SUPPORTING THE DEVELOPMENT OF SOFT-ERROR RESILIENT MESSAGE PASSING APPLICATIONS USING SIMULATION[*]

Christian Engelmann and Thomas Naughton
Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, TN, USA
email: engelmannc@ornl.gov and naughtont@ornl.gov

ABSTRACT
Radiation-induced bit flip faults are of particular concern in extreme-scale high-performance computing systems. This paper presents a simulation-based tool that enables the development of soft-error resilient message passing applications by permitting the investigation of their correctness and performance under various fault conditions. The documented extensions to the Extreme-scale Simulator (xSim) enable the injection of bit flip faults at specific of injection location(s) and fault activation time(s), while supporting a significant degree of configurability of the fault type. Experiments show that the simulation overhead with the new feature is ∼2,325% for serial execution and ∼1,730% at 128 MPI processes, both with very fine-grain fault injection. Fault injection experiments demonstrate the usefulness of the new feature by injecting bit flips in the input and output matrices of a matrix-matrix multiply application, revealing vulnerability of data structures, masking and error propagation. xSim is the very first simulation-based MPI performance tool that supports both, the injection of process failures and bit flip faults.

KEY WORDS
high-performance computing, fault tolerance, parallel discrete event simulation, fault injection

1 Introduction

Today's fastest high-performance computing (HPC) systems are capable of performing more than 10 PFlop/s (1 PFlop/s = 10^{15} floating-point operations per second) using the LINPACK benchmark [14]. These supercomputers typically consist of tens-of-thousands of compute nodes, containing hundreds-of-thousands of compute cores and up to 1 PB of main memory. Supercomputer vendors undertake great efforts in making sure that these systems run reliably within the given cost constraints. On the path to exascale computing [16], with systems operating at and above 1,000 PFlop/s by 2024, resilience against faults is becoming a major challenge due to increasing system sizes and shrinking process technology [6].

Understanding the impact of faults at runtime is an important part of designing HPC system architectures, system software and applications. Soft errors caused by radiation-induced bit flip faults [1] that escape hardware protection schemes, such as single-error correction double-error detection (SECDED) error correcting code (ECC), are of particular concern to HPC applications. These errors may result in software hangs, software crashes, or silent data corruption (SDC) [3].

Recent efforts in developing resilient solvers aim at improving the soft error detection and correction or masking capabilities of HPC applications without the need for full-scale dual or triple modular redundancy. The approaches in these efforts include using resilient data representation [11], algorithm-based fault tolerance (ABFT) [10, 4], and resilient iterative solvers [9]. Evaluating the resilience of these HPC applications requires the injection of bit flip faults and the observation of HPC application behavior, both at runtime, where resilience is measured in correctness and performance under various fault conditions.

1.1 Supporting the Development of Message Passing Applications using Simulation

The difference between the computational peak performance of HPC architectures and the computational performance achieved by different HPC applications can be as high as 50% [14]. Reducing this application-architecture performance gap, i.e., improving architecture efficiency and application productivity, requires an understanding of the runtime performance of applications on today's and tomorrow's HPC architectures.

HPC applications utilize the Message Passing Interface (MPI) for communication between compute nodes. MPI profiling tools aid in evaluating and improving application performance. Simulation-based MPI performance tools permit running a HPC application on a simulated HPC architecture without the need for the actual hardware. Since the runtime overhead of software simulation highly depends on simulation accuracy, a combination of small-scale system simulation with high accuracy and large-scale

[*]This work was sponsored by the U.S. Department of Energy's Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States 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 United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

system simulation with lower accuracy is used.

Similar to simulating future HPC architectures that don't exist yet, simulation-based MPI performance tools can also simulate faulty hardware in a controlled fashion without the need for faulty hardware. This allows observing HPC application correctness and performance under various fault conditions with simulated HPC architectures.

