Benchmarking the Effects of Operating System Interference on Extreme-Scale Parallel Machines*

Pete Beckman[1] Kamil Iskra[1] Kazutomo Yoshii[1] Susan Coghlan[1]
Aroon Nataraj[2]

[1]Argonne National Laboratory
Mathematics and Computer Science Division
9700 South Cass Avenue
Argonne, IL 60439, USA

[2]Department of Computer and Information Science
University of Oregon
Eugene, OR 97403, USA

{beckman,iskra,kazutomo,smc}@mcs.anl.gov
anataraj@cs.uoregon.edu

Abstract

We investigate operating system noise, which we identify as one of the main reasons for a lack of synchronicity in parallel applications. Using a microbenchmark, we measure the noise on several contemporary platforms and find that, even with a general-purpose operating system, noise can be limited if certain precautions are taken. We then inject artificially generated noise into a massively parallel system and measure its influence on the performance of collective operations. Our experiments indicate that on extreme-scale platforms, the performance is correlated with the largest interruption to the application, even if the probability of such an interruption on a single process is extremely small. We demonstrate that synchronizing the noise can significantly reduce its negative influence.

Keywords: microbenchmark, noise, petascale, synchronicity

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

1 Introduction

The interaction between operating and run-time system components on massively parallel processing systems (MPPs) remains largely a mystery. While anecdotal evidence suggests that translation look-aside buffer (TLB) misses, interrupts, and asynchronous events can dramatically impact performance, the research community lacks a clear understanding of such behavior at scale and on real applications. Are there levels of operating system (OS) interaction that are acceptable? How significant is the performance difference between global collective operations, such as barriers and reductions, in the presence of OS interference? Are there thresholds that can be tolerated for some applications? Which? These and other questions remain largely unstudied as we search to build ever-larger petascale MPPs and Linux clusters [12]. However, answers to these questions are critical to the designs and computational models of future architectures. Understanding how operating systems interact with applications and how interrupts, process scheduling, and I/O processing affect performance on large-scale systems is key to petascale systems research.

Operating system interference is commonly referred to as "noise." Intuitively, noise is the collection of background activities that involuntarily interrupt the progress of the main application. In this paper, we use *noise* to refer to the overall phenomenon but choose the term *detour* when discussing any individual noise-comprising event, for example, when an application is temporarily suspended to process an OS-level interrupt.

Most asynchronous activities not initiated or managed from user space can clearly be identified as noise, for example, interrupts to update an internal OS kernel clock. Many would also put TLB misses in this category, but we raise an objection to this position. A TLB miss occurs when an address supplied to the CPU by the application cannot be resolved by using the entries in the TLB; page table entries (PTEs) must then be consulted, either by the CPU itself or by the OS exception handler. While in many practical situations the exact times of TLB misses are difficult to predict, they clearly depend on the application's behavior; that is, they are not strictly asynchronous—TLB misses take place if the application accesses a large number of memory pages. Because of this causal relationship, it is debatable whether considering TLB misses as system noise is useful. Except for TLB misses within the OS code, the focus of improvement should be on memory layout of the user code. The same is clearly true for cache misses. We do not consider them to be noise. Similar problems arise from uneven data partitioning. Some nodes will compute longer in iterations, perturbing the synchronization of the system. All of these issues are most strongly tied to the application, not the asynchronous behavior of the OS. For this paper we focus solely on the effects of OS noise outside users' control—the core scalability of an OS for petascale architectures.

2 Synchronicity

Several modes of cooperation exist between processes in parallel applications. An important one is the *lockstep mode*, where periodically all processes coordinate their progress by using collective operations ranging from simplest barriers to complex all-to-all message exchanges. Because all processes must take part in the collective operation, the overall speed is frequently reduced to that of the slowest process. Hence, maintaining *synchronicity* between the processes is vital; ideally, each process should take exactly the same amount of time to perform the operations between the collectives. With the collective invoked on all processes at precisely the same moment, nodes will not be left idle, waiting for the others to catch up. If just one process experiences a significant delay arriving at the collective, however, the entire operation can suffer, and all remaining nodes will sit idle [20]. Large-scale clusters and MPPs are especially prone to this behavior because of the large number of processes involved: the probabilities of delays are cumulative, eventually turning into a near certainty.

Several common events can trigger a detour from the application code, not all of which result in OS noise as defined earlier. Some events have little impact on synchronization, while others can cause dramatic delays. Table 1 provides an overview of detours on a 32-bit PowerPC box running the Linux 2.4 kernel.

Table 1: Overview of typical detours.

Source	**Magnitude**	**Example**
cache miss	100 ns	accessing next row of a C array
TLB miss	100 ns	accessing rarely used variable
HW interrupt	1 μs	network packet arrives
PTE miss	1 μs	accessing newly allocated memory
timer update	1 μs	process scheduler runs
page fault	10 μs	modifying a variable after `fork()`
swap in	10 ms	accessing load-on-demand data
pre-emption	10 ms	another process runs

Five of the entries from Table 1 are associated with memory access, indicating how complicated memory management can be across increasingly complex hierarchies [19] on an OS that supports paging and virtual memory for efficiency and flexibility.

The smallest disturbances come from cache misses. If the data is not in cache, a cache line is loaded from main memory. A memory access normally takes around 100 ns.

When an instruction attempts to access memory at a virtual address that the CPU does not know how to translate to the corresponding physical address, a TLB miss occurs. The miss can take several hundred nanoseconds, provided a corresponding PTE is available. Otherwise, the OS

exception handler must create a new PTE entry for the virtual address, a process that could take a few microseconds.

An OS exception handler is invoked if an attempt is made to access a memory location that is protected. This process need not indicate an error; optimizations such as copy-on-write are implemented by using this mechanism. A detour on the order of 10 μs is possible in this case. The detour will be much longer if the page data needs to be read from disk; the speed of the disk access (typically around 10 ms) is a limiting factor then.

Hardware interrupts normally have a higher priority than do application processes. Interrupts cause a handler to be invoked. Even though they are designed for speed, interrupt handlers take from a few microseconds to at most a few hundred to complete. If computationally expensive operations are required, a handler may trigger additional processes, which are scheduled at a convenient time after the handler has completed initial work. For example, a hardware interrupt handler of a network driver simply sends an acknowledgment to the hardware and schedules a task to handle the newly arrived network packet(s) at a later time.

Multitasking operating systems are usually based on recurring *ticks*. A timer interrupt is periodically raised, and the interrupt handler is invoked. Counters and timers are updated; when a process runs out of its time slice, another process is run. Typically, the timer interval is in the range of 1 to 10 ms. The interrupt handler itself usually consumes several microseconds.

Obviously, the process scheduler can introduce long detours if the parallel application process is supplanted by some other process. A typical detour will then take at least 10 ms—the time slice size—unless the newly scheduled process voluntarily vacates the processor. Therefore, rogue processes on a cluster, particularly those not I/O bound and so using the full time slice, can be a significant problem.

One class of detours absent from Table 1 because of its unpredictability is the lack of balance between individual application processes. Poor programming excluded, some problems are simply inherently difficult to balance properly (e.g., when the time needed to process data depends on the data itself). Even assuming that such a problem is properly balanced at startup, if processing the data alters it and if multiple iterations of the algorithm are required, periodic load redistribution will be required to maintain a good balance. The dependence between processing time and data may be clearly visible in the algorithm, but it may just as well be a subtle effect of, for instance, different memory access patterns employed on different processes, resulting in substantially different cache hit ratios.

