

Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection

Kurt B. Ferreira and Patrick Bridges
Computer Science Department
The University of New Mexico
Albuquerque, NM 87131
{kurt,bridges}@cs.unm.edu

Ron Brightwell
Scalable System Software Department
Sandia National Laboratories *
Albuquerque, NM 87185-1319
rbbrih@sandia.gov

Abstract

Operating system noise has been shown to be a key limiter of application scalability in high-end systems. While several studies have attempted to quantify the sources and effects of system interference using user-level mechanisms, there are few published studies on the effect of different kinds of kernel-generated noise on application performance at scale. In this paper, we examine the sensitivity of real-world, large-scale applications to a range of OS noise patterns using a kernel-based noise injection mechanism implemented in the Catamount lightweight kernel. Our results demonstrate the importance of *how* noise is generated, in terms of frequency and duration, and how this impact changes with application scale. For example, our results show that 2.5% net processor noise at 10,000 nodes can have no impact or can result in over a factor of 20 slowdown for the *same* application, depending solely on how the noise is generated. We also discuss how the characteristics of the applications we studied, for example computation/communication ratios, collective communication sizes, and other characteristics, related to their tendency to amplify or absorb noise. Finally, we discuss the implications of our findings on the design of new operating systems, middleware, and other system services for high-end parallel systems.

1 Introduction

Recent research has shown that operating system (OS) interference is a key limiter of application performance in large-scale systems [7, 13]. This performance impact is generally attributed to interference with synchronization between application instances on distributed hosts by operating system services and associated daemons. Attempts to limit this “noise” or “jitter” in commodity systems by, for example, gang-scheduling

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

timer interrupts and system daemons or by dedicating specific processors to OS tasks, have to this point required applications to pay sizable performance costs. Custom lightweight kernels that perform work only when explicitly requested to by the application—thereby virtually eliminating OS noise—have been shown to have better application scalability than commodity kernels [3], but at the cost of a limited set of available OS services.

Characterizing the effects OS noise on different large-scale applications is crucial to understanding the tradeoffs involved in the design and implementation of operating systems for large-scale systems. For example, OS designers need such information not only to make informed decisions about which services to implement, but also how to decompose and schedule these services to minimize their impact on application performance. This knowledge is even more important in light of current efforts that aggregate OS activities and when considering the increasing amount of work an OS is likely to perform as the core count in processors continues to increase. Understanding application sensitivity to OS noise is also important for helping determine the appropriate system software options for a given workload and system.

There is already evidence that previous studies are insufficient for these purposes. Previous research that has focused on micro-benchmarks and improved methods for measuring idle OS noise signatures in order to compare operating systems [15] has led some OS developers to pursue complex OS modifications that dramatically reduce idle noise signatures, but that in turn significantly degrade application performance. For example, Cray reduced the timer interrupt frequency of Linux well below normal to demonstrate an idle noise signature equivalent to a lightweight kernel [18], but application performance was so poor that this kernel was essentially unusable in a production environment [6]. Similarly, collective communication micro-benchmark performance has been an emphasis of several noise studies [16, 1], despite the fact that evidence has shown no correlation between the impact of noise on application performance and the impact of noise on collective communication performance as measured with micro-benchmarks [13].

In this study, we directly quantify the application performance costs of local operating system interference on a range of real-world large-scale applications using over ten thousand nodes. We have implemented a kernel-based noise injection framework in the Catamount lightweight compute-node kernel [8] and have used this framework to inject various levels of noise into applications running on the Sandia/Cray Red Storm system [4]. The main contributions of this paper are:

- presentation of a kernel-based noise injection framework
- a characterization of the importance of how noise is generated to application performance

- an analysis of application influences that lead to noise absorption or amplification

To our knowledge, this is the first such analysis of the impact of noise using a kernel-based injection approach using an OS that is inherently noiseless.

The remainder of this paper is organized as follows. The following section offers a definition of OS noise, describes sources of noise in high-performance systems, and discusses strategies that others have used to mitigate the impact of noise on applications. In Section 3, we describe our approach to injecting kernel-level noise, and continue in Section 4 with a description of the test environment. Section 5 presents our results that illustrate the impact of different types of noise on application performance. An analysis of how the performance results correlate with application communication and computation characteristics is provided in Section 6. Section 7 describes our work in relation to previous research, and we conclude in Section 8 with a discussion of the implications of our findings for the design of future system software for high-end systems, as well as a discussion of possible directions for future research.

2 Background

OS interference, also referred to as noise or jitter, is caused by asynchronous interruption of the application by the system software on the node. This interruption can occur for a variety of reasons, from the periodic timer “tick” commonly used by many commodity operating systems to keep track of time to the scheduling points used to replace the currently running process with another task or kernel daemon. In each of these cases, processor cycles are taken away for the duration of the noise event, which can typically vary from a few microseconds to a few milliseconds.

The detrimental side effects of OS interference on massively parallel processing systems have been known and studied, primarily qualitatively, for nearly two decades. For example, the origins of the Catamount lightweight kernel can be traced back to the difficulties inherent in trying to use a commodity server OS on a massively parallel machine—OSF1 [19] on a 3744-processor Intel Paragon circa 1993. More recently, Petrini et al. [13] raised the visibility of noise in a thorough study on the performance impact of noise on a large-scale cluster built from commodity hardware components running a commodity operating system. This paper reinforced the lessons that had been learned several years earlier on special-purpose parallel platforms—that operating system behavior can potentially have detrimental impacts on application performance at scale.