1.2 The Extreme-scale Simulator

The Extreme-scale Simulator (xSim) [5] is such a simulation-based MPI performance tool. It simulates a HPC architecture at scale with lower accuracy using a parallel discrete event simulation (PDES) engine. xSim supports oversubscribed execution, i.e., the simulated HPC system can have significantly more simulated processor cores than the native HPC system xSim runs on. The PDES engine uses processor and network models to account for the simulated time of each simulated MPI process. Experiments using xSim scaled up to 134,217,728 (2^{27}) simulated MPI processes using a 960-core Linux cluster.

xSim uses the MPI performance tool interface, PMPI, to seamlessly intercept MPI calls made by the HPC application. It also intercepts certain POSIX calls, such as for system time and file input/output (I/O). xSim essentially virtualizes the MPI library and certain parts of the POSIX system call interface to simulate the performance properties of a HPC system. It supports C, C++, and Fortran, including 92 MPI functions. Running an MPI application in the simulator requires the following steps:

- Add the xSim header file to the application source code:
  - Add "#include xsim-c.h" for C applications
  - Add "#include xsim-cxx.h" for C++ applications
  - Add "#include xsim-f.h" for Fortran applications
- Recompile the application and link it with the xSim simulator library and the respective xSim programming language interface library:
  - Link with "-lxsim -lxsim-c" for C applications
  - Link with "-lxsim -lxsim-cxx" for C++ applications
  - Link with "-lxsim -lxsim-f" for Fortran applications
- Run the application with:
  - "mpirun -np <physical process cout> <application> -xsim-np <simulated process count> <simulation parameters> <application parameters>"

xSim already supports the injection of MPI process failures during the simulation [15], including proper simulation of MPI process failure detection, notification, and recovery using the user-level failure mitigation (ULFM) extensions [2] proposed by the MPI Fault Tolerance Working Group. It is capable of investigating performance under MPI process failure for resilient solvers using ABFT. xSim is the very first simulation-based MPI performance tool that supports ULFM and ABFT.

1.3 Contribution

While xSim supports the injection of MPI process failures, it does not yet support the injection of bit flip faults and the observation of an application's behavior, both at runtime. This paper details extensions to xSim to offer this capability and to permit the measurement of correctness and performance under various fault conditions for supporting the development of soft-error resilient MPI applications.

The newly added bit flip fault injection feature is novel in several ways: (1) it permits the specification of injection location(s), (2) it allows the specification of fault activation time(s), and (3) it supports a significant degree of configurability to study different fault scenarios in detail. The improved xSim is the very first simulation-based MPI performance tool that supports both, the injection of MPI process failures and the injection of bit flip faults.

2 Technical Approach

The core concept of the new bit flip fault injection feature for xSim relies on separating potential fault location sites from fault activation time and from fault features. Additionally, the use of a simulator offers several features that can be advantageous when developing resilience investigation tools for large scale systems.

2.1 Core Fault Injection Concepts

When evaluating the resilience of an application, *fault site selection* is an important factor to enable a clear understanding of error detection, correction and masking features. While randomly injecting bit flip faults into the process image of an MPI application at runtime can provide some insights, injecting bit flip faults into specific, known data structures yields significantly better insights as the resilience of individual data structures in the context of an application's algorithm is evaluated.

Similar to location selection (fault sites), *fault activation time* is an important factor as well. While injecting bit flip faults into the input data of an MPI application or into MPI messages at runtime can offer some knowledge, injecting bit flip faults into specific, known data structures across the life time of a HPC application yields significantly better knowledge about the masking of faults in the context of an application's algorithm.

When injecting bit flip faults for measuring the correctness and performance of applications, the *fault features* play a crucial role as well. While injecting single bit flip faults can provide some view of the resilience of an application, real-world faults are rarely single bit flips. These faults typically happen when they escape hardware protection schemes, such as SECDED ECC, which catch single bit flip faults. In fact, the failure mode of SECDED ECC is a triple bit flip within an 8-byte word.

2.2 Resilience Investigation Features

xSim is responsible for managing the parallel application, which uses the virtualized MPI and POSIX interfaces. The simulator offers the ability to control elements associated with the system, e.g., processor and network configuration, and process memory management. This affords users of xSim the ability to combine studies focused on application scale-up (performance) with inquiries into application behavior under error (resilience). This underlying control layer (simulator) also provides a useful basis for developing resilience investigation tools.

There are two competing properties that must be managed during fault injection experiments in order to maintain a coherent interaction between the system under test (SUT) (e.g., MPI application) and experiment controller (e.g., xSim). The level of *integration* with the target must be sufficiently close to provide appropriate context, i.e., application context, variables, and algorithmic phases. Also, the degree of *isolation* is crucial for providing protections to avoid corruption between the SUT and the controller.

