INVESTIGATING OPERATING SYSTEM NOISE IN EXTREME-SCALE HIGH-PERFORMANCE COMPUTING SYSTEMS USING SIMULATION

Christian Engelmann
Oak Ridge National Laboratory, Oak Ridge, TN, USA
email: engelmannc@ornl.gov

ABSTRACT
Hardware/software co-design for future-generation high-performance computing (HPC) systems aims at closing the gap between the peak capabilities of the hardware and the performance realized by applications (application-architecture performance gap). Performance profiling of architectures and applications is a crucial part of this iterative process. The work in this paper focuses on operating system (OS) noise as an additional factor to be considered for co-design. It represents the first step in including OS noise in HPC hardware/software co-design by adding a noise injection feature to an existing simulation-based co-design toolkit. It reuses an existing abstraction for OS noise with frequency (periodic recurrence) and period (duration of each occurrence) to enhance the processor model of the Extreme-scale Simulator (xSim) with synchronized and random OS noise simulation. The results demonstrate this capability by evaluating the impact of OS noise on MPI_Bcast() and MPI_Reduce() in a simulated future-generation HPC system with 2,097,152 compute nodes.

KEY WORDS
high-performance computing, performance evaluation, operating system noise, parallel discrete event simulation

1 Introduction

The top tier of the current-generation high-performance computing (HPC) systems is able to perform 10+ PFlop/s (1 PFlop/s = 10^{15} floating-point operations per second) using the LINPACK benchmark [12]. For example, the IBM BlueGene/Q Sequoia supercomputer at Lawrence Livermore National Laboratory has 1.57 million processor cores with a theoretical peak performance of 20.1 PFlop/s and a LINPACK performance of 16.3 PFlop/s. The system's computational efficiency for LINPACK is 81 %. The gap between the peak capabilities of the hardware and the performance realized by HPC applications, or in this case LINPACK, is referred to as the application-architecture performance gap, which is system and application dependent. Since the performance properties of different HPC system architectures can vary and the performance needs of different HPC applications are often not the same, HPC hardware/software co-design for future-generation HPC systems aims at closing this application-architecture performance gap by designing systems to meet application needs and applications to exploit system properties.

Performance profiling of HPC system architectures and of HPC applications is an essential part of HPC hardware/software co-design. While prototypes of future HPC architectures can be built for performance profiling at small scale, a large-scale deployment of a particular architecture is typically only available once the HPC system has been delivered to the customer due to the fact that such a system may easily cost $100M. System architects and application developers involved in HPC hardware/software co-design resort to simulation approaches to investigate the impact of different architectural parameters on application performance. Simulating an HPC system that is 10-to-100 times faster than today's HPC systems requires a significant amount of resources. Parallel discrete event simulation (PDES) solutions are often employed with cycle-accurate simulation capabilities at small scale and less accuracy at large scale (see Section 2.1).

The road to exascale supercomputing (1,000 PFlop/s) faces a number of challenges related to power consumption, performance, resilience, productivity, programmability, data movement, and data management [10]. To provide such level of computational performance, future HPC system architectures will need to have a correspondingly increased level of parallelism, i.e., number of employed processor cores, due to the end of frequency scaling. To achieve better energy efficiency, the number of cores will increase even further by using more, but less computationally powerful and significantly less energy demanding, cores. By 2020, an exascale system may have up to 1,000,000 compute nodes with 1,000-10,000 cores per node. At billion-way parallelism, even the tiniest performance problem may cause a severe reduction in computational efficiency due to synchronizing communication. HPC hardware/software co-design using simulation tries to identify and resolve such performance issues that are observed at extreme-scale, focusing on different architectural parameters, such as a variety of processor core, memory and network interconnect capabilities.

For example, a future HPC system may have a Dragonfly (torus + butterfly) or a Tofu (torus + hypercube) network, a certain mix of light- and heavyweight cores or a homogeneous set of light- or heavyweight cores, a packet or circuit-switched mesh on-chip network, and 3D stacked memory with different amounts of core-private and shared memory. As the hardware vendors march forward in developing these technologies, HPC hardware/software co-

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
Timer update	1 μs	Process scheduler runs
Page fault	10 μs	Modifying a variable after fork()
Pre-emption	10 ms	Another process runs

Table 1. Overview of typical detours [2]

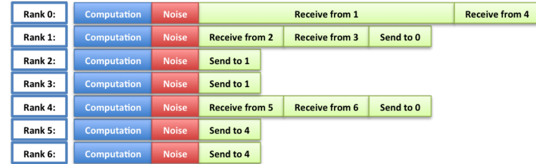
design efforts at HPC centers provide the guidance in making architecture decisions to enable the delivery of efficient HPC systems and design HPC applications to exploit the offered hardware features [1].

The research presented in this paper focuses on the problem of operating system (OS) noise as an additional factor to be considered for simulation-based HPC hardware/software co-design. As an OS primarily provides resource management to enable an application's execution, it inevitably interferes with its execution resulting in a slight performance degradation. Table 1 provides an overview of the impact of typical execution detours that are commonly referred to as OS noise [2]. On a single processor core, the OS noise ratio is defined by the amount of extra time spent on such detours. However, on multiple processor cores that execute a communicating parallel application, OS noise may be amplified or absorbed depending on the application's communication patters.

