time for each phase (referred to as qbt in section III - sum of compute and barrier times) was calculated across all processors. Maximum completion time ($\text{MaxQTBT}(\text{jitter})$) in each phase was averaged over a number of iterations to come up with the average maximum completion time ($\text{AvgMaxQTBT}(\text{jitter})$). This was compared with the average maximum completion time for the Parallel

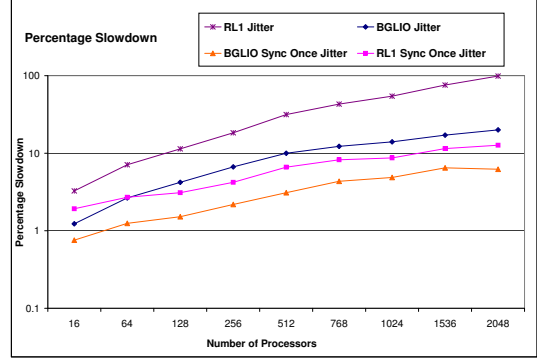


Fig. 5. Percentage slowdown for a compute phase under various Linux distributions and jitter synchronization settings

Benchmark (with the same work quanta of 1 ms) executing on Blue Gene/L without any jitter. Percentage slowdown (SD) for a n processor case, is then calculated as:

$$SD_n = \frac{\text{AvgMaxQTBT}_n(\text{jitter}) - \text{AvgMaxQTBT}_n(\text{no jitter})}{\text{AvgMaxQTBT}_n(\text{no jitter})} * 100 \quad (2)$$

For all the experiments, we first collect the 2 traces mentioned in section III-B - from the off-the-shelf Linux (Fedora core 5) running on Intel in runlevel 1, and from the highly optimized embedded Linux on BlueGene/L IO nodes. These traces are collected by running the TraceCollector for an hour using the methodology described in sections II and III.

A. Experiment 1: Impact of jitter present under various Linux distributions

The traces collected, are then used by the JitterEmulator to introduce jitter while the Parallel Benchmark executes. JitterEmulators on different nodes (*i.e.* processors) start at different randomly chosen points in the jitter trace. The work quanta for the Parallel Benchmark was set to 1 millisecond. The percentage slowdown for the above cases is shown in Figure 5. RL1 refers to the off-the-shelf Linux in runlevel 1 and BGL IO Node refers to the optimized embedded Linux used on BlueGene/L IO nodes.

It can be observed that as number of processors are increased from 16 to 2048, the percentage slowdown reaches close to 99% with runlevel 1 jitter. The highly optimized embedded Linux used on BlueGene/L IO nodes, fares much better, and in the worst case, the percentage slowdown reaches close to 20% at 2048 processors. The percentage slowdown numbers reported by our jitter emulator framework are in the same range as those reported by Petrini et al. [2] on a real system (ASCI Q) running a commodity OS (AIX) where they detected nearly 100% performance degradation at 4096 processors .

B. Experiment 2: Impact of synchronizing jitter once at startup across all nodes

This experiment is carried out in a similar way to the above experiment except one difference. Instead of starting from different randomly chosen points in the jitter trace, the

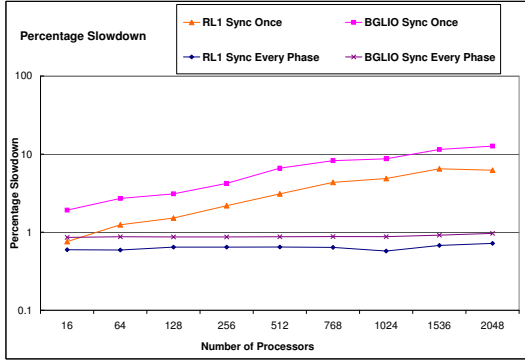


Fig. 6. Comparing the percentage slowdown for one time jitter synchronization with a per phase jitter synchronization

Jitter Emulators on different nodes start at the same randomly chosen point to emulate synchronization of jitter across all nodes. This results in a one time synchronization of jitter across all nodes at the start of work. Experiment is carried out for both off-the-shelf Linux runlevel 1 trace and the BlueGene/L IO node Linux trace. Work quanta for the Parallel Benchmark is kept the same as the above experiment (1 millisecond).