Previous investigations have suggested that the global performance cost of noise is due to the variance in the time it takes processes to participate in a collective operation such as `MPI_Allreduce`. Many par-

allel applications operate in a synchronous manner with distinct compute-then-communicate phases. For some codes, the communication phase can involve fine-grain global communication operations. For these applications, noise can delay one or more processes from reaching the global communication phase, forcing other processes to wait and *accumulating* the effects of noise on multiple nodes [11]. In addition, noise can similarly slow individual collective communication operations, particularly when a tree-based collective operation is implemented in software and the non-leaf nodes are delayed. In some cases, this situation has caused collective operations that usually take microseconds or milliseconds, to take seconds to complete [7, 13].

3 Approach

To evaluate the impact of noise on HPC applications, we built a kernel-level noise injection framework into the Catamount lightweight operating system that runs on the Red Storm machine at Sandia National Labs. Catamount is an ideal choice due to its extremely low native noise signature and demonstrated record of scalability. In addition, integration within Catamount allows for testing noise effects on a well-balanced machine (in terms of relative compute to communication performance) at scales of over ten thousand nodes.

Our noise injection framework provides the ability to specify a per-job noise pattern to be generated by the operating system during application execution. Parameters for noise generation pattern include: the frequency of the noise, duration of each individual noise event, the set of participating nodes, and a randomization method for noise patterns across nodes.

Noise is generated using a timer interrupt mechanism within Catamount. During job launch, the noise pattern parameters are distributed to each node. If the node should not randomize the interrupt frequency, the node sets the timer interrupt frequency equal to the specified noise frequency. When a timer interrupt is generated, Catamount interrupts the application and spins in a tight busy-loop for the duration requested. If a randomized frequency and/or duration is requested, the given parameter is interpreted to be the arithmetic mean of a Poisson-distributed random variable. The actual durations and inter-arrival times for noise generation are then calculated using this random variable.

4 Test Environment

In this section, we provide an overview of the hardware and software environment of our test system. This includes a description of our test platform, the three applications we ran on this platform, and performance analysis tools used to gather performance data about these applications.

4.1 Platform

The machine used for our experiments is the Red Storm system located at Sandia National Laboratories. Red Storm is a Cray XT3/4 series machine consisting of over 13,000 nodes, with each compute node containing a 2.4 GHz dual-core AMD Opteron processor and either 2 GB or 4 GB of main memory. Additionally, each node contains a Cray SeaStar [2] network interface and high-speed router. The SeaStar is connected to the Opteron via a HyperTransport link. The current generation SeaStar is capable of sustaining a peak unidirectional injection bandwidth of more than 2 GB/s and a peak unidirectional link bandwidth of more than 3 GB/s. For our tests, we changed the system to run the Catamount lightweight kernel containing our noise injection framework instead of the normal production version of Catamount.

4.2 Applications

We have collected results from three applications, CTH, SAGE, and POP, that represent important HPC modeling and simulation workloads. These applications represent a range of different computational techniques, all frequently run at very large scales (i.e. tens of thousands of nodes), and are key applications to both the United States Departments of Energy and Defense. We made three runs of each application under a variety of noise injection patterns for processor counts from two to over ten thousand nodes and observed the application slowdown in comparison to a run with no noise. We briefly describe these applications below.

CTH [5] is a multi-material, large deformation, strong shock wave, solid mechanics code developed by Sandia National Laboratories with models for multi-phase, elastic viscoplastic, porous, and explosive materials. CTH supports three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical, and spherical meshes, and uses second-order accurate numerical methods to reduce dispersion and dissipation and to produce accurate, efficient results. It is used for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues.

SAGE, SAIC's Adaptive Grid Eulerian hydrocode, is a multi-dimensional, multi-material, Eulerian hydrodynamics code with adaptive mesh refinement that uses second-order accurate numerical techniques [9]. It represents a large class of production applications at Los Alamos National Laboratory. It is a large-scale parallel code written in Fortran 90 and uses MPI for inter-processor communications. It routinely runs on thousands of processors for months at a time.

The Parallel Ocean Program (POP) [10] is an ocean circulation model developed at Los Alamos National Labs that is capable of ocean simulations as well as coupled atmosphere, ice, and land climate simula-

tions. Time integration is split into two parts: baroclinic and barotropic. In the baroclinic stage, the three-dimensional vertically-varying tendencies are integrated using a leapfrog scheme. The baroclinic stage consists of a preconditioned conjugate gradient solver which is used to solve for the two-dimensional surface pressure.

4.3 Performance Analysis Tools

In an effort to provide insight on how operating system interference affects the performance of these applications, we used two different tools for collecting application performance data. First, tracing data was collected for each application using the Cray Performance Analysis Tool (CrayPAT) to investigate global application performance. Second, we used the MPIP MPI profiling library to collect statistics on the individual MPI operations performed.

CrayPAT is a performance analysis tool provided with the Cray XT series of machines. It provides tools to instrument a program at key points in order to trace the application execution. In addition to tracing capabilities, CrayPAT also provides access to the hardware performance counters on a node. We used CrayPAT to track how the time in each MPI function varies with the scale of the application.

MPIP [17] is a lightweight, scalable MPI profiling library that collects statistical data on MPI function calls. We used this profiling library to gather statistics such as the frequency of MPI function calls and the sizes of messages used in various MPI functions.