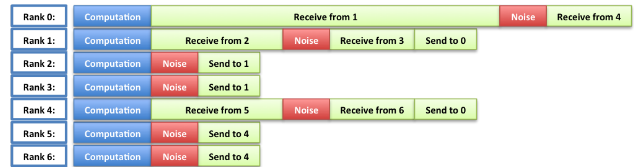
Figure 1 shows a basic example of an MPI application executing a simple tree-based collective, such as an MPI_Reduce(), with 7 MPI ranks, in a (a) noise-free environment, (b) a system with synchronized noise, and (c) a system with random noise. The colored bars show the different execution stages of each MPI rank with a horizontal time axis from left to right. The noise periods may come, for example, from a timer interrupt. The execution time with synchronized noise is simply extended by the noise on all MPI ranks simultaneously. In this case, the noise may occur during any pure computation phase. While the OS noise ratio in the entire system with synchronized noise is equivalent to that of a single MPI rank, in this example, in the system with random noise, it is amplified by the waiting for communication. Similarly, OS noise can also be absorbed during waiting for communication (not shown in this example). This basic example provides the motivation for investigating OS noise in extreme-scale systems as noise amplification can become a serious issue with a large number of communicating MPI ranks, especially with globally synchronizing collective communication.

OS noise has been investigated in the past as an isolated performance issue in large-scale systems (see Section 2.2). However, in the context of HPC system design, OS noise is only one out of many design factors. While OS noise has been acknowledged as a performance issue, its importance and interplay with other system design factors are unknown. For example, should a system have a globally synchronized hardware clock across all nodes to synchronize OS noise caused by timer interrupts? From past research, it is clear that this mitigation would certainly

(a) Noise free



(b) Synchronized noise



(c) Random noise

Figure 1. Example: Impact of OS noise on MPI_Reduce()

lower overall OS noise interference. However, does the application performance gain justify the involved deployment cost? Are there other system design factors that are more important or that can even mitigate OS noise in a different way, e.g., through side effects?

This paper represents the first step in considering OS noise in HPC hardware/software co-design by adding an OS noise injection feature to an existing simulation-based HPC co-design toolkit. Given the accuracy trade-off that is required to enable extreme-scale simulation, the work in this paper focuses on an appropriate abstraction of OS noise, which is injected in to a simulated HPC system to permit performance evaluation of applications on different future-generation system architectures.

2 Related Work

The following discusses related research and development in HPC hardware/software co-design using simulation and in investigating HPC operating system noise.

2.1 Hardware/Software Co-Design

The BigSim [18] project was initiated in 2001 to study programming issues in large-scale HPC systems. The *BigSim Emulator* is for application testing and debugging at scale and build atop Charm++ and Adaptive MPI [8]. It supports up to 100,000 simulated MPI processes distributed over 2,000 native processors. However, it does not offer time-accurate simulation. The *BigSim Simulator* is for identification of performance bottlenecks and uses a trace-driven PDES that models architectural parameters of a HPC system. For time-accurate simulation, it supports a variable-resolution processor model and a detailed network model. While it uses a PDES to maintain accuracy, it does not support running native applications.

The recently developed Structural Simulation Toolkit (SST) [17] offers cycle-accurate simulation of novel compute-node architectures, including processor, memory, and network. It is a modular PDES framework atop MPI that scales to a few hundred nodes with different levels of accuracy, utilizing a variety of external modeling and simulation tools. Its value is in the ability to investigate the performance of future architectures and to generate models for larger-scale simulations. SST does support running native applications. SST/macro is a complementary simulation toolkit that processes output from the MPI tracing library DUMPI for performance evaluation similar to the BigSim Simulator approach. Similar to the BigSim Emulator/Simulator, SST and SST/macro enable the synergy between small-scale cycle-accurate and large-scale communication-accurate simulations. While SST is mature, it is quite complex to use. SST/macro is currently under development and not generally available.

The Extreme-scale Simulator (xSim) [3, 4, 7] is a recently developed performance investigation toolkit that permits running HPC applications in a controlled environment with millions of concurrent execution threads. It allows observing application performance in a simulated extreme-scale system for hardware/software co-design. Using a lightweight PDES, xSim executes an application on a much smaller HPC system in an oversubscribed fashion with a virtual wall clock time, such that performance data can be extracted based on a processor and a network model with the appropriate simulation scalability/accuracy trade-off. xSim is designed like a traditional performance tool, as an interposition library that sits between the MPI application and the MPI layer, using the MPI performance tool interface. It is able to run up to 134,217,728 (2^{27}) communicating MPI tasks, each with its own process context, using just a 960-core Linux-based cluster.

Other trace-driven PDES solutions for investigating application performance exist. DIMEMAS [6] processes traces from MPIDTrace and generates trace files that are suitable for the two performance tools, PARAVER [16] and Vampir [9]. Other performance modeling and simulation tools for HPC co-design include (a) communication-level tracing tools, (b) architecture-level (processor and memory) emulators/simulators, and chip-, node-, and system-level power consumption and heat dissipation models.