Percentage slowdown for the synchronized traces is shown along with the normal traces in Figure 5.

It can be observed, that synchronizing jitter across all nodes brings great benefits. At 2048 processors, percentage slowdown is decreased from 99% to 13% for runlevel 1 jitter and from 20% to 6% for the BlueGene/L IO node Linux jitter. Our results are in the same range as those reported by Terry et al. [17] where they detected a performance improvement of nearly 50% through synchronized scheduling of processes.

C. Experiment3: Impact of synchronizing jitter in each compute phase

In the previous experiment, even though the JitterEmulators start at the same index, they get desynchronized fairly soon because of the variance in sleep times (timer resolution on BlueGene/L is $130 \mu s$ on an average with a standard deviation of $20 \mu s$ - pitfall 2 mentioned in section IV). We observed that the maximum desynchronization between the jitter emulators at different nodes at the end of all work was roughly $3820 \mu s$ for BlueGene/L IO node Linux trace and $5300 \mu s$ for Linux runlevel 1 trace. In order to overcome this and emulate perfectly synchronized jitter, we repeated the above experiment by replacing a one time start up synchronization with synchronization in each compute phase. We do this by resetting the trace index to the same random index across all nodes in each compute phase. Percentage slowdown numbers for perfect synchronization experiment are compared with one time synchronization numbers in Figure 6.

The curves for synchronization in each compute phase show linear scaling for both BlueGene/L IO Node Linux trace as well as Linux runlevel 1 trace. The following important observations can be made:

- 1) Even a minor drift of around 3 ms in case of BGLIO trace (or 5 ms in case of RL1 trace) can cause a degradation of 6% for BGLIO trace (and around 13% in case of runlevel 1 trace). These curves represent worst case slowdown numbers with jitter synchronization. The flat curves, on the other hand, represent best case slowdown and show linear scaling (perfect synchronization).
- 2) Even though the per phase synchronization curves are flat, percentage slowdown numbers for the runlevel 1 trace (0.7%) are lower than the optimized BGLIO Node kernel (0.9%). An analysis of the two traces indicate that the runlevel 1 trace consists of infrequent but long duration jitter whereas BGLIO Node trace consists of frequent but very short duration jitter. In the runlevel 1 trace, only 0.6 jitter samples on an average hit each Parallel Benchmark iteration whereas in the BGLIO trace, nearly 1.04 jitter samples on an average hit each iteration. This indicates, that for perfect synchronization, it doesn't really matter what kernel is being used and even an off-the-shelf kernel may outperform an optimized kernel. However, the situation changes drastically once the nodes start getting out of sync, and the optimized kernel has much lower slowdown as compared to the off-the-shelf kernel, which has infrequent but long duration jitter.

These two settings represent the two extremes of jitter synchronization and in practice, jitter is synchronized neither every phase nor just once, but every few compute phases.

VI. VALIDATING THE OVERALL APPROACH

To validate the accuracy of slowdown prediction results using jitter emulation, we ran the Parallel Benchmark on a real cluster and the results were compared with percentage slowdown numbers reported by JitterEmulator. The real cluster that we got access to had 13 nodes. Each node had 32 Power 6 4GHz processors, giving us a total of 416 processors. They all ran SUSE Linux Enterprise Server 10 SP2 with 2.6.24 kernel. First, we collected the jitter traces from a single node by running the Trace Collector. In order to emulate 32 cores of a single SMP node, 32 instances of Trace Collector were run simultaneously (with each instance bound to a unique processor) and 32 different traces were collected. This is because different cores of a single SMP node, have a much higher degree of correlation in their jitter profiles and they need to be modeled as a group. During emulation, processors are divided into sets of 32 and JitterEmulators in each such set use one of these 32 jitter traces. Analysis of jitter traces revealed that a scale factor of 6 enabled us to cover 65% of the trace. We first conducted a single node validation by emulating a jitter trace from one of the 32 cores using a scale factor of 6. Figure 7 plots the jitter distribution collected from a BlueGene/L node with emulated jitter (using a scale factor of 6) and that from a real trace as probability distributions using Parzen window density estimation technique. The KL divergence measure for the two distributions in Figure 7 is 2510.