5 Results

5.1 Overview

In this section, we present results showing the impact of different types of noise on the performance the three applications described in the previous section. First, we measured the impact of two representative commodity noise signatures; a high-frequency, low-duration similar to the timer interrupt profile measured on a moderately loaded commodity InfiniBand cluster, and a low-frequency, high-duration noise signature similar to that of periodic kernel daemon on a loaded commodity platform. Following this, we present results showing how application performance varies for a fixed total amount of noise with changing frequency/duration characteristics and for increasing noise with either frequency or duration fixed.

The majority of these signatures represent a 2.5% net processor noise signature. We focus on 2.5% profiles due to both specific measurements made on commodity systems and results of previous research

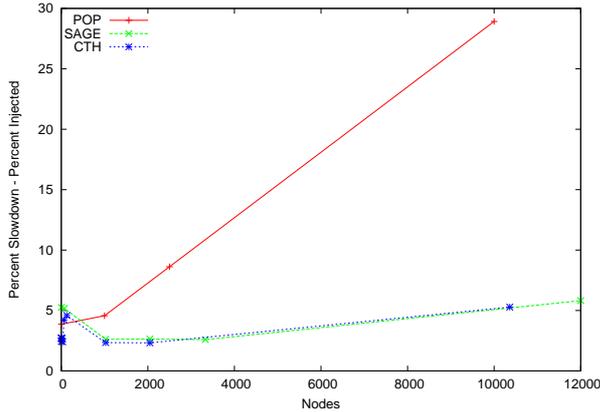
that demonstrate the importance of this noise level [13]. It is important to note that unloaded systems (e.g. those doing no communication, I/O, or memory management activities) can have lower noise signatures with corresponding lower overheads. However, we believe these unloaded noise patterns are not realistic for characterizing the behavior of real-world HPC applications, and Nataraj’s recent results showing significant OS overhead from scheduling and ACPI interrupts on loaded HPC Linux systems [11] support this view.

5.2 Noise Impact

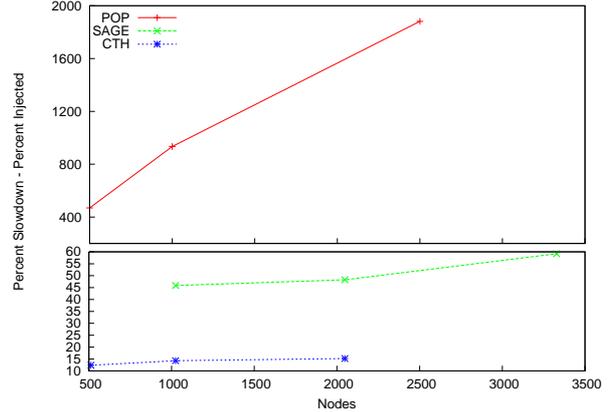
Figure 1 illustrates the performance impact of two noise signatures on POP, CTH, and SAGE. This slowdown represents the global accumulation of noise after the local net processor injected noise has been removed. For example, if we inject 2.5% net CPU noise and measure a 20% performance slowdown, the global accumulation of noise is only 17.5%. This percentage reflects the sensitivity of an application to a given noise signature. Note that this global accumulated noise percentage can be negative. This has been termed *absorption* [11] and can happen, for example, when a noise event occurs while the application is in `MPI_Wait`—the noise is absorbed during the normal application wait time.

Figure 1(a) illustrates the impact of a high-frequency, low-duration noise signature similar to that of the timer interrupt profile measured on a loaded a general-purpose commodity operating system like Linux. This 2.5% net CPU signature has an injection frequency of 1000 Hz and duration of 25 microseconds. These noise parameters correspond to a worst-case timer interrupt signature measured on a loaded commodity InfiniBand Linux cluster. For CTH and SAGE, the impact of the noise remains nearly constant and relatively minimal as scale increases—at these scales, CTH and SAGE are relatively insensitive to high-frequency, low-duration. POP, on the other hand, performs dramatically worse for this noise signature as the number of nodes increases—a nearly 30% slowdown at scale.

Figure 1(b) shows the impact of a low-frequency, high-duration noise signature; this 2.5% net CPU signature reflects the behavior of an application process competing with an intermittent kernel thread (e.g. a Linux bottom-half handler) or system daemon. This signature has a 10 Hz frequency and a 2500 microsecond duration. As in the previous results, this noise signature does not greatly affect the performance of CTH. On the other hand, SAGE incurs an accumulated slowdown of over 50% at this scale from a 2.5% net processor signature. Most impressive is the global impact noise has on POP. At these scales, the slowdown of POP due to the global effect of injected noise is nearly 2000%. This performance trend continued for POP out to ten thousand nodes, but due to limited dedicated system time, POP runs were stopped after reaching a factor of 20 slowdown.



(a) Loaded Timer Interrupt Signature



(b) Loaded Schedule Signature

Figure 1: Performance slowdown for two common commodity noise signatures. Each signature corresponds to 2.5% net processor noise. Loaded timer interrupt is a high-frequency, low duration signature while schedule is low-frequency, high-duration signature

5.3 Parameter Sensitivity

In the previous section, we showed the impact of common commodity noise signatures on application performance. For identical percentages of time spent in noise, we saw vastly different impact on application performance. In this section, we investigate the sensitivity of these applications to a wider range of frequency and duration noise signature parameters. We first look at sensitivity from the perspective of a fixed percentage of net noise, and then extend those results to increasing amounts of net noise on larger node counts.