As such, the simulation based approach to developing resilience investigation tools strikes a good balance between isolation and integration. There is excellent integration with the SUT because the simulator is directly linked into the parallel application and oversees application resource allocations (e.g., memory) during execution. The degree of isolation between the SUT and simulator is moderate, and therefore care is taken when introducing the faults into the target. In xSim, the isolation property is maintained by closely tracking the memory allocations and restricting the data corruptions to application specific regions. Also, the types of faults that can be introduced are closely controlled to ensure coherence of the PDES.

3 Implementation

xSim maintains a stack for each simulated MPI process and performs user-space context switches. The simulator also maintains a global heap for each simulated MPI process, which is saved/restored during a context switch. xSim also tracks the dynamically allocated memory for each simulated MPI process separately. Using these three features, xSim has full access to each simulated MPI process stack, global heap, and dynamically allocated heap.

A potential fault location site is specified either on the command line or registered/deregistered dynamically with the simulator at runtime. A site can be the entire stack, the entire global heap, the entire dynamically allocated heap, or a global variable (symbol). Dynamically registered/deregistered sites additionally include raw addresses that can point to a location in the stack, global heap or dynamically allocated heap. Global variables or raw addresses used as sites are bound by specified sizes. If the global variable or raw address is a pointer to a dynamically allocated heap segment that should be used as a site, the site's size is specified as 0 and looked up at runtime using xSim's dynamic memory allocation tracking feature.

Bit flip faults are injected by creating a schedule for injections using simulated MPI rank and simulated MPI process time pairs. Upon the time of a scheduled injection, xSim randomly choses an address within the specified injection sites. xSim does not have fine-grain control over the simulated MPI processes, i.e., employs cooperative process scheduling. As such, a simulated MPI process needs to yield to xSim, e.g., via a simulated MPI or POSIX call, to allow for injection. The injection schedule is checked for any bit flip faults that need to be activated prior to xSim relinquishing control back to the application. To permit fine-grain injections, a `check()` function has been added to xSim to temporarily hand over control to the simulator to check the injection schedule.

The type of bit flip fault to be injected is specified on the command line by two values. The first value is the number of bits to flip, while the second value is the number of consecutive bytes to consider for randomly flipping the bits in. The randomly chosen address within the specified injection sites is aligned with the number of consecutive bytes. For example, randomly flipping 3 bits in 8 consecutive bytes will simulate faults that escape SECDED.

4 Experimental Evaluation

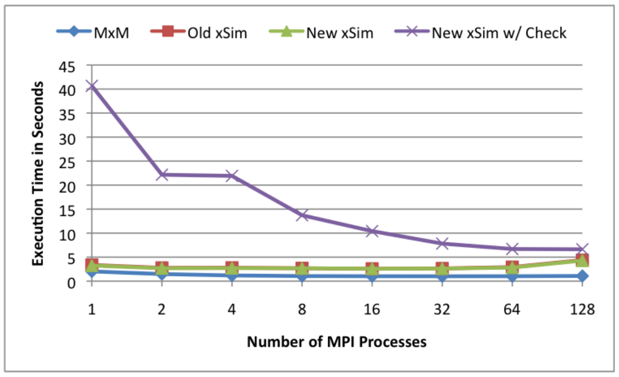
The simulator performance overhead and simulation accuracy of the newly added bit flip fault injection feature has been evaluated and bit flip fault injection experiments were performed to demonstrate the usefulness of the new feature.

4.1 Evaluation Setup

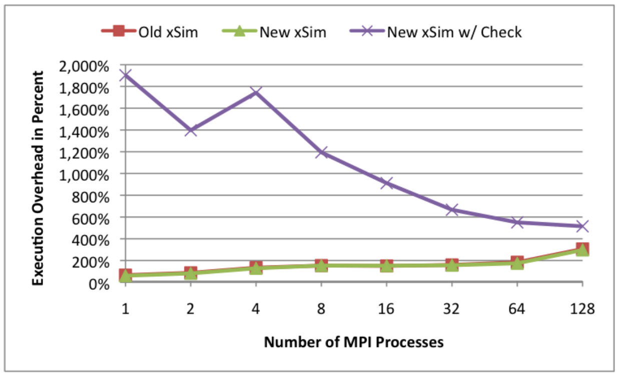
The experiments were performed on a 128-core Linux cluster computer with 16 compute nodes, two 2.4 GHz AMD Opteron 2378 processors per node, 4 cores per processor, 8 GB RAM per node, and a non-blocking 1 Gbps Ethernet interconnect. The system is running Ubuntu 14.04 LTS, Open MPI 1.10.1, and GCC 4.8.