2.2 Operating System Noise

In 2005 [2], the OS noise was measured on a cluster running Linux, a Cray XT3 system running the lightweight Catamount OS, and an IBM Blue Gene/L system running a specialized lightweight OS (CNK). The measured noise ratio was 0.12 % for Linux with noise periods of 8.5-109.7 μs (6.2 μs average), 0.002 % for Catamount with noise periods of 1.2-9.5 μs (2.1 μs average), and 0.000029 % for CNK with noise periods of 1.8 μs. This work further investigated the impact of OS noise on collective MPI operations. Noise with frequencies of 10 Hz, 100 Hz, and 1 kHz and with periods of 16 μs, 50 μs, and 200 μs was injected into MPI_Barrier(), MPI_Allreduce(), and MPI_Alltoall() on a 16,384-node Blue Gene/L system. The results show an impact on the execution time of an MPI_Barrier() of up to 26 % for synchronized noise and 26,800 % for random noise, demonstrating the amplifying effect of OS noise. The discussion in this work also hints at the dependence of the noise impact on the implementation of the collective MPI operation, i.e., at the noise vulnerability of a specific collective MPI algorithm.

A 2008 study [14], analyzed the impact of OS noise on application performance. The approach focused on OS-level instrumentation, in addition to application instrumentation, using the KTAU and TAU performance tools to investigate the effects of noise amplification and absorption in parallel applications. The experiments with an injected noise frequency of 1 kHz and different noise periods of 10, 50, and 100 μs using a synthetic bulk-synchronous benchmark on up to 2048 processors show that the KTAU/TAU combination can differentiate between the noise already in the system and the injected noise. Using LU and CG from the NAS parallel benchmark suite [15] on up to 32 processors, different sources of noise are identified, such as *timer interrupt* and *schedule* events, that impact the benchmarks' runtime. Using Sweep3D on up to 32 dual-core processors, the contribution of these different sources of noise to the overall noise and to the accumulated (amplified) noise is investigated. Overall, this study demonstrates the complexity of accurate OS noise impact analysis.

Research in 2011 [5], investigated the impact of architectural parameters, such as network bandwidth, on OS noise absorption. The study focused on the Parallel Ocean Program (POP), SAIC's Adaptive Grid Eulerian (SAGE) hydrocode, and CTH, a shock physics code, using the Cray Red Storm XT3/4 system at Sandia National Laboratory with up to 3360 dual-core nodes and the lightweight, noiseless Catamount OS. The results show that an injected OS noise of 10 Hz to 10 kHz with a fixed noise ratio of 2.5 % can be absorbed better with a lower network interconnect bandwidth. The relative slowdown introduced by the OS noise at 50 % and 25 % bandwidth was only roughly half for POP, about a third for SAGE, and approximately a sixth for CTH. This work further identified differences between the benchmarks when only a certain number of nodes are noisy and when these noisy nodes are randomly organized or grouped sequentially in neighboring MPI ranks. It also hinted at the capability of better noise absorption with non-blocking MPI collectives.

In 2011 [13], a more quantitative analysis of noise in the Linux OS using OS-level instrumentation focused on recording the frequency and duration of individual noise sources, e.g., page fault, different interrupt types, and process preemption, and visualizing the noise trace data using Paraver [16]. The experiments targeted the Sequoia benchmarks (AMG, IRS, LAMMPS, SPHOT, UMT) [11] to break down the individual sources of OS noise for each benchmark and to identify the noise frequency and period for each source. The results demonstrate the existence

Noise Frequency	Noise Period	Noise Ratio
10 Hz	100 μs	0.1 %
100 Hz	10 μs	0.1 %
1 kHz	1 μs	0.1 %
10 Hz	500 μs	0.5 %
100 Hz	50 μs	0.5 %
1 kHz	5 μs	0.5 %
10 Hz	1 ms	1.0 %
100 Hz	100 μs	1.0 %
1 kHz	10 μs	1.0 %

Table 2. Noise ratios derived from noise frequency/period

of application-specific OS noise footprints caused by page faults and interrupt-driven I/O activity.

There are a few other OS noise studies focusing on collective MPI communication and MPI applications in large-scale HPC systems with noisy OSs or with injected OS noise. Their results are similar to the discussed work.

3 Design and Implementation

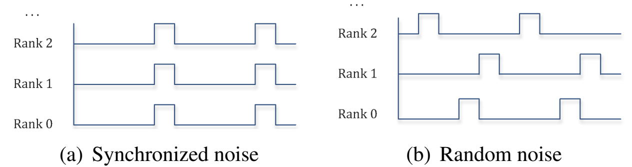
The purpose of this work is to enable the investigation of the performance impact of OS noise on applications in conjunction with other HPC system design parameters in future-generation extreme-scale systems with potentially up to 1,000,000,000 cores. This will allow for an optimal design of the hardware/software environment based on the target system's and applications' noise properties.