5.3.1 Fixed Total Noise

Figure 2 shows the the performance impact of a constant 2.5% net processor noise signature with varied frequency and duration on POP, SAGE, and CTH for different node counts. Again, this slowdown represents the global accumulation of noise after the local net processor injected noise has been removed. The overwhelming trend from this data is the impact of lower-frequency, higher-duration noise signatures. Both SAGE and CTH in fact see basically no impact for frequencies greater than 25 Hz for these fixed percentage noise profiles. In addition, for both SAGE and CTH, node count has very little effect on the impact of injected noise.

POP’s performance, on the other hand, changes dramatically with changing noise distribution and scale. First, as we saw previously, the impact of noise on POP’s performance is much greater than for CTH and

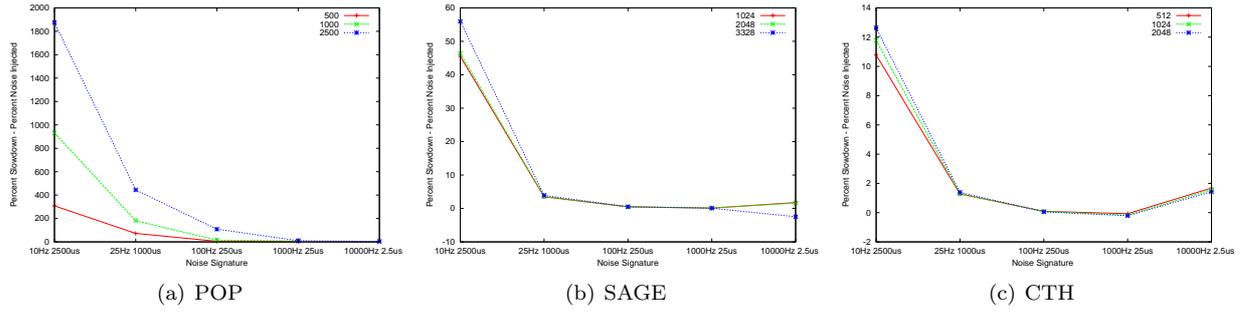


Figure 2: Sensitivity of POP, SAGE, and CTH to noise frequency and duration parameters for a fixed 2.5% net processor noise signature

SAGE, with POP having a nearly 1900% slowdown to a 2.5% CPU noise signature compared to nearly 60% for SAGE. In addition to the absolute differences in impact, there is a marked difference in the trends of these graphs—while CTH and SAGE’s performance does not change with scale, POP performance under low-frequency, high-duration noise degrades dramatically as scale increases. In addition, while CTH and SAGE are essentially sensitive to only the largest duration signatures tested, POP is sensitive, to varying degrees, to a much broader range of noise patterns.

5.3.2 Increasing Noise Signatures

We have established the sensitivity of our applications to higher-duration noise patterns for a fixed percentage of noise; we now investigate the impact of varying the percentage of injected noise by changing only noise frequency or duration. Figure 3 shows the global performance impact of noise for various net processor injected noise for durations from one to fifty microseconds. Figure 3(a) shows the results for POP. It is interesting to note POP’s sensitivity to duration. For each of the plotted durations, the difference between the first and last point is an increase by a factor of five in net injected noise. With this factor of five increase, POP performance degrades by at most 20-25%. This again shows POP’s sensitivity to the duration of a noise pattern and its relative insensitivity to the frequency of the noise pattern.

Figure 3(b) illustrates the duration sensitivity of CTH. In contrast to POP, CTH is relatively insensitive to the durations in this range. In fact, CTH is able to absorb a portion of the noise injected. Negative values on the Y-axis represent the amount of the injected noise that the application was able to absorb. Therefore, CTH can absorb around 20% of the injected noise in this range. We omitted SAGE results because in this duration range, SAGE showed very little global performance impact with varying small degrees of absorption and accumulation of the injected noise.

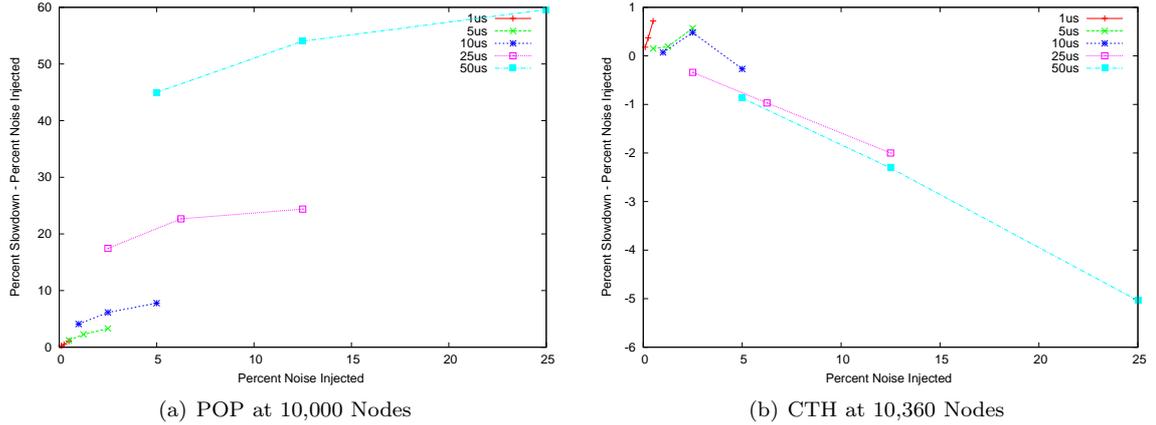


Figure 3: Noise duration sensitivity of POP and CTH for varying net processor injection percentages

6 Analysis

In this section, we investigate measured communication and computation characteristics of our tested applications in order to better understand the performance impact of OS noise. For each application, we present the amount of time spent in communication versus computation, and how communication time is spent in different MPI functions. We then combine this data with the performance results from the previous section to understand how each application’s communication demands effects its sensitivity to noise. Although we tested our noise framework on application scales of over 10,000 nodes, we could not run our applications under tracing for node counts greater than 512 due to limitations of CrayPAT. Given the scalability of these applications, the trends and conclusions we outline are appropriate for larger node counts.