For an extreme-scale cluster, only some of these detours will ultimately lead to a dramatic desynchronization of parallel operations. Unsynchronized noise creates a problem, as its effects increase with an increasing number of processes. Even very long detours—in the range of several milliseconds—have little overall effect as long as they occur at the same time on all processors [16].

At the other end of the scale, exceedingly short detours, such as cache misses, take an order of magnitude less time than the fastest collective operations. They do not contribute significantly to desynchronization if their frequency is similar on all processes.

Lightweight kernels optimized for compute nodes, such as IBM Blue Gene/L's (BG/L) BLRTS [13] or Cray XT3's Catamount [11], try to avoid many of these detours through a simplified architecture, for example, by not supporting general-purpose multitasking. From the entries in Table 1, cache misses are the only ones that will certainly occur; on some architectures TLB misses also cannot be avoided (depending on the amount of memory addressed by the CPU, the maximum page size, and the TLB size). Some hardware interrupts and timer updates are also possible, but in a far more limited number than in a general-purpose OS.

3 Noise Measurements

To explore the effect of noise on extreme-scale machines, we begin by gathering real benchmarks from existing platforms. In this section, we describe experiments conducted to measure the inherent noise of several operating systems.

3.1 Accurate Time Measurement

Since detours can be very short, careful benchmarking is critical. Measuring cache or TLB misses is outside our scope of interest; still, in order to measure hardware interrupts, a clock-time measurement function with a submicrosecond precision is required. Thus, the commonly used POSIX `gettimeofday()` system call is not quite good enough: even if its precision matches its resolution (which is not guaranteed), it will still have a precision of only 1 μs. Besides, as we will show later, on some systems invoking it takes several microseconds, simply because of the system call overhead.

Most CPUs provide a precise CPU timer that can usually be read by using just a few assembly instructions, so it only takes some 10 ns to 100 ns to obtain a new value. The timer is synchronized with the CPU clock. The updating frequency is either the same as the CPU frequency (thus, the precision will be 1 ns on a 1 GHz CPU), or it equals the *timebase*, which is lower than the CPU frequency by a fixed factor. In the latter case, the precision will be somewhat lower but still well under a microsecond on any modern CPU. So long as power-saving variable clock frequency capabilities are not enabled, measurements will be accurate.

The overhead of reading the timer is CPU specific. The counter itself is usually 64 bit, so, at least on architectures with 32-bit registers, an implicit or explicit atomic operation may be required to obtain a consistent reading. Table 2 shows the overhead of reading the timer, and, for compari-

Table 2: Overhead of reading the CPU timer and of calling `gettimeofday()`. Experiments were conducted in April 2006.

Platform	CPU	OS	CPU Timer [μs]	`gettimeofday()` [μs]
BG/L CN	PPC 440 (700 MHz)	BLRTS	0.024	3.242
BG/L ION	PPC 440 (700 MHz)	Linux 2.6	0.024	0.465
Laptop	Pentium-M (1.7 GHz)	Linux 2.6	0.027	3.020

son, the overhead of calling `gettimeofday()`, on a BG/L compute node (CN) and an I/O node (ION), as well as on an x86 Linux laptop. As the table shows, using the CPU timer is easily one to two orders of magnitude less expensive than calling `gettimeofday()`, in addition to providing a more accurate result.

3.2 Noise Measurement Technique

To measure noise, we use a benchmark loop as shown in Figure 1. This loop detects detours and stores information about them in an array for later processing. It will finish when the recording array gets full; on a busy system, this situation will take place almost immediately, because of the frequency of context switches. On the other hand, this loop can iterate for a long time on a virtually noiseless system such as the BG/L compute node OS.

```
cnt=0;
min_ticks=INFINITY;
current=rdtsc();

while(cnt<N) {
  prev=current;      /* keep the previous timer value  */
  current=rdtsc();   /* obtain the current timer value */

  td=current-prev;
  if(td>threshold) {
    detour[cnt++]=prev;
    detour[cnt++]=current;
  }
  if(td<min_ticks) min_ticks=td;
}
```

Figure 1: Acquisition loop of the noise measurement benchmark.

In the acquisition loop, the current timer value is repeatedly sampled (using a custom `rdtsc()` function) at a very high rate. If the code is allowed to run undisturbed, the sampling will essentially be periodic, since the same set of instructions is executed in each iteration. Randomly occurring detours bring disturbances into that process; these are determined by simply subtracting the timer value obtained during the previous iteration from the current one. We record the start and end time of each detour. Since for this set of experiments cache, TLB, and other memory effects are not considered, the benchmark loop does not exercise the memory. Instead, it correctly measures the interruptions forced by the OS when the application is quiescent. The threshold level used

for this benchmark was 1 μs. For modern machines, an ordinary interrupt handler takes several microseconds (see Table 1).

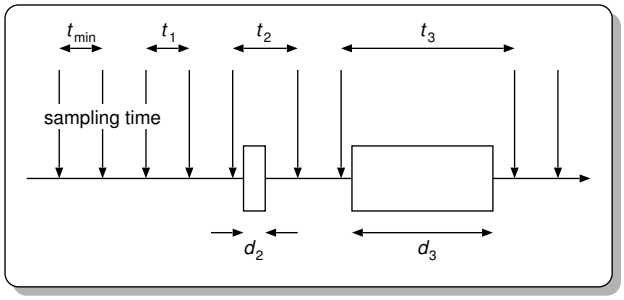

Figure 2: A sample of detours.

Figure 2 shows how the benchmark (Figure 1) regularly samples the clock until interrupted by a detour. The vertical arrows pointing downward represent sampling points; empty rectangles are the detours. Three cases are shown:

1. No detour occurs, so t_1 equals $t_{\min}$ (which is the final value of `min_ticks` from Figure 1).
2. A short detour of length d_2 takes place. The intersample period t_2 is approximately equal to $t_{\min} + d_2$ (it may be slightly larger because executing the detour code may flush the acquisition loop from the CPU cache). Here t_2 is below the threshold, so the detour will not be recorded.
3. A longer detour of length d_3 takes place. This time $t_3 \simeq t_{\min} + d_3$ is above the threshold, so the detour will be recorded.

The minimum iteration time $t_{\min}$ is important because it determines the maximum resolution of the benchmark. A sample of the results captured on several platforms can be found in Table 3. The results clearly indicate that all sampled architectures are capable of instrumenting 1 μs events. The exact $t_{\min}$ values depend on the CPU frequency, but also on other factors such as the quality of the branch prediction and compiler optimization. Furthermore, the OS can set memory page attributes, such as cache inhibit or page guard on pages where the loop resides. If so, the minimum iteration time will be different between two platforms even if the underlying hardware is the same—this effect can be observed on BG/L. The vastly superior timer resolution of the AMD Opteron CPU in Cray XT3 can be attributed to its 64-bit extensions: most operations in the loop are performed on 64-bit integers, and the other platforms, featuring 32-bit CPUs, must implement those in software.