As discussed, the various sources of OS noise are either application dependent, such as page faults, or OS dependent, such as timer interrupts. A highly detailed simulation of individual OS noise sources is impractical at extreme scale as the processor and network models of the simulation toolkit would need to identify individual execution detours caused by the application and inject individual execution detours caused by the OS, all at extreme scale parallelism. As such a detailed simulation is simply not scalable due to the complexity involved for maintaining detailed models at extreme scale, a simpler abstraction of OS noise is targeted instead.

The Extreme-scale Simulator (xSim) [3, 4, 7] (see Section 2.1) is the only HPC co-design toolkit that exists to date that is appropriate for this purpose as it permits the investigation of HPC system and application performance properties at extreme scale by trading off accuracy for scalability. The presented work focuses on enhancing xSim with an OS noise injection feature.

Past work already used the abstraction of OS noise frequency (periodic recurrence) and period (duration of each occurrence) for OS noise injection experiments. This work reuses this abstraction by enhancing xSim's processor model with an OS noise injection feature that is able to simulate OS noise with a configurable noise frequency, noise period, and noise ratio (see Table 2). xSim's processor model has been modified to allow for the configuration of an OS noise frequency and period, and to include the injected OS noise in the accounting of execution time.

The current processor model in xSim operates as follows. For every simulated MPI process, xSim's PDES



(a) Synchronized noise (b) Random noise

Figure 2. Generated noise injected into the simulation

maintains a virtual time that is equivalent to the actual execution time of the simulated MPI process, scaled to the performance of a simulated processor as defined by the processor model. Similar to advanced operating system support for process-specific execution clocks, every time slice a simulated MPI process executes on the native system is measured, adjusted by the processor model, and recorded. The processor model effectively adjusts the difference between the execution on the native processor and on the simulated processor using a basic performance model. While this is a rather simple model, it is scalable and extendable, such as by including processor performance counters.

To allow for OS noise investigation, the basic scaling processor model was modified and extended with an accounting of injected OS noise as executed time. OS noise is injected, i.e., simulated, by adding waste time to the execution time of simulated MPI processes using a noise frequency/period generator for each simulated MPI process. For example, with a native-to-simulated processor scaling of 100 %, i.e., the native processor has the same performance as the simulated one, and a noise frequency/period with a ratio of 1 %, a native execution slice of a single simulated MPI process of 100 ms will be recorded as 101 ms (with an extra waste time of 1 ms) if this execution slice experienced a simulated OS noise event.

For synchronized OS noise (see Figure 2(a)), events are injected based on the configured frequency/period with all simulated processes simultaneously starting their frequency at simulated time 0 (process start). In other words, each simulated MPI process has the same frequency, period, and start time configured for its OS noise generator, thus all MPI processes have the same OS noise generator.

For randomized noise (see Figure 2(b)), each simulated MPI process starts the frequency at a different, randomly chosen, time within the first occurrence time window (1/noise frequency). In other words, each simulated MPI process has the same frequency and period configured for its OS noise generator, but the start time is different. This generates an OS noise pattern that is different between simulated MPI processes, but is still regularly recurring for each simulated process. This mechanism was favored over a probability-based distribution of OS noise events for each simulated process due to its lower simulation complexity. It was also chosen to allow for deterministic and repeatable experiments as the randomly chosen start point uses a deterministic random number generator seed.

To enable the simulation of OS noise absorption during times of simulated MPI process inactivity, e.g., no OS noise is experienced when waiting for a message or when

sleeping on a timer, the accounting of OS noise was excluded for such simulated process inactivity.

Certain messaging-related activity, such as sending or receiving a message, does involve the processor, such as to copy data between the network device and the user-provided MPI buffers, and may experience OS noise. Networking-related times of simulated process activity and inactivity are managed by xSim's network model, using xSim's processor model to account for such activity that involve the processor. This way, no OS noise is experienced by a simulated MPI process while simply waiting for an MPI message, but is experienced for sending or receiving one. This distinction is essential to account for OS noise during MPI collectives.

4 Results

To demonstrate the capabilities of the newly added OS noise injection feature, several series of experiments were executed focusing on the performance of the MPI_Bcast() and MPI_Reduce() collective communication operations in a simulated noise-free environment and in simulated synchronized and randomly noisy HPC systems with different parameters for injected OS noise frequency and period.

The experiments were performed on a 960-core Linux-based cluster with 40 nodes, two 1.7 GHz AMD Opteron processors/node, 12 cores/processor, 64 GB RAM/node, and a bonded dual non-blocking 1 Gbps Ethernet network. The system is running Ubuntu Linux 10.04.3 LTS and Open MPI 1.5.5. Despite its relatively small size, in comparison to the targeted extreme-scale systems, this system has been particularly useful for experimenting with xSim due to its large amount of total RAM (2.5 TB).

The first series of experiments investigates the performance of MPI_Bcast() and MPI_Reduce() on a future-generation HPC system to obtain baseline numbers. The simulated system has 2,097,152 (2^{21}) nodes, organized in a 128×128×128 3-D torus with 1 μs link latency and 32 GB/s link bandwidth. The eager threshold is set to 256 kB. An MPI+X, e.g., MPI+OpenMP, programming model is assumed, i.e., one simulated MPI process is placed on one node and the number of cores/node is not considered. For simplicity, the processor model is set to 100 % scaling. The number of nodes and the network architecture/performance are based on rough estimates for an exascale system. MPI_Bcast() and MPI_Reduce() are executed using a benchmark with a binomial-tree implementation of the MPI collectives. MPI_Reduce() performs a sum.