The simulator performance evaluation, simulation accuracy evaluation and fault injection experiments utilize a basic parallel matrix-matrix multiply application. This application was chosen as it is deterministic even when bit flip faults are injected at runtime into the input/output matrices, unlike iterative solvers that can observe a change in convergence rate. The main purpose of the experiments is to demonstrate the usefulness of the new fault injection feature, which can be easier done with a deterministic application. The basic parallel matrix-matrix multiply application simply reads in the input matrices A and B from files on MPI process 0 and distributes the necessary chunks of matrix A to the other MPI processes. The matrix B is broadcast to the other MPI processes. Each MPI process performs its portion of the matrix-matrix multiplication $C = A \times B$, i.e., $C_p = A_p \times B$, and then sends its partial result matrix C_p back to MPI process 0. The individual partial result matrices C_{1-n} are received by MPI process 0 and the final result matrix C is saved to a file. The `check()` function is optionally called before every multiplication of a matrix element to permit fault injection right at that point in time.

xSim was configured to simulate the processor and network performance of the 128-core Linux cluster. A scaling processor model was configured with a scaling factor of 1.0 to match simulated and native execution time. The network model was configured with a shared memory model

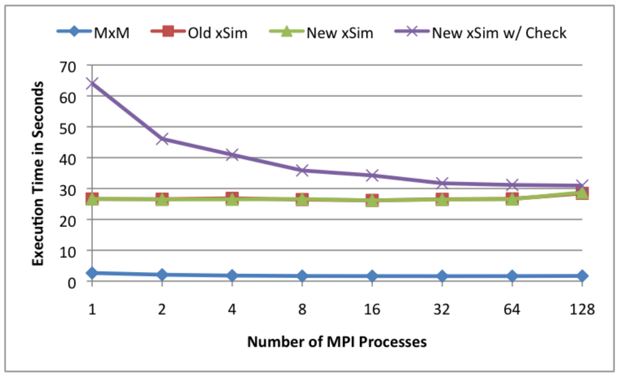


(a) Comparison of execution times in seconds

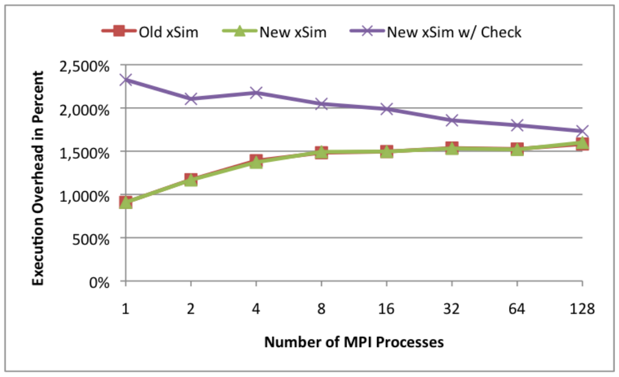


(b) Simulator execution overheads in percent

Figure 1. Performance evaluation of the matrix-matrix multiply application vs. running it atop the old xSim vs. running it atop the new xSim, all without file I/O



(a) Comparison of execution times in seconds



(b) Simulator execution overheads in percent

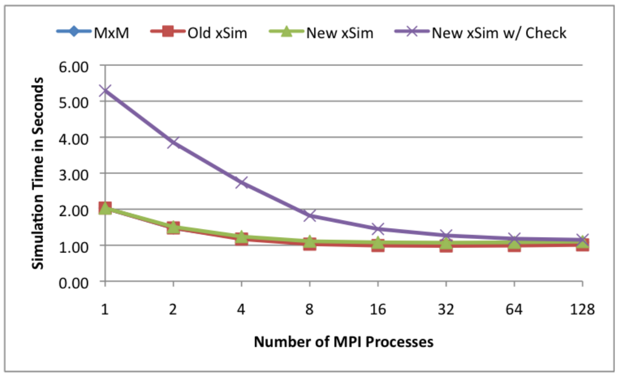
Figure 2. Performance evaluation of the matrix-matrix multiply application vs. running it atop the old xSim vs. running it atop the new xSim, all with file I/O

for the 8 on-node processor cores and a Gigabit Ethernet model for the network interconnect. It was also configured with contention modeling for on- and off-node traffic. The file system model of xSim is a new and untested feature and was switched off, i.e., file I/O calls do not incur any simulated time. Every POSIX file I/O call, however, is intercepted by xSim. To permit POSIX-compliant file system simulation, the current implementation globally synchronizes the PDES engine on every POSIX file I/O call to serialize all POSIX file I/O calls in simulated time. This can lead to excessive runtime overheads. Since the matrix-matrix multiply application reads in and writes out each matrix element with a `fscanf()` or `fprintf()` call, fault injection can be performed when xSim relinquishes control back to the application after each call.