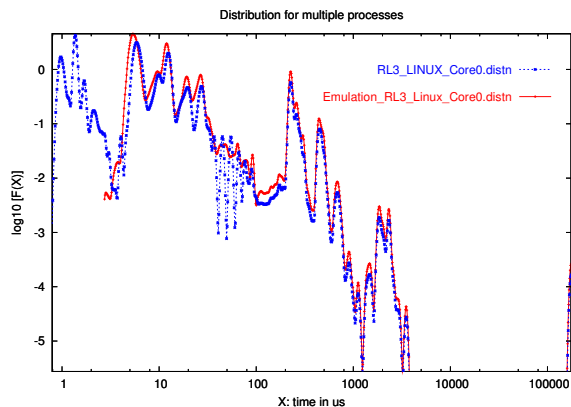


Fig. 7. Single node validation of jitter emulation using a scale factor of 6 for a real cluster node with Linux in runlevel 3

We then conducted a run of the Parallel Benchmark on the real cluster, while increasing the number of processors gradually from 32 to 416. Parallel Benchmark was used with a work quanta of 1 ms. These nodes were not connected by a high speed interconnect and hence we chose to consider the effect of jitter only during the compute phase (and how faster tasks wait for slower tasks at the barrier). The effect of jitter encountered during barrier (*i.e.* during message passing over the network) was not considered as it depends on the nature of the interconnect. Emulation results for total completion time (*i.e.* sum of compute and barrier time - referred to as *qbt* in section III) on BlueGene/L will match real cluster results only if the cluster has a high speed interconnect. The real cluster run was followed by a full multi node emulation run on BlueGene/L while number of processors were increased from 32 to 416. The settings for Parallel Benchmark was kept identical as the real run.

For both the runs, Maximum compute time (*i.e.* maximum *qt* using the terminology used in section III) across all processors was computed in each phase to capture the effect of jitter. Maximum compute time (MaxQT) in each phase was averaged over a number of iterations and then a percentage slowdown for compute time for a *n* processor case was calculated as follows:

$$SD_n = \frac{AvgMaxQT_n(jitter) - AvgMaxQT_n(no\ jitter)}{AvgMaxQT_n(no\ jitter)} * 100 \quad (3)$$

The above equation is identical to the slowdown equation mentioned earlier in section V, except that, here we make use of only *qt* (compute time), and not of *qbt* (compute+barrier time). For the real cluster run, the compute quanta (1 ms) is taken as the reference no jitter AvgMaxQT (AvgMaxQT(no jitter)). A comparison of the percentage slowdown numbers for emulation and real runs is given in Figure 8. The numbers match fairly closely and this validates our approach. As the number of processors are increased, the slowdown due to jitter during the compute phase starts worsening as the probability that one of the tasks gets hit by a high duration jitter starts increasing.

This is only an initial validation of our approach as we

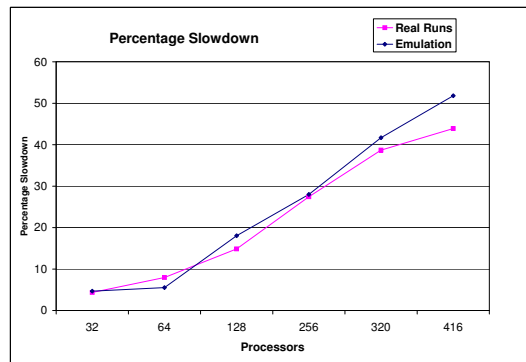


Fig. 8. Validation of slowdown prediction by comparing emulation results with data from a real cluster