Figure 3 shows the performance of MPI_Bcast() on the simulated future-generation HPC system without any noise and with MPI message payloads of 1 B, 1 kB, 1 MB, and 1 GB (2^0-2^{30} bytes). The scaling effect of the binomial-tree implementation is clearly visible, as well as, the impact of the increasing payload. As MPI_Bcast() and MPI_Reduce() have the same payloads and binomial trees, only slight differences (not shown) were observed due to the fan-out communication pattern of MPI_Bcast() and the fan-in communication pattern of MPI_Reduce(), as well as, the additional sum operation on the data.

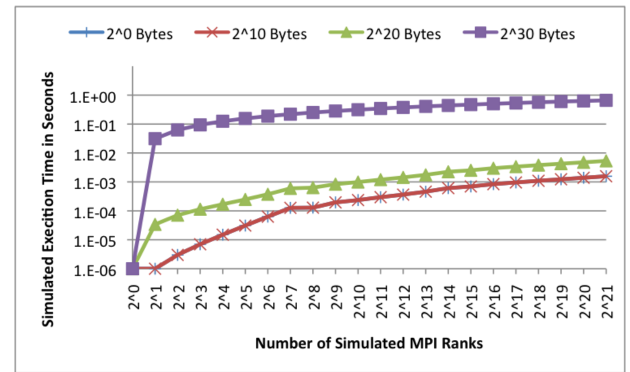
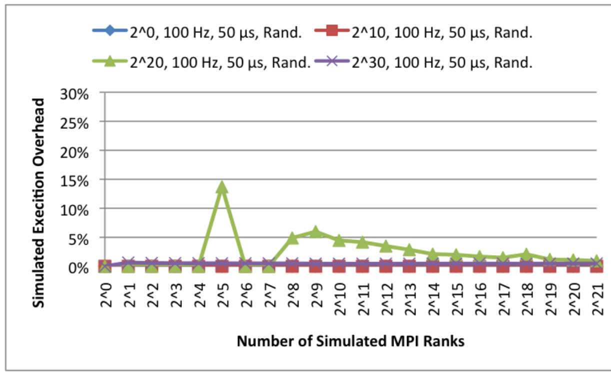


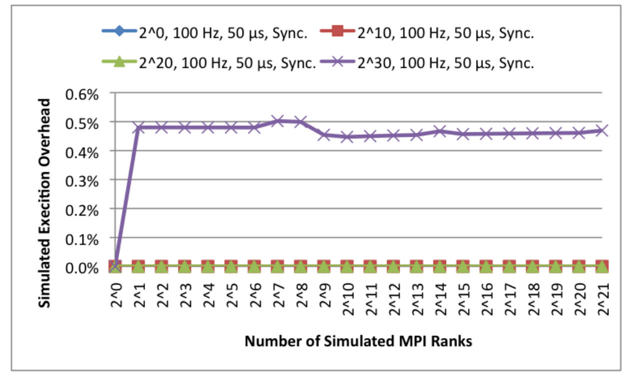
Figure 3. Noise-free MPI_Bcast()

In a second series, synchronized and random OS noise is introduced with a noise frequency of 100 Hz and a noise period of 50 μs (a 0.5 % noise ratio). Figure 4 illustrates the performance of MPI_Bcast() and MPI_Reduce() on the simulated HPC system with this noise and with MPI message payloads of 2^0-2^{30} bytes. Figures 4(a), for MPI_Bcast(), and 4(c), for MPI_Reduce(), show a significant performance impact of up to 14 % and 28 %, respectively, for random noise with an MPI message payload of 1 MB (2^{20} bytes). This seems to be a noise amplification sweet spot for this particular simulated HPC system as this significant performance reduction is not present for the other payloads and/or at all scales. Figures 4(b), for MPI_Bcast(), and 4(d), for MPI_Reduce(), demonstrate zero impact for synchronized noise with MPI message payloads of $<$1 GB (2^{30} bytes), and an overhead roughly equivalent to the average noise ratio at 2^{30} bytes. This series of experiments shows that the developed simulation capability can demonstrate the noise amplification effect. It also shows that this effect is only present under certain conditions, such as with a specific scale and payload, resulting in sweet spots for noise amplification.