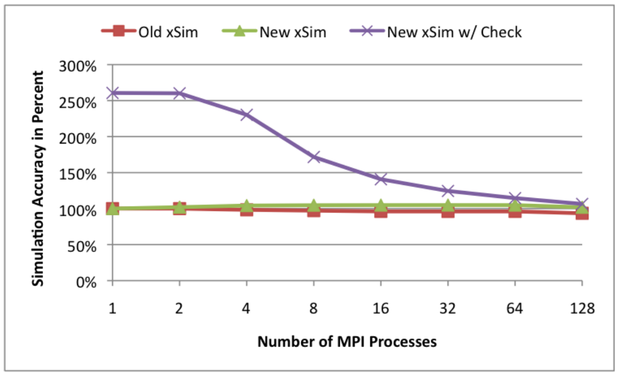
4.2 Simulator Performance Evaluation

To assess the overhead of the newly added bit flip fault injection feature, the basic parallel matrix-matrix multiply application was executed using 512×512 matrices containing double-precision values, (1) natively, (2) atop the previous version of xSim and (3) atop the new version of xSim. The conducted experiments scale from 1 to 128 MPI processes on the 128-core Linux cluster, where each simulated MPI process resides in a physical MPI process. No oversubscription was employed, where multiple simulated MPI processes reside in a physical MPI process, to offer a fair comparison. Each data point in the experiments is an average that was obtained from 10 different runs.

Figure 1 shows the wall-clock execution time in seconds and corresponding simulator overheads in percent of the native application (MxM), the previous version of xSim (Old xSim), the new version of xSim without calling the `check()` function before every matrix element multiplication (New xSim), and the new version of xSim with calling the `check()` function (New xSim w/ Check), all without reading in the input files and writing out the output file. The native application execution (MxM) drops to only 53% of its serial execution time with 128 MPI processes, as the amount of parallelized computation time is mostly offset by the amount of communication. The previous version of xSim (Old xSim) and the new version of xSim without calling the `check()` function exhibit the same simulator overheads (∼60% for serial execution and ∼300% at 128 MPI processes). The new version of xSim with calling the `check()` function (New xSim w/ Check) shows a significant simulator overhead increase that scales with the amount of computation time (∼1,900% for serial execution and ∼515% at 128 MPI processes). The performance impact of the $512 \times 512 \times 512 = 134,217,728$ (O(n^3)) `check()` calls evenly distributed over the MPI processes is clearly visible. As the `check()` calls are getting distributed over more MPI processes, the overhead correspondingly decreases.

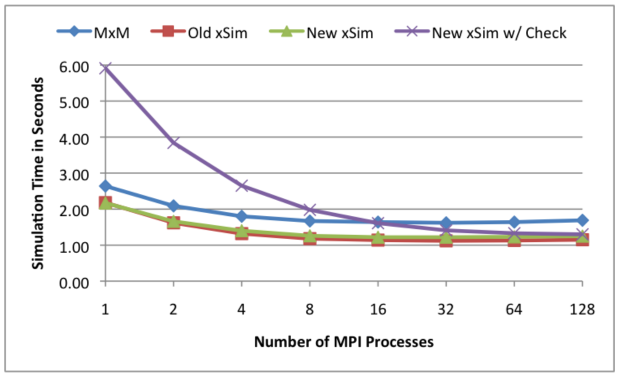


(a) Comparison of native vs. simulated execution times in seconds

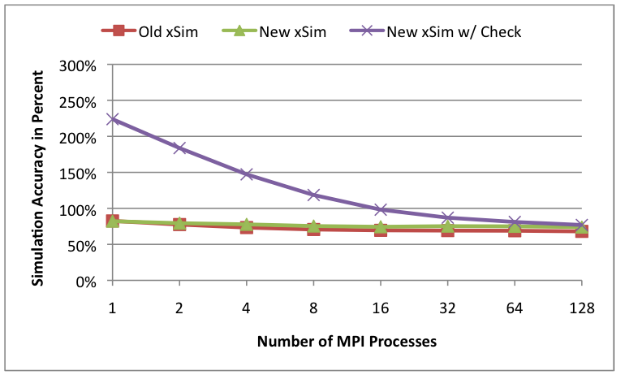


(b) Simulation accuracy in percent

Figure 3. Simulation accuracy evaluation of the the old xSim vs. the new xSim with the matrix-matrix multiply application without file I/O