validate only the slowdown during the compute phase. We are currently in the process of validating our approach on a real cluster with a high speed interconnect, where we can validate the slowdown both during the compute and the barrier phase. Another interesting point to note is that the slowdown experienced by the parallel benchmark in the compute phase due to jitter may be modeled without doing actual barriers. We are planning to extend our current jitter emulation framework to do emulation without barriers (*e.g.*, by removing the barrier calls in lines 3 and 7 in Algorithm 1 and recording just *qt*), which can help us do emulation on a much larger processor count than the actual physically available BlueGene processors. This can be used to study the slowdown in the compute phase due to OS jitter at a much larger task count.

VII. RELATED WORK

Several studies conducted on real large scale HPC systems have demonstrated the debilitating effects of OS jitter on scaling of collectives. One of the earliest works that reported the phenomenon of OS interference on parallel program performance was conducted by Petrini et al. [2]. They observed nearly 100% performance degradation at 4096 processors on ASCI Q. A more recent work compares the performance of three leading supercomputers: Blue Gene/L, Red Storm, and the ASCI Purple by conducting various experiments on these systems [6]. While these studies provide useful insights into the impact of OS jitter on parallel program performance, they require access to these real systems.

There also has been prior work to study the impact of jitter and predict its impact on scalability using an analytical framework [18] as well as simulation [12]. Since jitter characteristics on different operating systems and under various configurations can vary a great deal as shown in [11], it is non-trivial to generate accurate models. Although analytical studies, and simulation experiments provide the first step towards understanding the problem, emulation takes it closer to reality by replaying traces collected on real systems.

The work that can be considered closest to ours is by Beckman et al. [10]. While their approach is similar to ours, we improve upon that work on at least four counts. First, their

work suffers from the *14 μ s overhead limitation* (the overhead of *settimer* system call) on BlueGene/L - pitfall 1 mentioned in this paper, which does not allow them to introduce any jitter that is less than 14 μ s. This is a severe restriction as our analysis reveals that close to 98% of the jitter encountered on modern off-the-shelf Linux distributions is less than 14 μ s. Thus, their emulation results are not representative of real world Linux distributions. To overcome this limitation, we make use of a scaling technique that allows us to introduce even the smallest possible jitter. Second, they *make use of purely synthetic jitter*, while we improve upon the single node benchmark reported in [10] to collect traces from real Linux systems. We use these traces to do a trace driven emulation of jitter on Blue Gene/L rather than injecting purely synthetic jitter with hypothetical periodicity and duration. It appears that there is some work in progress in the ZeptoOS project [9] that attempts to record real jitter traces and replay them to explore system performance. Third, they *do not take into account the variance in sleep calls* and hence their emulation of synchronized jitter does not represent perfect synchronization. We overcome this limitation by resetting the trace index to the same randomly chosen value across all nodes in each compute phase instead of doing it just once at startup. This helps us emulate perfectly synchronized jitter. Finally, we *validate our approach by comparing the results from jitter emulation against slowdown numbers from a real cluster*.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we presented the design and implementation of an emulation framework that can be used to predict scalability of large clusters in the presence of OS jitter. We pointed out the shortcomings of previous such emulation approaches that may lead to inaccuracies in jitter emulation and presented innovative techniques that should be used to overcome these shortcomings. One of these techniques included scaling up all jitter values (and the runtimes of the parallel application) and scaling down the final results so as to allow the jitter emulator to accurately emulate even the smallest possible jitter. The framework relies on trace-driven emulation of OS jitter on a jitter-free platform such as Blue Gene/L. It uses benchmarks from our earlier work. We validated our jitter emulation technique on a single node by comparing the jitter profiles from an Intel machine running Linux with that of a Blue Gene/L node with jitter emulator. We also validated the scalability prediction results using jitter emulation with data from a real cluster. The results match closely and this validates our approach.