This noise measurement technique is not without limitations. It is meant for identifying inherent noise only: the system is expected to be idle, and the benchmark itself is small and simple enough to generate no user-triggered detours when running. It will not measure any memory management overhead or detours stemming from processing MPI messages in the background as they arrive from a communication link. For example, arrivals of relatively short (1–2 KB) TCP/IP messages

Table 3: Minimum acquisition loop iteration times. Most experiments were conducted in May 2005, XT3 in Aug. 2005.

Platform	CPU	OS	t_{min} [ns]
BG/L CN	PPC 440 (700 MHz)	BLRTS	185
BG/L ION	PPC 440 (700 MHz)	Linux 2.4	137
Jazz node	Xeon (2.4 GHz)	Linux 2.4	62
Laptop	Pentium-M (1.7 GHz)	Linux 2.6	39
XT3	Opteron (2.4 GHz)	Catamount	7

Table 4: Statistical overview of the results. Most experiments were conducted in May 2005, XT3 in Aug. 2005.

Platform	Noise ratio [%]	Max detour [μs]	Mean detour [μs]	Median detour [μs]
BG/L CN	0.000029	1.8	1.8	1.8
BG/L ION	0.02	5.9	2.0	1.9
Jazz node	0.12	109.7	6.2	8.5
Laptop	1.02	180.0	9.5	7.0
XT3	0.002	9.5	2.1	1.2

results in the kernel taking detours of 10–20 μs.

3.3 Noise Measurement Results

We have applied our noise measurement technique to several different platforms. Table 4 presents a statistical analysis overview of the results obtained. Figures 3 to 5 provide a closer look at the actual data. Within the figures, plots on the left are time series graphs: the x axis denotes the execution time since the start of the benchmark, the y axis the length of a detour that took place at that time (if any). These plots give a good idea of the noise pattern. Plots on the right also provide the length of the detours on the y axis, but the x axis is sorted by the detour length, providing a better overview of the percentage of detours of a particular length.

Looking at the results in Table 4, we see that the noise ratio can vary widely between the platforms. The differences in the maximum detour length observed, while also large, are comparatively much smaller. The mean and median are relatively close to each other, indicating that the noise distributions lack extremely long detours. Our claim, which we further discuss in Section 5, is that the performance of extreme-scale parallel applications is affected mostly by the longest detours observed, and not by the noise ratio.

The data gathered from the compute node of IBM BG/L stands out from its peers (Fig. 3, top). As of this writing, BG/L is the largest MPP architecture available. The maximum detour is more than three times less than the other platforms. The system is virtually noiseless. The only periodic interrupt is a decrement timer: because the decrement register is a 32-bit integer, it would

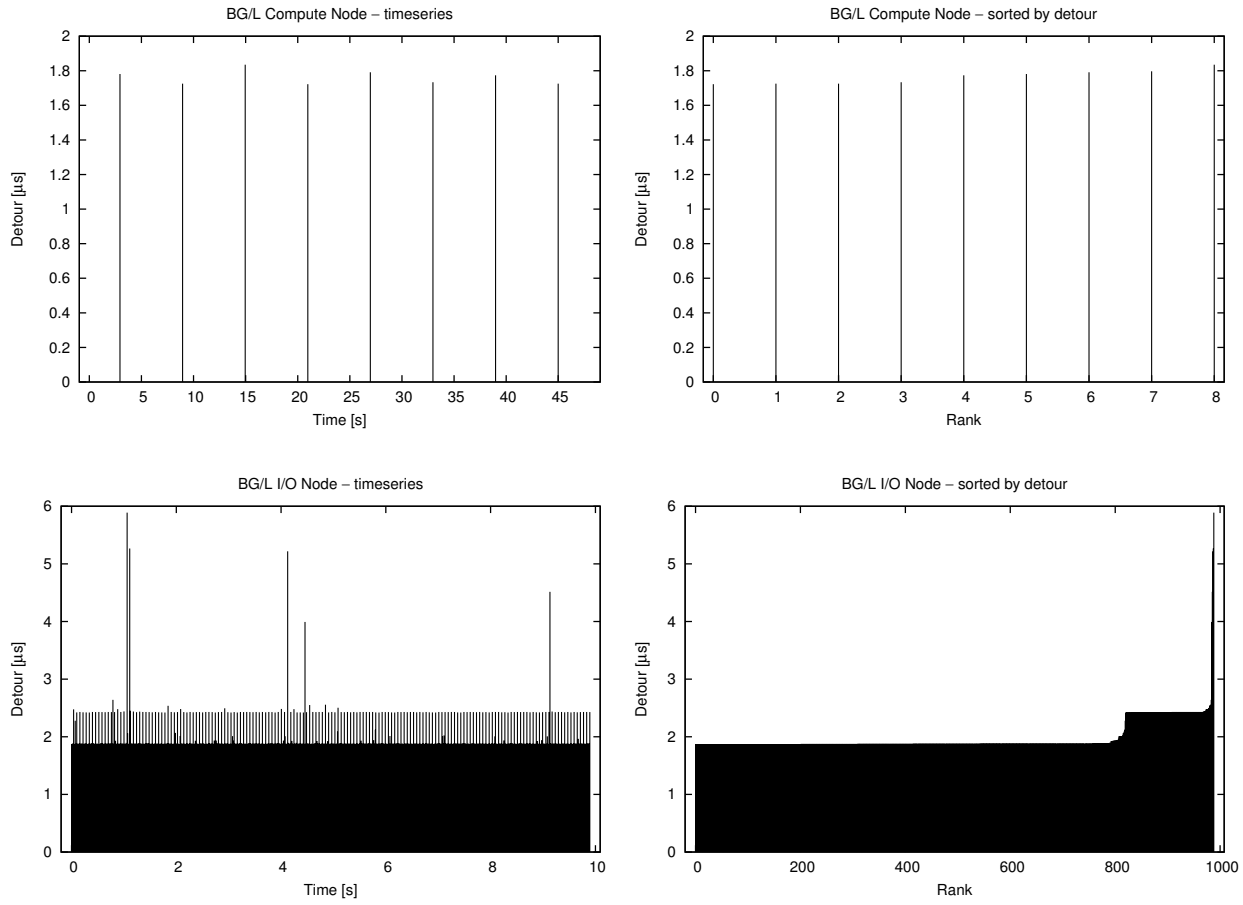

Figure 3: Noise measurements on BG/L: compute node (*top*) and I/O node (*bottom*).

underflow after approx. 6.1 s ($2^{32}/700\,\text{MHz}$), so it gets reset in an interrupt handler every 6 s. On BG/L, however, even that interrupt is automatically removed when the user code does not call user-level timers.

It is interesting to compare this with the data obtained on BG/L I/O node (Fig. 3, bottom), as the two platforms have identical CPUs, so the differences can be attributed squarely to the operating systems used: a specialized lightweight kernel on compute nodes and an embedded Linux on I/O nodes. From the data, three types of behavior can be observed. First, 80% of the detours are 1.8 μs and correspond to a Linux timer update scheduled for every 10 ms. Second, 16% are slightly longer, approximately 2.4 μs, because on every sixth timer interrupt the process scheduler is run. Third, a handful of detours are between 3 and 6 μs.