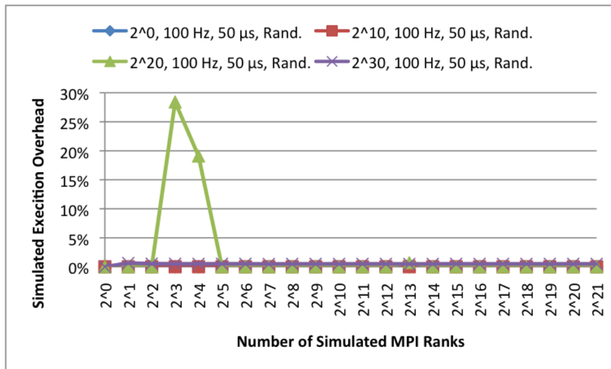
The third set of experiments investigates the discovered noise amplification sweet spot further. At 2^{20} bytes MPI message payload, random OS noise is introduced with a frequency of 100 Hz and different periods of 10 μs, 50 μs, and 100 μs, and the frequency is changed between 10 Hz, 100 Hz, and 1k Hz while the noise period is adjusted to maintain a 0.5 % noise ratio. Figure 5 shows the performance of MPI_Bcast() and MPI_Reduce() on the simulated HPC system. Figures 5(a), for MPI_Bcast(), and 5(c), for MPI_Reduce(), reveal that the OS noise amplification directly depends on the noise period. The longer the period, the more execution overhead is observed. The overhead for MPI_Bcast() and MPI_Reduce() can be as high as 35 % and 58 %, respectively. Figures 5(b), for MPI_Bcast(), and 5(d), for MPI_Reduce(), demonstrate that the OS noise amplification is fixed to the OS noise frequency of 100 Hz. This series of experiments demonstrates the investigative capabilities offered by the developed OS noise injection feature. It also provides the insight that the occurrence of noise amplification is correlated to its frequency and its impact is



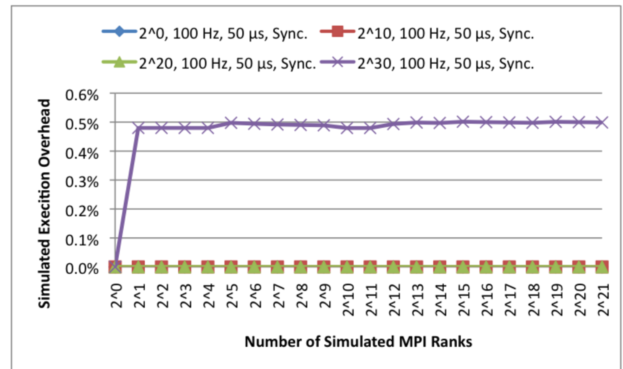
(a) MPI_Bcast() with random noise



(b) MPI_Bcast() with synchronized noise

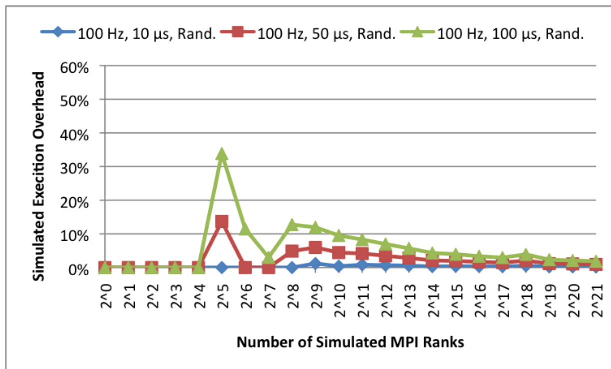


(c) MPI_Reduce() with random noise

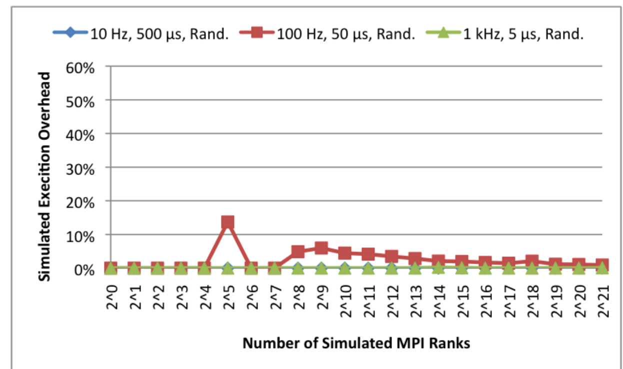


(d) MPI_Reduce() with synchronized noise

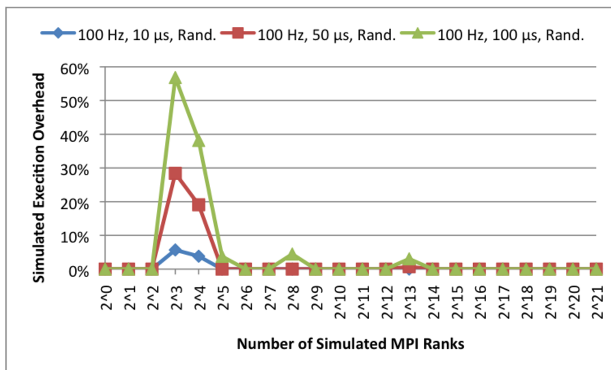
Figure 4. Noisy MPI_Bcast() and MPI_Reduce() with 2^0-2^{30} bytes payload