6.1 POP

Recall from Section 5 that POP is sensitive to a variety of noise signatures, with a slowdown for a 2.5% net processor signature varying from 30% in the high-frequency, low-duration case to over 1900% in the low-frequency, high-duration case. Figure 4 illustrates the ratio of computation and communication time and provides a breakdown of communication time per MPI function for POP from 10 to 512 nodes. Figure 4(a) shows how the communication/computation ratio changes with node count. As node count increases, POP spends an increasing amount of time in communication. Figure 4(b) provides a breakdown of the increasing communication time by MPI function. As node count increases, an increasing amount of POP’s runtime is spent in MPI collective operations – in this case `MPI_Allreduce`, `MPI_Bcast` and `MPI_Barrier`. These collective operations are those operations that have been shown to amplify OS noise. In addition, POP

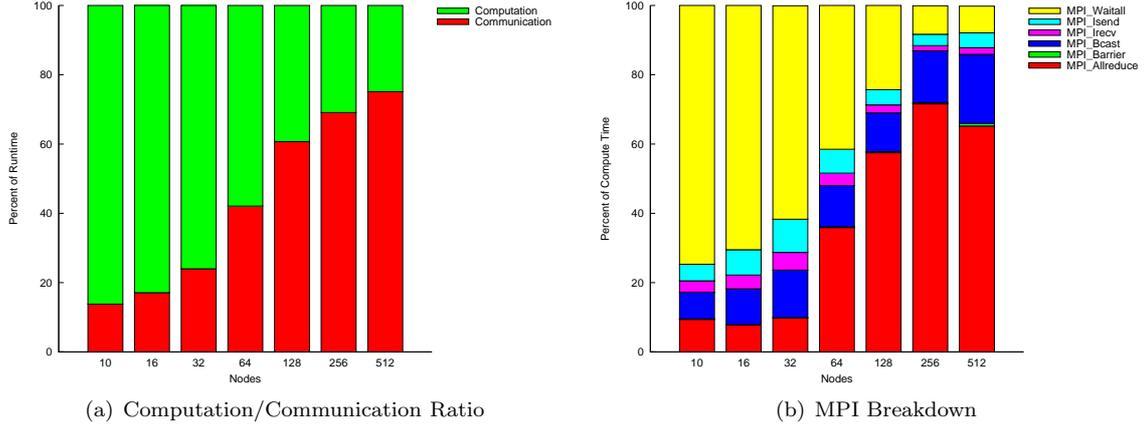


Figure 4: POP computation/communication ratio and percentage of communication time per MPI function

performs 20 times as many `MPI_Allreduce` operations per time interval as SAGE and CTH. All of these factors lead to POP’s sensitivity to operating system noise.

6.2 SAGE and CTH

We now turn our attention to the application characteristics of CTH and SAGE that lead to their varying degrees of noise sensitivity. Recall again from Section 5 that CTH is relatively insensitive to noise; in fact it absorbs a certain significant percentage of OS noise, while SAGE shows a sensitivity to low-frequency, high-frequency noise signatures.

6.2.1 Similarities between SAGE and CTH

Figure 5 illustrates the communication and computation ratios for CTH and SAGE for node counts from 2 to 512. For both of these applications, we see that as node count increases, the portion of communication time increases in a logarithmic fashion. This is in contrast to that of POP, shown in Figure 4(a), and we believe this explains the large difference in noise sensitivity between POP and the other two applications.

6.2.2 Communication Differences between SAGE and CTH

While SAGE and CTH are both less sensitive to noise than POP, they do show differences in how they react to low-frequency, high-duration noise. To further determine why SAGE is more sensitive to low-frequency, high-duration noise than CTH, we examine the characteristics of these collective operations in greater detail and the sensitivity to noise of these operations. Specifically, we examine how much time SAGE and CTH