Compared to other platforms, the detours from BG/L I/O node Linux are actually quite short. Jazz (Fig. 4, top) is a relatively standard commodity Linux cluster. In spite of far more capable CPUs, the maximum detour length is more than an order of magnitude larger. The difference between a specialized lightweight kernel and an optimized embedded Linux kernel is far less than

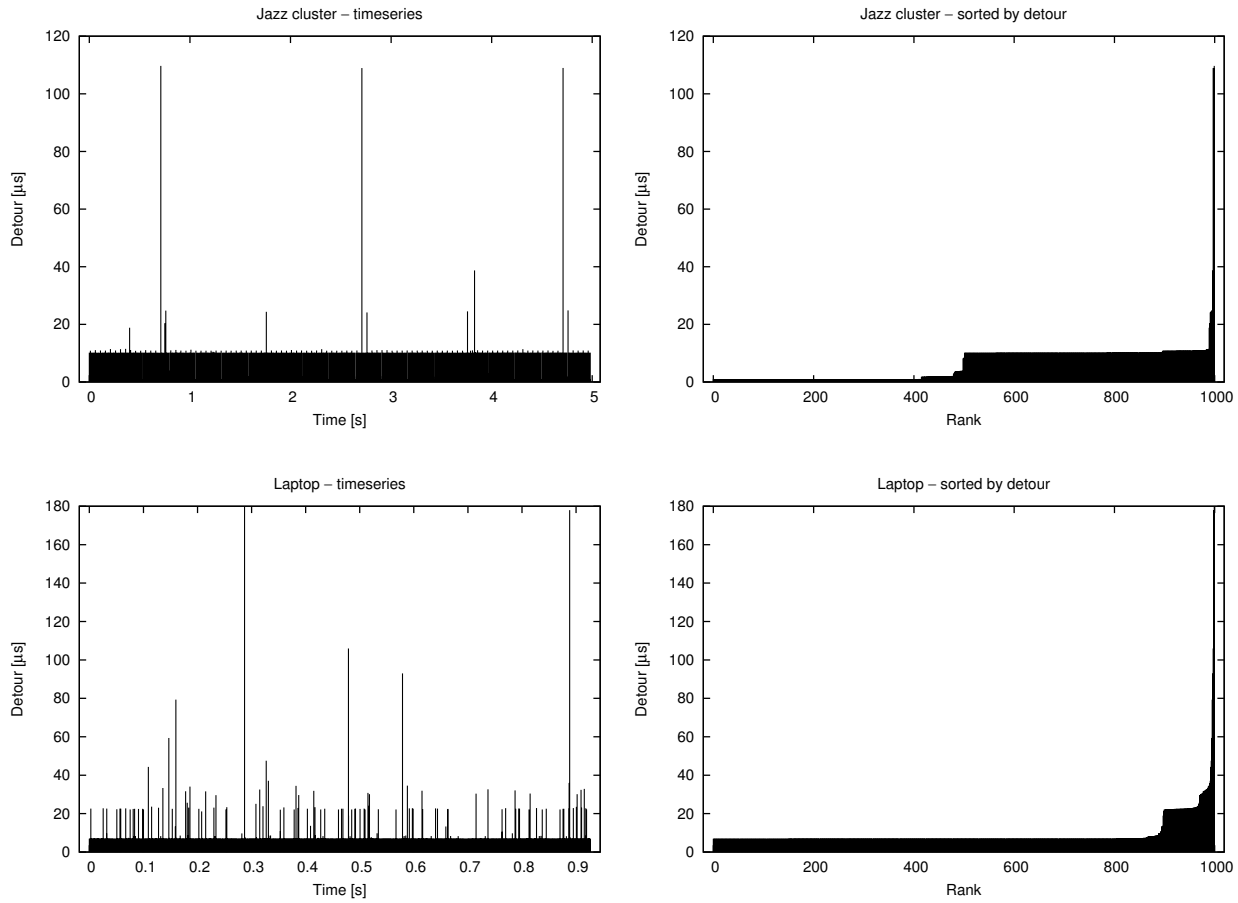

Figure 4: Noise measurements on Linux platforms: Jazz cluster node (*top*) and a laptop (*bottom*).

the difference between two different Linux systems. The kernel is in fact the *least* responsible for these differences. BG/L I/O nodes run a fairly standard embedded Linux kernel, without sophisticated low-latency patches. The dramatic difference stems from other processes run on these Linux platforms. BG/L I/O node Linux is trim; Jazz, on the other hand, even though optimized for cluster computation, maintains detour-causing background processes that perform job management (startup, termination) and monitoring tasks. Often, these extra processes are mistakenly included while discussing the noise native to an operating system, instead of separating the noise inherent to the OS from the configuration of the system.

We can also compare the noise on BG/L with that measured on another lightweight MPP kernel: Catamount, running on the compute nodes of Cray's large-scale MPP XT3 systems (Fig. 5). The noise ratio (Table 4) is clearly superior to any of the Linux platforms but is still much higher than that of BLRTS running on BG/L compute nodes. In fact, the maximum and mean are slightly higher than on BG/L I/O nodes running Linux. The median, on the other hand, is the lowest of all platforms tested, indicating that while XT3 is far from being noiseless, its detours are generally

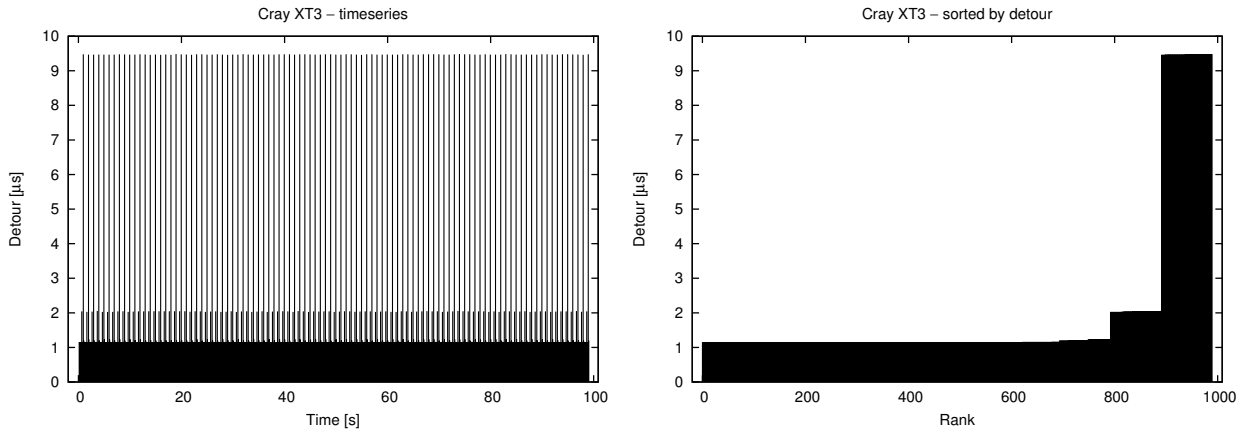

Figure 5: Noise measurements on XT3 compute node.

short. It is difficult to attribute all of the differences to kernel design. The 64-bit AMD is significantly faster than the PPC at the heart of BG/L. Until a portable lightweight kernel can be run on both BG/L and x86 hardware, exact comparisons cannot be made.

4 Noise Injection

The extremely low inherent noise of the BG/L compute node kernel makes it suitable as a test harness for injecting artificial noise and measuring its influence on application performance. Beginning with the nearly noiseless operation of BG/L, we inject noise and explore the impact on applications that require synchronous behavior. This strategy should allow us to get an impression of how such applications would perform if they ran on top of a noisier kernel—performing such experiments in reality was infeasible for us because of a lack of sufficiently massive parallel machines that support kernels other than BLRTS.