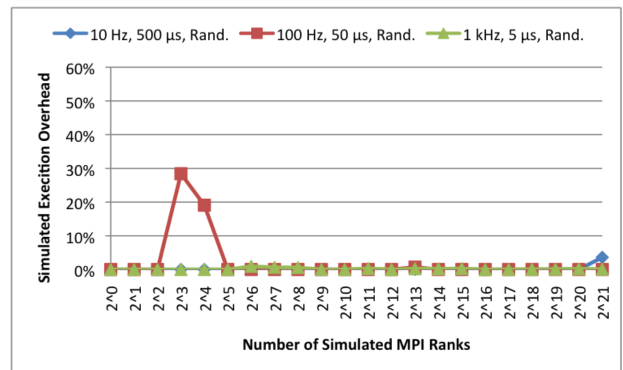
(a) MPI_Bcast() with different period (changing noise ratio)



(b) MPI_Bcast() with different frequency/period (fixed noise ratio)

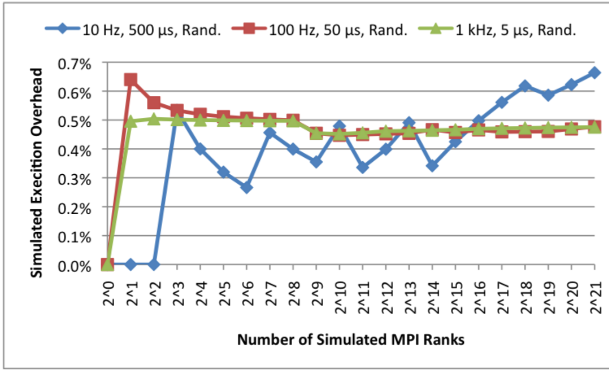


(c) MPI_Reduce() with different period (changing noise ratio)

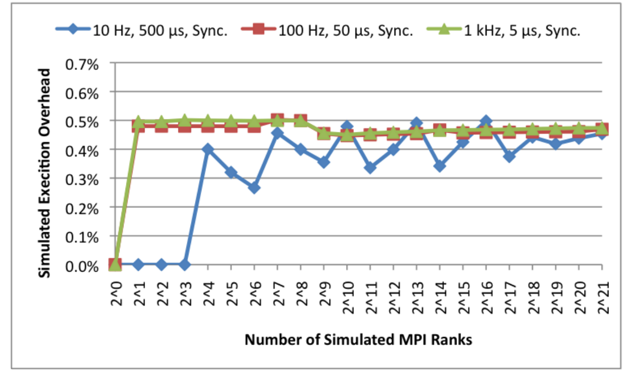


(d) MPI_Reduce() with different frequency/period (fixed noise ratio)

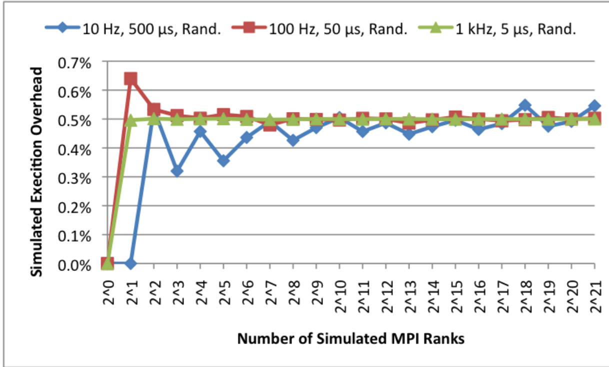
Figure 5. MPI_Bcast() and MPI_Reduce() with 2^{20} bytes payload and different random noise



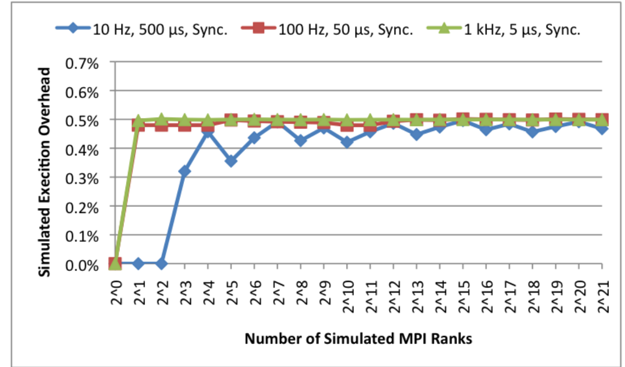
(a) MPI_Bcast() with random noise



(b) MPI_Bcast() with synchronized noise



(c) MPI_Reduce() with random noise



(d) MPI_Reduce() with synchronized noise

Figure 6. MPI_Bcast() and MPI_Reduce() with 2^{30} bytes payload and different noise frequency/period (fixed noise ratio)

defined by its period.

In the fourth series of experiments, the introduced overhead at 1 GB (2^{30} bytes) MPI message payload is evaluated with a synchronized and a random noise frequency of 10 Hz, 100 Hz, and 1k Hz and an adjusted noise period to maintain a 0.5 % noise ratio. Figure 6 shows the performance on the simulated HPC system. While there are differences between synchronized and a random noise, such as the slight noise amplification with random noise for the MPI_Bcast() in Figure 6(a) and for the MPI_Reduce() in Figure 6(c), all results in Figure 6 demonstrate the effect of OS noise absorption, i.e., an execution overhead of less than the 0.5 % noise ratio, at lower scales. This series of experiments shows that the developed simulation capability can further demonstrate the noise absorption effect. Specifically, it shows that absorption occurs at small scale and at various sweet spots. The reduction in overhead depends on noise frequency and period, as well as, on payload.