We demonstrated the application of this emulation framework to study OS jitter through a comparative scalability study of an off-the-shelf Linux distribution with a minimal configuration (runlevel 1) and a highly optimized embedded Linux distribution, running on the IO nodes of BlueGene/L. Our results indicate that an optimized OS along with a technique to synchronize jitter can reduce the performance degradation due to jitter from 99% (in case of off-the-shelf Linux without any synchronization) to a much more tolerable

level of 6% (in case of highly optimized BlueGene/L IO node Linux with synchronization) at 2048 processors. Furthermore, perfect synchronization can give linear scaling with less than 1% slowdown, regardless of the type of OS used. However, as the nodes start getting desynchronized with respect to each other, even with a minor skew across nodes, the optimized OS starts outperforming the off-the-shelf OS.

In our future work, we intend to use the jitter emulator framework to study scalability behavior of clusters running real time kernels as well those using virtualized platforms. We are also working on some coscheduling techniques to mitigate OS jitter and we will make use of the emulation framework to evaluate its efficacy. We are also planning to use real parallel applications and other publicly available parallel benchmarks and run them with our emulation framework to predict their scalability in presence of OS jitter. Scaling up the compute phase of some of these real applications might be non trivial.

REFERENCES

- [1] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "ACM SIGOPS Operating Systems Review," in *Operating System Issues for Petascale Systems*, 2006.
- [2] 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.
- [3] T. B. G. Team, "An Overview of the Blue Gene/L Supercomputer," in *ACM Supercomputing*, 2002.
- [4] "Cray XT4 - <http://www.cray.com/products/xt4/>." [Online]. Available: <http://www.cray.com/products/xt4/>
- [5] "Top 10 Supercomputers circa Jun 2007 - <http://www.top500.org/lists/2007/06/>." [Online]. Available: <http://www.top500.org/lists/2007/06/>
- [6] A. Hoisie, G. Johnson, D. Kerbyson, M. Lang, and S. Pakin, "A Performance Comparison through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/L, Red Storm, and Purple," in *ACM Supercomputing*, 2006.
- [7] "Right-weight Linux Kernel Project at Los Alamos National Laboratory." [Online]. Available: <http://public.lanl.gov/cluster/projects/index.html>
- [8] L. S. Kaplan, "Lightweight Linux for High-Performance Computing," in *LinuxWorld.com*, December 2006. [Online]. Available: <http://www.linuxworld.com/news/2006/120406-lightweight-linux.html>
- [9] "Zeptoos: The small linux for big computers." [Online]. Available: <http://www-unix.mcs.anl.gov/zeptoos/>
- [10] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale," in *IEEE International Conference on Cluster Computing*, 2006.
- [11] P. De, R. Kothari, and V. Mann, "Identifying Sources of Operating System Jitter Through Fine-Grained Kernel Instrumentation," in *IEEE Cluster*, 2007.
- [12] P. De and R. Garg, "The Impact of Noise on the Scaling of Collectives: An Empirical Evaluation," in *High Performance Computing (HiPC)*, 2006.
- [13] J. Moreira, M. Brutman, J. Castanos, and T. Engelsiepen, "Designing a Highly-Scalable Operating System: The Blue Gene/L Story," in *ACM Supercomputing*, 2006.
- [14] E. Parzen, "On the estimation of a probability density function and the mode," in *Annals of Math. Stats., Vol. 33, pp. 1065-1076*, 1962.
- [15] "Blue Gene at Lawrence Livermore National Laboratory." [Online]. Available: https://asc.llnl.gov/computing_resources/bluegene/
- [16] "Kullback-Leibler Divergence." [Online]. Available: http://en.wikipedia.org/wiki/Kullback-Leibler_divergence
- [17] 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.
- [18] S. Agarwal, R. Garg, and N. K. Vishnoi, "The Impact of Noise on the Scaling of Collectives," in *High Performance Computing (HiPC)*, 2005.