The operations most sensitive to desynchronizing detours are collectives. With many of them, if even one of the CPUs is late to the collective operation, the entire operation will be delayed (this is the case with all three collectives we discuss below). For example, if only one of possibly thousands of nodes suspended the local application and scheduled a different process for a time slice, that single 10 ms detour on one CPU would suddenly stall the collective dramatically. On a machine such as BG/L, with some fast collectives taking just a few microseconds, such a misconfigured system would slow the collective operation by a factor of more than 1000.

To explore this behavior, we focused solely on the MPI collective operations expected to be highly sensitive to noise (barrier, allreduce, and alltoall) and injected random delays. The results presented below can thus be considered a *worst-case scenario*, as real-world applications perform collectives for only a fraction of their execution time.

A real-time interval timer was used to periodically force execution of a delay loop. We explored both synchronized and unsynchronized noise. In our implementation, the difference is only at initialization: with the unsynchronized injection, individual processes of a parallel job are delayed by a random interval before the first injection is scheduled. A barrier is performed before the benchmark measurements start, in order to synchronize the execution progress of the processes; no further explicit (de)synchronizations are performed.

Figure 6 presents the results of several collective operations. Each collective was tested in configurations ranging from a single midplane (512 nodes) to 16 racks (16,384 nodes) on the IBM T. J. Watson Research Center 20-rack BG/L "BGW" system in Oct. 2005, as of this writing the third fastest computer in the world [17]. The results shown are for experiments performed in *virtual node mode*, in other words, when both CPU cores on each node are occupied by application processes. We injected noise at frequencies ranging from 10 Hz (interval 100 ms) to 1 kHz (interval 1 ms). The minimum detour injected was 16 μs—the overhead of the interval timer used. We tried several larger values; in addition to 16 μs, Figure 6 shows results for 50, 100, and 200 μs. In general, we found the performance of the noise-free experiments to be almost identical to that of the experiments with synchronized 16 μs noise at 100 ms intervals; we have thus omitted the former results from the plots in order to avoid clutter (see also Figure 7).

The results for the simplest *barrier* can be found at the top of Figure 6. Barriers on BG/L are implemented by using a dedicated *global interrupt* network, providing excellent performance. As can be observed, synchronized noise (Fig. 6, top left) only slightly affects the performance—by 26% in the worst case. Unsynchronized noise (Fig. 6, top right) presents more of a challenge—execution time increases by up to a staggering factor of 268. However, that statement alone does not tell the full story. While the absolute increase in execution time is important, more interesting is the relationship between performance and detour length. As can be observed, that relation is mostly linear, and it saturates at twice the time length of a detour (check the curve for interval 1 ms). As far as we know, barriers in virtual node mode are implemented by first synchronizing the two processes running on the same node and then synchronizing all nodes over the network. Each of these steps can be slowed by as much as a single detour time, but no more than that, simply because nodes execute each step independently, in parallel. Interesting, there appears to be another saturation point at the level equal to a single detour length (check the curve for interval 100 ms). The relationship between the execution time and the node count seems to be nonlinear, at least for high injection intervals: there is a critical value of parameters, where a phase transition takes place between a very efficient execution largely unaffected by noise, and a less efficient one, where the effect of noise is linear (note that the node count axis is logarithmic; the effect is even more apparent in a linear scale). We stress that we do not see any prohibitive, superlinear execution time growth related to the size of the machine.

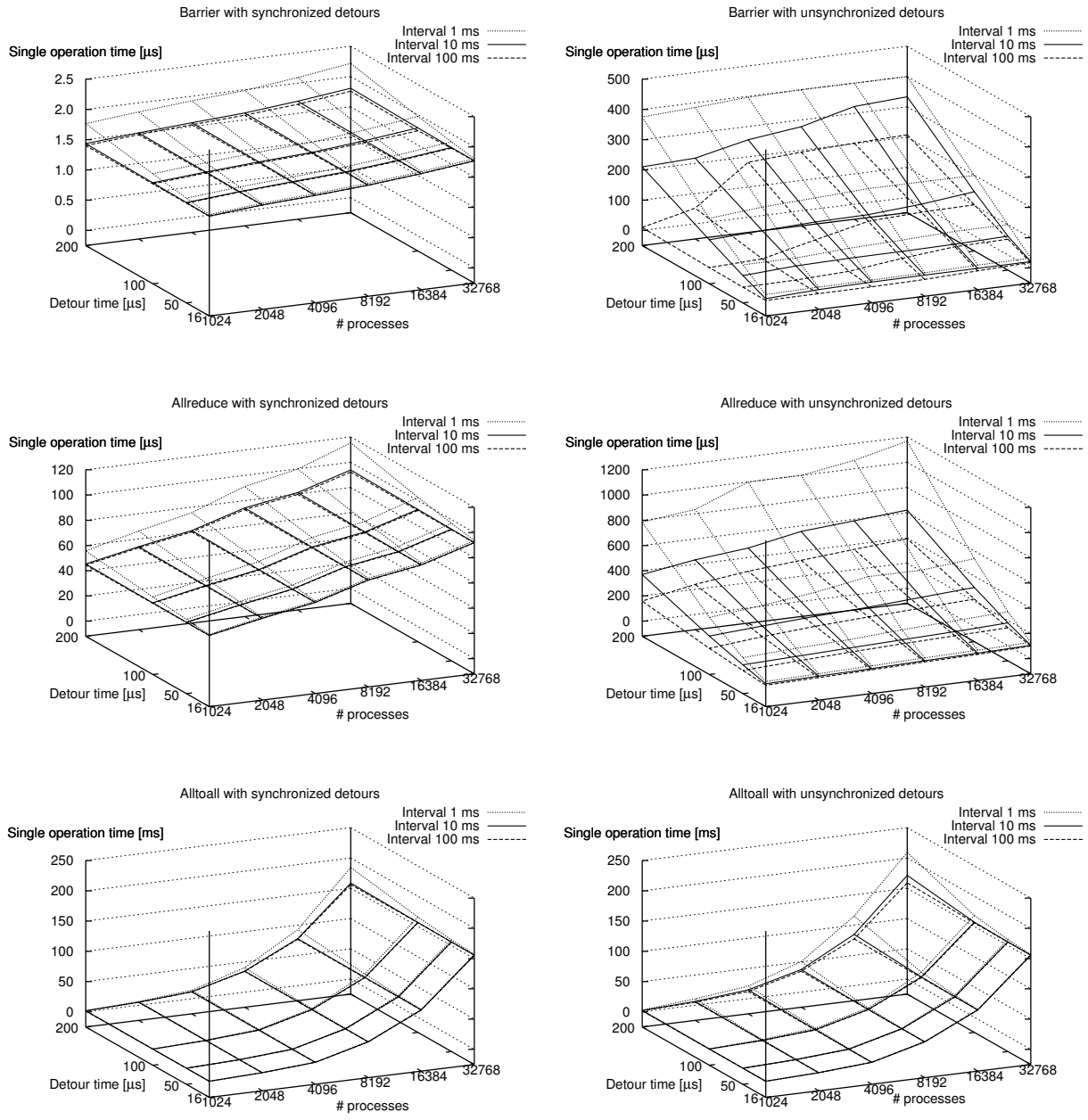

Figure 6: Performance of collective operations barrier (*top*), allreduce (*middle*), and alltoall (*bottom*) in the presence of artificially injected noise, synchronized (*left*) and unsynchronized (*right*).