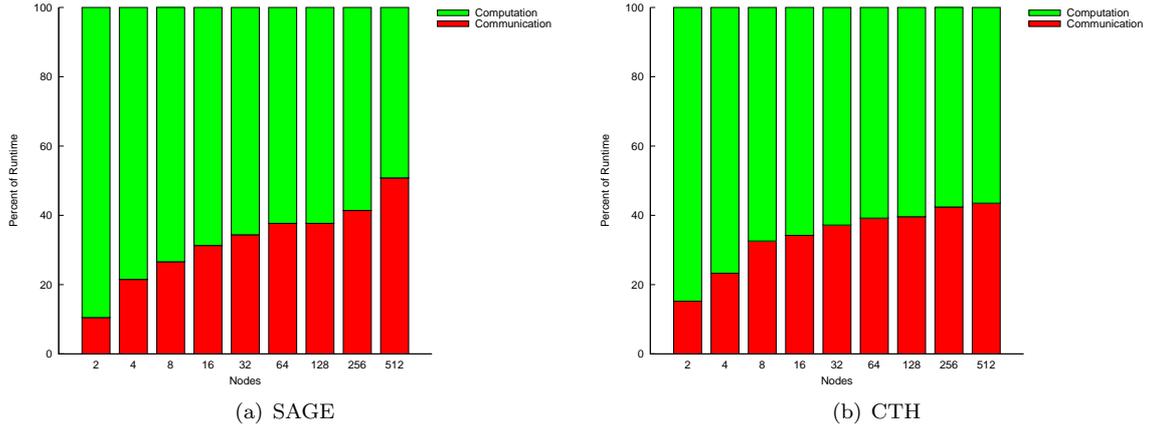


Figure 5: Communication/Computation ratio for SAGE and CTH

spend in different communication calls, the specific details of these calls, and the performance difference that noise makes based on these differences.

Differences in MPI Calls. Figure 6 shows the percentage of communication time spent per MPI function for both CTH and SAGE, and shows the primary differences in communication behavior between these two applications. Specifically, figure 6(a) shows that SAGE spends the majority of its communication time in `MPI_Allreduce` and `MPI_Wait`. CTH, on the other hand, spreads its time across `MPI_Allreduce`, `MPI_Bcast`, `MPI_Send`, `MPI_Wait`, and `MPI_Isend`, as shown in Figure 6(b). Note also that although SAGE spends more time in `MPI_Allreduce`, each of these two applications spend a similar amount of time in collective operations, with the majority of time in collective being in `MPI_Allreduce` for SAGE and `MPI_Allreduce` and `MPI_Bcast` for CTH.

Figure 7 shows the breakdown by size in bytes for `MPI_Allreduce` and `MPI_Bcast` for both CTH and SAGE on a 512-node application run. From this figure, we see that although both CTH and SAGE perform a similar number of `MPI_Allreduce` operations, the sizes of those operations vary in size between the two applications. Specifically, CTH has an average `MPI_Allreduce` size of approximately 32 bytes, while SAGE’s average is approximately 8 bytes.

Figure 7(b) shows the breakdown of broadcast sizes for CTH. Note we do not include SAGE in this figure as it does not perform a significant number of `MPI_Bcast` operations. From this figure we see the majority of the broadcast operations are smaller and the average is around 1024 bytes in length.

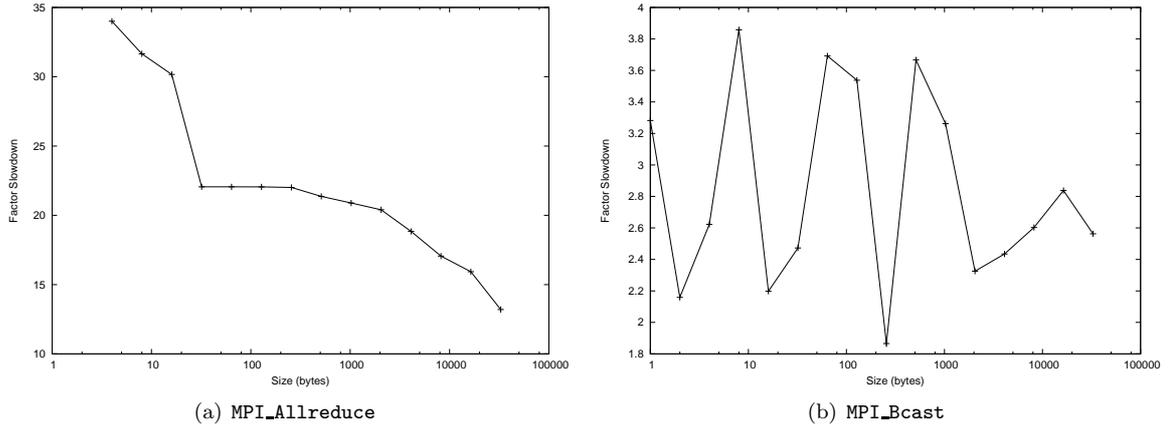


Figure 8: Performance impact of `MPI_Allreduce` and `MPI_Bcast` for a 2.5% net processor noise signature with a 10Hz frequency and 2500us duration

Impact of MPI Usage Differences on Noise Sensitivity. To understand how these different uses of MPI effect each application’s sensitivity to noise, we examined the performance of these specific operations under the 2.5% low-frequency noise signature to which SAGE demonstrated sensitivity in section 5. Figure 8 shows the performance impact of this signature on `MPI_Allreduce` and `MPI_Bcast` operations for 128 nodes. From this figure, we can see that `MPI_Bcast` is much less sensitive to this noise signature than `MPI_Allreduce`, with `MPI_Bcast` showing a factor of 2–4 slowdown and `MPI_Allreduce` showing a factor of 12–35 slowdown. In addition, small `MPI_Allreduce` calls appear to be much more sensitive to this noise signature than larger operations.

Based on this, we believe that SAGE’s sensitivity to OS noise comes from a combination of sources: SAGE spends more time in `MPI_Allreduce` than CTH, `MPI_Allreduce` operations are more sensitive to noise over `MPI_Bcast` operations, and SAGE’s smaller `MPI_Allreduce` operations are impacted to greater degree than CTH’s larger ones. In addition, CTH is likely able to absorb some of the injected noise due to the fact that it spends 60% of its time in operations that can potentially absorb noise (`MPI_Send`, `MPI_Wait`, and `MPI_Recv`).