5 Conclusions

In summary, this paper represents the first step in including OS noise in HPC hardware/software co-design by adding an OS noise injection feature to the xSim HPC co-design toolkit. Using the common abstraction of OS noise with frequency and period, xSim's processor model was enhanced with synchronized and random OS noise simulation. The experiments investigating the impact of OS noise on MPI_Bcast() and MPI_Reduce() in a simulated HPC system with 2,097,152 compute nodes demonstrated this new capability, including showing OS noise amplification, absorption, and sweet spots. The existing knowledge confirmed with the performed experiments include the insight that the occurrence of noise amplification and absorption is correlated to noise frequency and period. The new knowledge obtained from the experiments is the existence of sweet spots for both. It is further noted that an OS noise overhead explosion at extreme scale, as suggested by earlier experiments, could not be confirmed.

The value of the developed solution lies in its capability to investigate specific HPC architectures using parametrized studies. In the context of co-design, many architectural parameters are bound by physical limitations of the hardware, cost limitations of the procurer, and performance characteristics of the targeted applications. Based on this parameter envelope, the design of an HPC system can be further refined using the developed capability. For example, a known system design can be investigated for its OS noise amplification and absorption sweet spots to tune the OS and runtime system by changing the OS kernel service (tick) interval frequency and period, or to decide if a separate "noise core" is needed to offload and consolidate noisy activities to one core per compute node. Further decisions based on OS noise simulation studies may consider selecting an OS kernel or distribution based on the system's

noise sensitivity. This may include using a lightweight OS kernel on compute cores and a heavyweight OS kernel on service cores (for I/O) within the same compute node.

Future work targets different aspects in extending the developed feature: 1) different OS noise frequencies/periods may be injected at the same time to more closely simulate real system behavior, 2) different OS noise patterns may be injected for each MPI process to simulate noise cores and different OS kernels within a compute node when considering an MPI-everywhere programming paradigm, 3) a processor model with performance counter support may be used to enable the simulation of application-dependent noise, such as page faults, and 4) the impact of OS noise on real applications may be investigated using proxy/mini applications.

6 Acknowledgements

This work was sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy (DOE) and was performed at Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC. This manuscript has been authored under Contract No. DE-AC05-00OR22725 with the DOE. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

References

[1] Advanced Scientific Computing Research, U.S. Department of Energy. Scientific Discovery through Advanced Computing: Co-Design Centers, 2012. http://science.energy.gov/ascr/research/scidac/co-design.

[2] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. Benchmarking the effects of operating system interference onextreme-scale parallel machines. *Cluster Computing*, 11:3–16, 2008. 10.1007/s10586-007-0047-2.

[3] S. Böhm and C. Engelmann. xSim: The extreme-scale simulator. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 280–286, 2011.

[4] C. Engelmann and F. Lauer. Facilitating co-design for extreme-scale systems through lightweight simulation. In *IEEE International Conference on Cluster Computing (Cluster): Workshop on Application/Architecture Co-design for Extreme-scale Computing (AACEC)*, pages 1–8, 2010.

[5] K. Ferreira, P. Bridges, R. Brightwell, and K. Pedretti. The impact of system design parameters on application noise sensitivity. *Cluster Computing*, pages 1–13. 10.1007/s10586-011-0178-3.

[6] S. Girona, J. Labarta, and R. M. Badia. "Validation of dimemas communication model for MPI collective operations". In *Lecture Notes in Computer Science: European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, volume 1908, pages 39–46, 2000.

[7] I. S. Jones and C. Engelmann. Simulation of large-scale HPC architectures. In *International Conference on Parallel Processing (ICPP): International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, pages 447–456, 2011.

[8] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng. "Programming petascale applications with Charm++ and AMPI". In *Petascale Computing: Algorithms and Applications*, pages 421–441. 2007.

[9] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing*, pages 139–155. 2008.

[10] P. Kogge et al. ExaScale computing study: Technology challenges in achieving exascale systems. Technical report, Defense Advanced Research Project Agency (DARPA) Information Processing Techniques Office (IPTO), 2008. http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf.

[11] Lawrence Livermore National Laboratory, Livermore, CA, USA. ASC Sequoia Benchmarks, 2012. http://asc.llnl.gov/sequoia/benchmarks.

[12] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 List of Supercomputer Sites, 2012. http://www.top500.org.

[13] A. Morari, R. Gioiosa, R. Wisniewski, F. Cazorla, and M. Valero. A quantitative analysis of os noise. In *Parallel Distributed Processing Symposium (IPDPS), IEEE International*, pages 852 –863, 2011.

[14] 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 *ACM/IEEE conference on Supercomputing*, SC '07, pages 29:1–29:12, 2007.

[15] National Aeronautics and Space Administration. NAS Parallel Benchmarks, 2012. http://www.nas.nasa.gov/Resources/Software/npb.html.

[16] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31, 1995.

[17] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, 2011.

[18] G. Zheng, G. Kakulapati, and L. V. Kale. "BigSim: A parallel simulator for performance prediction of extremely large parallel machines". In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.