The second collective operation tested was *allreduce*. There are at least two sorts of reduction operations on BG/L. Certain simple cases can be handled by the network hardware; others require cooperation with the message layer code linked with the application, or even of the application code itself. The results shown here are for the latter case, as noise has a more interesting influence then. Allreduce with a synchronized noise (Fig. 6, middle left) behaves quite similarly to a barrier,

only the logarithmic complexity of the operation in respect to the number of processes is more apparent. The behavior with unsynchronized noise (Fig. 6, middle right) is different. Depending on perspective, it can be characterized as either less susceptible to noise than is the performance of barriers (execution time increase by at most a factor of 18) or worse overall (the increase observed is over 1000 μs). The larger degree of cooperation required from the application processes by the reduction operation means that there are more opportunities for noise to influence the performance. As the algorithm is logarithmic, the maximum slowdown is not fixed as it was with barriers, but also increases logarithmically with the number of processes. As with barriers, execution time is mostly linear in relation to detour length, although a careful observer will note a slight superlinearity. We attribute that behavior to the fact that as the detour length increases, the percentage of CPU cycles left to the application decreases, and that effect is nonlinear.

The last collective operation we tried was *alltoall* (Fig. 6, bottom). Unlike the previous two, it has a linear complexity with respect to the number of nodes, so on a massively parallel machine like BG/L its performance leaves something to be desired: we had to label the z axis in milliseconds to fit the plots. Noise injection has a comparatively minor influence on the performance, possibly because the detours we inject are too fine-grained compared to the collective itself. Part of the reason might also be that alltoall has a higher degree of parallelism than do the other collectives tried, so occasional detours do not stall the whole operation; another part of the explanation is that at this scale, alltoall on BG/L saturates communication links, forcing processes to stall [2]. This situation is confirmed by the results with synchronized noise, which were the only tests where the slowdown was smaller than the percentage of CPU cycles taken away by noise—performance will not be affected if a process has nothing to do anyway at the time a detour occurs. With unsynchronized noise, the slowdown we observe ranges from 173% for 1,024 processes to 34% for 32,768 processes, although in absolute terms the latter is the highest, reaching around 53 ms. The superlinear increase with the detour length is more pronounced with alltoall (see Fig. 6, bottom right) than with other collectives. However, we point out that the noise ratio where that happens is very high: there is a detour of 200 μs every 1000 μs. This is more like a cacophony than noise, and hence affects the performance in that way.

Figure 7 presents some of the same data as Figure 6, only this time we have put the results with synchronized and unsynchronized noise in the same plot, to allow for an easier comparison. We are showing the results of allreduce with 16 μs detours injected every 1 ms, which produces a fairly realistic 1.6% noise ratio. The vanilla, noise-free results are also included. As can be observed, there is very little difference between the results with synchronized noise and the noise-free ones, proving the effectiveness of synchronization. Unsynchronized noise at this noise ratio adds a fairly constant overhead of around 35 μs per collective operation.

In addition to the experiments in virtual node mode, we performed analogous ones in *copro-*

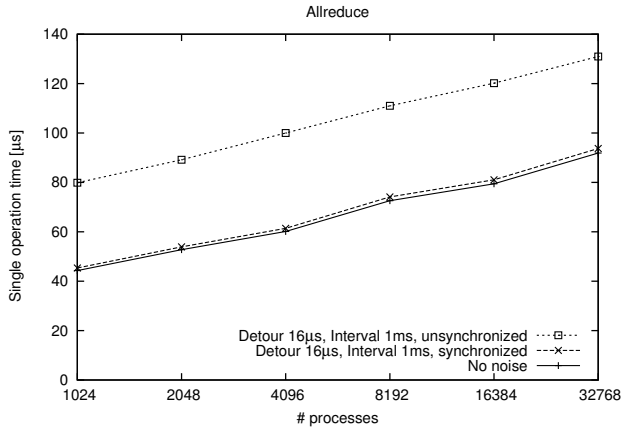

Figure 7: Performance of allreduce with 1.6% noise ratio.

cessor mode, that is, when only one application process per node is run and some message-passing services are offloaded onto the second core. One might expect that this separation would make coprocessor mode far more insensitive to noise. Experiments have shown, however, that the influence of noise is similar irrespective of the execution mode; presumably that is the case because even in coprocessor mode the bulk of communication-related operations are performed by the main CPU core.

5 Noise Distributions

The results presented so far suggest that coarse-grained noise is more detrimental to an application's performance than is fine-grained noise. To verify this hypothesis, in Oct. 2006 we ran a number of experiments where we kept the per-process percentage of the injected noise constant but varied the noise distribution. Figure 8 presents a sample of the results.

In these experiments we injected noise in amounts meant to more closely correspond to reality, based on earlier results presented in Table 4. As in Figure 6, we have the number of processes on the x axis and the detour length on y (to keep the noise percentage constant, we varied other injection parameters, such as the interval, when adjusting the detour length). This time, on the z axis we show execution time magnification, which is simply a ratio of the average execution time of a single benchmark iteration with and without noise. The benchmark in question is based on the one used in Section 4, only this time as a collective operation we used an allreduce that was accelerated by the BG/L collective network hardware.

Overall, the results clearly support our hypothesis: large detours cause far greater slowdowns than do smaller detours, even if the noise percentage is kept constant. With larger noise ratios, such as 1% (Fig. 8, bottom right), this is the case for all job sizes. However, with the smallest ratios,