7 Related Work

As mentioned previously, Petrini et al. [13] most recently raised the visibility of the impact of OS noise on application performance. Their thorough study investigated performance issues from OS noise on a large-scale cluster built from commodity hardware components, running a commodity operating system, and

running a cluster software environment designed for data center applications. While the findings of this paper from an OS perspective were largely well known, such as turning off unnecessary system daemons (like the line printer daemon), the paper brought to light several important new findings relevant to OS noise. First, the authors developed a micro-benchmark specifically for measuring OS noise on a parallel machine; such benchmarks were previously non-existent. Second, the paper demonstrates the inability of communication micro-benchmarks to accurately reflect and/or predict application performance. Specifically, the micro-benchmark performance of a global reduction communication operation was increased by a factor of seven. Yet, despite the application spending more than half of its time performing this operation, the application showed negligible performance improvement. Finally, the authors offered a conjecture that OS noise is most damaging when the application resonates with OS noise—high-frequency, fine-grained noise only affects fine-grained applications and low-frequency, coarse-grained noise only affects coarse-grained applications.

Similarly, Jones et al. [7] attacked the OS noise problem observed in global reduction communication operation performance on a large IBM system running AIX. Their approach was to coordinate “rogue” system activity by gang scheduling daemons across all of the nodes. This strategy led to a factor of 3 improvement in the time to perform a global reduction operation on several hundred nodes, and also allowed the use of all 16 processors on a node rather than dedicating one processor to only running system tasks. Due to the complexity of doing application-level analysis, their results were limited to global reduction micro-benchmark results.

Sottile and Minnich [15] described the limitations of the Fixed Work Quantum (FWQ) micro-benchmark and suggested that a Fixed Time Quantum (FTQ) micro-benchmark is a better approach for measuring OS interference. The FTQ benchmark eliminates the possibility of the influence of OS noise on the measurements, while the FWQ benchmark is itself susceptible to OS noise.

Tsafir et al. [16] attempted to quantify the effect of OS noise using a probabilistic approach. They also implemented a micro-benchmark similar to the FTQ benchmark and instrumented a Linux kernel to log OS interrupts. They stipulate that OS timer ticks are the main source of OS noise and that a tickless kernel is a possible solution to the noise problem. However, tickless kernels do not entirely address the issue. One important component that most massively parallel processing systems have that clusters do not is an infrastructure that monitors the health of the system using some type of heartbeat mechanism. These heartbeats are very infrequent – usually one per second – and small in duration, but their signature is very similar to OS ticks.

Beckman et al. [1] investigated the effect of user-level noise on an IBM BG/L system. This system runs a custom lightweight OS like Catamount that demonstrates very little noise. The authors showed that a properly configured Linux kernel can have a noise signature similar to that of a lightweight kernel. They then injected noise using a user-level communication library and measured the impact on communication micro-benchmarks. Their results showed that noise levels had to be very high in order to show any real impact. Given Petrini’s previous work showing no direct correlation between application performance and collective communication micro-benchmark performance [13], and the authors’ reliance solely on such synthetic micro-benchmarks on a single platform we question their broad conclusion that “running a general-purpose OS such as Linux on massively parallel machines should be viable”.

More recently, Nataraj et al. [11] used the KTAU toolkit to investigate the kernel activities of a general-purpose operating system. This toolkit instruments the Linux kernel to collect measurements from various kernel components including system calls, scheduling, interrupt handling, and network operations. The authors begin by showing the effectiveness of the KTAU toolkit for measuring the OS noise in Linux. In addition, they show how their toolkit can be used to track the accumulation and absorption of noise during the communication stages of an application. However, this work, unlike ours, only presented results from a 128-node development system that may or may not generalize to a massively parallel machines containing tens of thousands of nodes. More importantly, while their tool can be used to identify possible sources of noise, the authors did not relate the effects of noise to the performance of a large-scale application (e.g. the largest source of noise may not be the most harmful).

8 Conclusions

8.1 Summary of Results

In this paper, we present the first systematic study of the effect of OS noise on the performance of a range of applications on a large-scale system. Our results show that OS noise can have a dramatic impact on the performance of applications at scale, even at comparatively low noise levels. For example, we show that 1000 Hz $25\mu\text{s}$ noise interference—an amount of interference measured on a large-scale commodity Linux cluster—can cause a 30% slowdown in application performance on ten thousand nodes. Our results also show that, even for applications that are relatively insensitive to noise, this noise level can cause a 5% application slowdown above and beyond the slowdown resulting from the CPU time the OS directly takes from the application. We also show that infrequent noise, similar to that resulting from intermittently running kernel

service threads, can cause a 15% to 1900% slowdown in the performance of applications on only 2500 nodes.

More generally, our results show the importance of quantifying *how* noise is generated in high-performance systems, and that simply reducing the total noise in the system may not improve application performance. Specifically, our results show that applications are often able to absorb substantial amounts of high-frequency, low-duration noise but tend to amplify low-frequency, high-duration noise. For example, that an application notorious for amplifying noise (POP) amplifies high-duration low-frequency noise by 1900% at 2500 nodes, but does not amplify high-frequency low-duration noise. Similarly, a noise-tolerant, coarse-grained application such as CTH is better able to absorb high-frequency low-duration noise than it is able to absorb high-duration low-frequency noise.