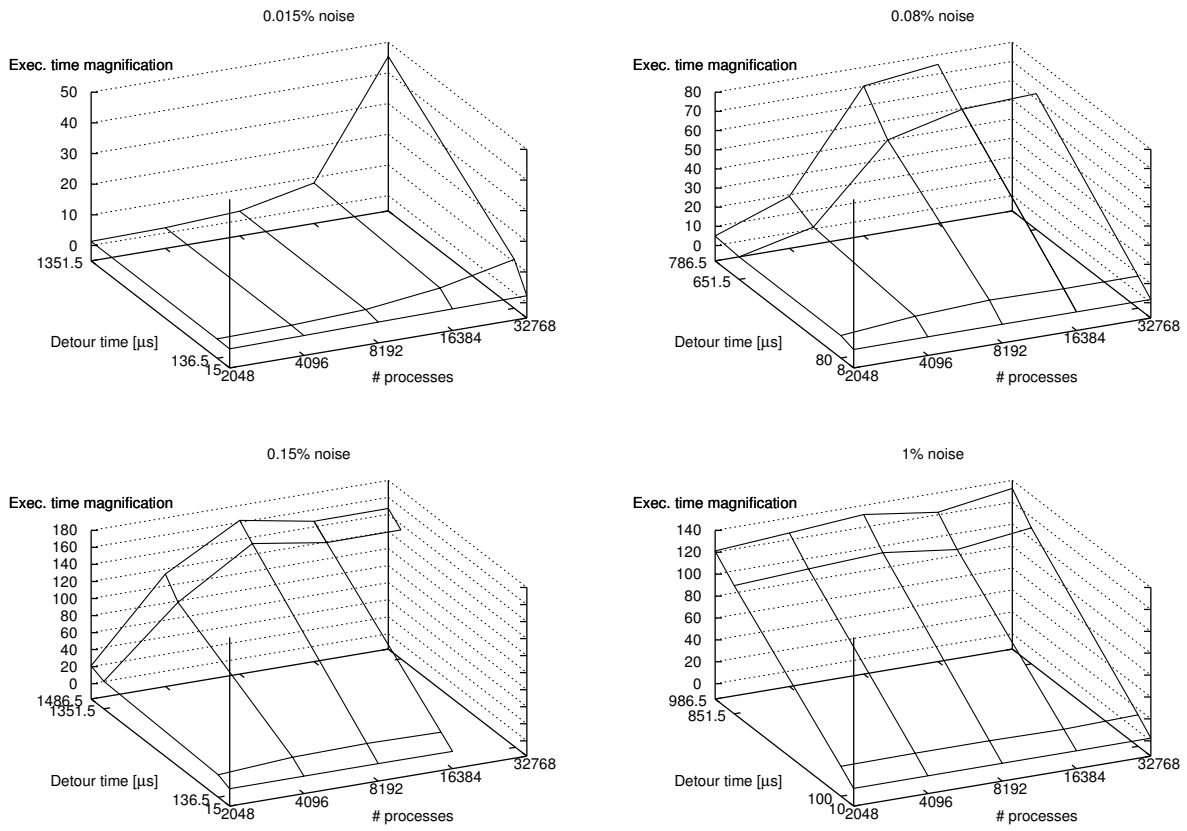

Figure 8: Execution time magnification when injecting constant amounts of noise into a benchmark based on allreduce (incomplete data sets on 32K nodes for noise ratios of 0.08% and 0.15% are due to time constraints in our access to such a large machine configuration).

in particular 0.015% (Fig. 8, top left), execution times increase significantly only for the largest process counts. More worrying, it appears as if, in that case, we will see the dreaded explosion in run-time as the number of processes increases. We have included two more plots with interim noise ratios of 0.08% and 0.15% to show that that is in fact not the case. As can be observed, execution time magnification does not exponentially grow *ad infinitum* with an increasing process count, but fairly quickly levels off or even slightly decreases. The larger the noise ratio, the sooner this effect takes place—for 1% noise, it is not visible even for the smallest process count tested. The leveling-off is due to noise saturation—with a fixed per-process noise ratio, the total amount of noise in the system increases linearly with an increasing process count. It reaches a stage when at any given point in time, a detour is almost guaranteed to be taking place somewhere—adding more processes, and thus more detours, will simply force the latter to overlap in time, significantly reducing their ability to do any (further) damage.

6 Discussion

Operating system noise has been studied by others for several years, and the roots can be traced back to over a decade. The experiments described by Burger et al. [5], while focusing on another aspect of the problem (evaluation of gang scheduling and demand paging on massively parallel systems), employ techniques similar to what we used in Section 4: they inject artificial delays to simulate the effect of sharing CPUs with other processes. In a more recent study, Petrini et al. [14] found nonessential processes to be responsible for a significant slowdown of the ASCI Q machine and devised techniques to identify the sources of noise and eliminate them. However, because the difficulties stemmed from a misconfigured system running printer daemons and other nonessential processes, these results cannot be generalized to the nature of noise inherent in tick-based operating systems.

Several studies have shown that, on four-CPU SMP machines, the overall parallel job performance is better if one of the CPUs is left idle [10] so that it can handle the interrupts or other processes. The remaining CPUs can remain tightly synchronized. Coscheduling processes of a parallel application across the whole machine allowed Jones et al. [9] to reduce the execution time of collectives such as allreduce by a factor of 3 on a large IBM SP.

Sottile and Minnich [15] argue that microbenchmarks based on a *fixed work quantum* principle do not provide enough insight; they recommend using a *fixed time quantum* principle as an alternative, since it makes the results much easier to analyze with established techniques from signal processing and spectral analysis. The benchmark introduced in Section 3 samples the CPU timer as frequently as possible, performing a minimal, constant set of operations between the samples. This means that it works according to the fixed *work* quantum principle, since sample intervals will not be constant if detours occur. The fixed *time* quantum principle would be impractical in our case because the overhead of timer interrupts on BG/L is over 10 μs—much more than the shortest detours we are interested in. We still did our best to avoid cache effects by storing information only on detours that were above a predefined threshold.

In an initial theoretical study, Agarwal et al. [1] determined that noise can drastically reduce the performance of collective operations, but only with some noise distributions, such as heavy-tailed or Bernoulli. This work was followed with an experimental evaluation by Garg and De [7]. On the other hand, Petrini et al. [14] claim that, at least in case of fine-grained applications, short but frequent detours on all nodes are more detrimental to the performance than are long but less frequent ones on just a few nodes. They further claim that performance is affected most if noise *resonates* with an application, that is, if their granularities are similar. We believe that claim to be only partially true. Obviously, fine-grained noise will have little effect on a coarse-grained application, as it simply will not be able to desynchronize the processes in any significant way—

we could see that in case of the expensive alltoall collective (Fig. 6, bottom). However, we see no reason why coarse-grained noise should not affect a fine-grained application. On the contrary, its effects are likely to be devastating, as one could conclude from looking at the results of lightweight barriers (Fig. 6, top right). Essentially, even fairly infrequent detours become very likely with a rapidly increasing number of processes; once they are close to certain to occur, they dwarf all the shorter, but more frequent detours.

This phenomenon was confirmed by Tsafrir et al. [18]. Using a probabilistic model, they show that the impact of noise on a parallel job is linearly proportional to the number of nodes, but only if noise probability is small enough. Once the job exceeds a particular size, a detour is nearly certain to occur, and further increases in node count do not affect noise. This result confirms our findings from Section 4 regarding barriers. According to their model, for 100k nodes, one needs a per node noise probability no higher than 10^{-6} per phase (i.e., between two collectives) for a machinewide probability of a detour to be lower than .1. They identify fine-grained clock ticks to be a major source of overhead: even though 1 kHz ticks take no more than 1% of CPU time, they slow a microbenchmark used by at least 40%—on *one* node. Cache pollution due to an execution of kernel code is blamed for that, and eliminating ticks is the recommended solution, because synchronizing such frequent events on a massively parallel machine might be impractical.

Brightwell et al. [4] compared performance between Linux and a lightweight Cougar kernel (a predecessor of Catamount) on the ASCI Red machine. While we cannot draw broad conclusions from their comparison of Linux with an interrupt-driven TCP adapter to Cougar's far more efficient transport layer, the authors do provide many useful insights. They observe that not-massively-parallel code running under Linux can perform significantly better than on Cougar, probably because of improvements continuously made to the compilers and libraries by free software developers. Essentially, the overheads of maintaining a state-of-the-art software bundle are much higher for a niche product than for the mainstream. In a more recent comparison of Linux and Catamount, Hudson and Brightwell [8] conclude that even though the availability of CPU for applications running on an optimized Linux on Cray XT3 is slightly better than that of Catamount, its network performance as measured by a microbenchmark is several times worse, on as few as 28 nodes. While some slowdown with high-performance jobs is to be expected because of the more complex memory hierarchy of Linux (see Section 2), such a large difference is difficult to explain. With just a few dozen nodes, noise certainly should not be an issue.

As MPP platforms become more popular, they are put to new uses, requiring more capabilities from the kernel. For example, some BG/L users have been requesting support for dynamically loaded libraries so that tools such as Python can be used on the compute nodes. Basically, users want MPPs to be more like the systems they are used to; it may be difficult to achieve that goal and at the same time maintain the performance advantage of lightweight kernels.

7 Conclusions

This paper focused on the synchronicity of processes in parallel scientific applications and the desynchronizing effects that can be introduced via the OS. We provided an overview of typical detours that can be attributed to contemporary computer architectures and general-purpose operating systems. We pointed out that so far as synchronicity is concerned, only some of those detours are actually relevant.

In Section 3, we used a microbenchmark to measure noise on several platforms. Our results, which are an extension of work published in [3], indicate that while specialized lightweight kernels have a clearly superior noise ratio, the average detour length among all platforms tested is of the same order of magnitude. Even a fairly standard Linux kernel can have a low maximum detour length, provided that the hardware it manages is fairly simple and the set of processes limited. With sophisticated low-latency patches or real-time enhancements [6], the differences in maximum detour length compared to lightweight kernels would likely be even smaller. The differences in noise ratio could be mostly eliminated with a move to a tickless kernel.

To get more insight into the effect of noise on synchronicity, in Section 4, we benchmarked the performance of several collective operations under various levels of artificially injected noise. While the slowdown in many cases is rather large, the experiments represent a worst-case scenario; a real-world application would perform collective operations far less frequently and thus would be affected far less. The slowdown is not cumulative in any significant way: we do not see an explosive growth in execution time, relative to either the number of nodes or the detour length.

The most significant result of this paper is that detours need to be quite large in order to significantly impact performance on extreme-scale architectures, as shown in Section 5. The detour times for the BG/L I/O node Linux were all less than 6 μs—without any special latency-reducing patches or other optimizations. The *minimum* noise we could artificially inject for BG/L was 16 μs, with the resulting data hardly distinguishable from the case where there was no noise at all. It is not until detours as long as 50 μs occur every 1 ms that any appreciable impact can be seen. This result strongly indicates that the noise from even tick-based operating systems with unsynchronized schedulers, such as Linux, would have little impact on overall system performance. However, a single rogue stealing an occasional timeslice could slow collectives by a factor of 1000. Clearly, impact is an issue of scale; it is dominated by the relationship between the absolute performance of collective operations to the longest unsynchronized detours in the system. For this reason, the noise within an extreme-scale Linux cluster may pose little real performance impact. Without the benefit of a lightning-fast global interrupt and tree-reduction networks, such as are available on BG/L, the noise introduced by the Linux kernel can be relatively small compared to that of collectives formed from point-to-point operations (even on BG/L we could see this effect for alltoall).

We believe that unless extra processes or interrupt processing dramatically desynchronizes a Linux cluster, OS noise does not cause significant performance degradation.

An idea we are currently testing is to mask interrupts (and, thus, most sources of noise) and enable them for only short periods. The intent is to coalesce frequent, fine-grained interrupts into large groups, potentially decreasing the overhead. One could conclude from our experiments that this would be a bad idea, as larger, less frequent detours are clearly more of an issue. However, that is true only with unsynchronized noise. The experiments also show what an improvement a simple initial synchronization of noise can bring, especially for more lightweight collectives; we see an execution time magnification of only a few percent then. Larger, less frequent detours are much easier to synchronize across the whole machine, so noise coalescence should be worthwhile. Thus, noise should not pose serious problems even on extreme-scale machines, as long as we can keep it synchronized. We point out, however, that BG/L has been designed with a single clock source driving all the devices in the whole, multirack machine. Synchronizing a machine based on commodity hardware, where each node has several independent time sources, might be more of a challenge.

All in all, we believe the data gathered confirms that running a general-purpose OS such as Linux on massively parallel machines should be viable and is definitely worth pursuing.

Acknowledgment: We gratefully acknowledge the IBM T. J. Watson Research Center for making it possible for us to use the BGW system to perform the large-scale experiments described in this paper.

References

[1] S. Agarwal, R. Garg, and N. K. Vishnoi. The impact of noise on the scaling of collectives: A theoretical approach. In *Proceedings of the 12th International Conference on High Performance Computing*, volume 3769 of *Springer Lecture Notes in Computer Science*, pages 280–289, Goa, India, Dec. 2005.

[2] G. Almási. Private communication, 2006.

[3] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. Operating system issues for petascale systems. *ACM SIGOPS Operating Systems Review*, 40(2):29–33, Apr. 2006.

[4] R. Brightwell, R. Riesen, K. Underwood, T. B. Hudson, P. Bridges, and A. B. Maccabe. A performance comparison of Linux and a lightweight kernel. In *Proceedings of the 5th IEEE International Conference on Cluster Computing*, Kowloon, Hong Kong, China, Dec. 2003.

[5] D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. *Journal of Supercomputing*, 10(1):87–104, Mar. 1996.

[6] S.-T. Dietrich and D. Walker. The evolution of real-time Linux, Nov. 2005. http://www.linuxdevices.com/files/rtlws-2005/SvenThorstenDietrich.pdf.

[7] R. Garg and P. De. Impact of noise on scaling of collectives: An empirical evaluation. In *Proceedings of the 13th International Conference on High Performance Computing*, volume 4297 of *Springer Lecture Notes in Computer Science*, pages 460–471, Bangalore, India, Dec. 2006.

[8] T. Hudson and R. Brightwell. Network performance impact of a lightweight Linux for Cray XT3 compute nodes. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Tampa, FL, Nov. 2006.

[9] 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 *Proceedings of the ACM/IEEE Conference on Supercomputing*, Phoenix, AZ, Nov. 2003.

[10] T. R. Jones, L. B. Brenner, and J. M. Fier. Impacts of operating systems on the scalability of parallel applications. Technical Report UCRL-MI-202629, Lawrence Livermore National Laboratory, Mar. 2003.

[11] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 47th Cray User Group Conference*, Albuquerque, NM, May 2005.

[12] W. Kramer and C. Ryan. Performance variability of highly parallel architectures. In *Proceedings of the International Conference on Computational Science*, volume 2659 of *Springer Lecture Notes in Computer Science*, Melbourne, Australia and St. Petersburg, Russia, June 2003.

[13] J. E. Moreira et al. Blue Gene/L programming and operating environment. *IBM Journal of Research and Development*, 49(2/3):367–376, Mar. 2005.

[14] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Phoenix, AZ, Nov. 2003.

[15] M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *Proceedings of the 6th IEEE International Conference on Cluster Computing*, pages 371–377, San Diego, CA, Sept. 2004.

[16] P. Terry, A. Shan, and P. Huttunen. Improving application performance on HPC systems with process synchronization. *Linux Journal*, 127:68–73, Nov. 2004.

[17] TOP500 Supercomputer Sites. http://www.top500.org/.

[18] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th International Conference on Supercomputing*, pages 303–312, Cambridge, MA, June 2005.

[19] R. van der Pas. Memory hierarchy in cache-based systems. Technical Report 817-0742-10, Sun Microsystems, Nov. 2002.

[20] A. Wagner, D. Buntinas, D. K. Panda, and R. Brightwell. Application-bypass reduction for large-scale clusters. In *Proceedings of the 5th IEEE International Conference on Cluster Computing*, pages 404–411, Kowloon, Hong Kong, China, Dec. 2003.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.