In terms of applications behavior, our results confirm the widespread theory that the sensitivity of an application to OS noise is strongly related to the communication/computation ratio of a program, the amount of collective communications used by an application, and something previous researchers had not noted—the size of these collective communications. In particular, our results lead us to believe that applications that use extremely small collective communications (e.g. 8-byte `MPI_Allreduce`) appear much more susceptible to noise than ones that use even slightly larger versions (e.g. 32 bytes) of the same collectives.

8.2 Implications for System Software Design

Overall, our results reinforce the importance of being noise-conscious when designing system software for large-scale parallel systems. Operating system designers must carefully consider the performance vs. usability tradeoffs involved in every service, as some services can have dramatic performance impact at scale. While this is challenging, our results also provide guidance on *how* to implement services that system designers deem necessary—designers should focus on decomposing periodic services into small pieces that run more often to reduce the impact of unavoidable system noise on application performance.

Unfortunately, a number of system software designers appear to be taking the opposite approach and are aggregating periodic work into longer, less-frequent interruptions [18]. Our results suggest that while this may reduce the noise signature of these systems in noise measurement micro-benchmarks such as FTQ [15], it is also likely to degrade application performance. In general, many of our application results also reinforce the importance of using applications to evaluate system performance, and that relying on only synthetic micro-benchmarks is insufficient [12].

In addition, the susceptibility of small collectives to noise has substantial implications for system software designers. Because these operations seem to dramatically amplify even small amounts of noise, system soft-

ware that seeks to support applications requiring such communication must be particularly noise conscious. In particular, techniques such as lightweight kernels [14], non-blocking collective communication primitives, and/or hardware-based collective implementations that mitigate the performance impact of noise on applications appear to be critical for supporting important applications—especially considering the increasing node count that may be required to achieve multi-petascale or exascale levels of performance.

8.3 Future Work

There are several avenues of future work related to this study. First, we would like to analyze more applications in order to increase our understanding of application sensitivity to noise. While the set of applications covers a range of important problems and scalable computational techniques, additional application experimentation would further increase our understanding of the relationship between OS noise, communication primitive performance, and application performance. Obtaining access to large-scale applications, problem sets, appropriate application scientist expertise, and dedicated system time to run these applications has proven challenging, but we believe that this approach is key to understanding the overall impact of OS noise in large-scale parallel applications.

We are also interested in analyzing how basic OS services, for example memory management, can influence the generation and impact of noise. We are exploring modifications to our noise framework that allow Catamount’s memory management strategy to be more representative of a general-purpose OS like Linux. Specifically, we are implementing a non-contiguous memory page allocation scheme that better resembles the way Linux allocates and manages physical memory pages. We suspect that this capability will further degrade the performance of our emulated general-purpose noise signature to be more representative of what happens in a commodity OS environment.

Finally, we are also interested in exploring the influence of system balance on OS noise. Our conjecture is that systems that are less balanced in terms of network performance are more likely to be able to absorb noise than those that are more evenly balanced. An application that is network bound seems less likely to be impacted by processor cycles taken away by OS activities. We are exploring hardware-based mechanisms for degrading network injection bandwidth and network link bandwidth on the Cray XT to give it a balance ratio similar to a large commodity cluster.

References

- [1] 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 Conference on Cluster Computing*, September 2006.
- [2] R. Brightwell, T. Hudson, K. T. Pedretti, and K. D. Underwood. SeaStar Interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3), May/June 2006.
- [3] R. Brightwell, R. Riesen, K. Underwood, P. G. Bridges, A. B. Maccabe, and T. Hudson. A performance comparison of Linux and a lightweight kernel. In *Proceedings of the 2003 IEEE International Conference on Cluster Computing (Cluster 2003)*, December 2003.
- [4] W. J. Camp and J. L. Tomkins. Thor’s hammer: The first version of the Red Storm MPP architecture. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.
- [5] J. E.S. Hertel, R. Bell, M. Elrick, A. Farnsworth, G. Kerley, J. McGlaun, S. Petney, S. Silling, P. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings of the 19th International Symposium on Shock Waves, held at Marseille, France*, pages 377–382, July 1993.
- [6] J. Harrell. Cray, Inc., Director of Software Architecture. Personal Communication.
- [7] T. Jones, W. Tuel, L. Brenner, J. Fier, P. Caffrey, S. Dawson, R. Neely, R. Blackmore, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of SC’03*, 2003.
- [8] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*, May 2005.
- [9] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 37–48, Denver, CO, 2001. ACM Press.
- [10] D. J. Kerbyson and P. W. Jones. A performance model of the parallel ocean program. *Int. J. High Perform. Comput. Appl.*, 19(3):261–276, 2005.

- [11] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman. The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *Proceedings of SC'07*, 2007.
- [12] D. A. Patterson and J. Hennessey. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, Third edition, 2004.
- [13] F. Petrini, D. 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 SC'03*, 2003.
- [14] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira. Designing and implementing lightweight kernels for capability computing, 2008. Submitted for Publication.
- [15] M. J. Sottile and R. G. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *IEEE Conference on Cluster Computing*, September 2004.
- [16] D. Tsafir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *ACM International Conference on Supercomputing*, June 2005.
- [17] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2001.
- [18] D. Wallace. Compute Node Linux : Overview, progress to date, and roadmap. In *Proceedings of the 2007 Cray User Group Annual Technical Conference*, May 2007.
- [19] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the 1993 Winter USENIX Technical Conference*, pages 449–468, January 1993.