(a) Comparison of native vs. simulated execution times in seconds



(b) Simulation accuracy in percent

Figure 4. Simulation accuracy evaluation of the the old xSim vs. the new xSim with the matrix-matrix multiply application with file I/O

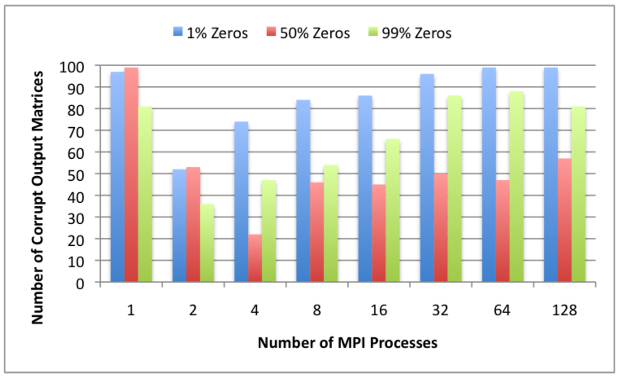
Figure 2 illustrates the wall-clock execution time and corresponding simulator overheads with reading in the input files and writing out the output file. The native application execution (MxM) drops to only 64% of its serial execution time with 128 MPI processes, as the additional amount of time needed for file I/O is not parallelized. The previous version of xSim (Old xSim) and the new version of xSim without calling the `check()` function exhibit the same simulator overheads (∼910% for serial execution and ∼1,600% at 128 MPI processes). The increase in overhead in comparison to Figure 1 is entirely due to the file system model. Each of the $2 \times 512 \times 512 = 524,288$ (O(n^2)) `fscanf()` and $512 \times 512 = 262,144$ (O(n^2)) `fprintf()` calls stops and synchronizes the simulation. Each call also checks for a fault injection upon returing to the application. The new version of xSim with calling the `check()` function (New xSim w/ Check) shows again a significant simulator overhead increase that scales with the amount of computation time (∼2,325% for serial execution and ∼1,730% at 128 MPI processes). This execution has a total of 135,004,160 potential injection points just from the file I/O and `check()` function calls.

4.3 Simulation Accuracy Evaluation

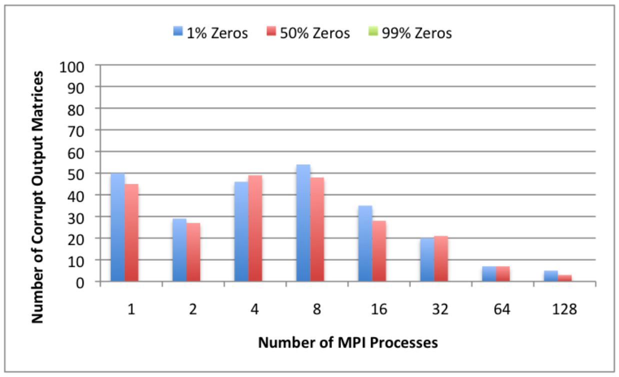
The simulator performance evaluation experiments were also used to evaluate the simulation accuracy. A simulation is considered accurate if the simulated execution time matches 100% of the native execution time.

Figure 3 shows the simulation accuracy of the previous version of xSim and the new version of xSim in comparison to the native execution of the application, all without reading in the input files and writing out the output file. The previous version of xSim and the new version of xSim without calling the `check()` function exhibit similar simulation accuracy (100% for serial execution for both, ∼94% at 128 MPI processes for the previous version of xSim, and ∼102% for the new version). The new version of xSim with calling the `check()` function shows a significant simulation accuracy decrease that scales with the amount of computation time (∼261% for serial execution and ∼106% at 128 MPI processes). This is expected as the `check()` function is called before every multiplication, reducing processor cache efficiency. This increases the runtime of the application in simulated time. Such fine-grain instrumentation for fault injection always interferes with the system under test.

Figure 4 illustrates the simulation accuracy of the previous and the new version of xSim in comparison to the application with reading in the input files and writing out the output file. The impact of the disabled file system model is clearly visible. The previous and the new version of xSim without calling the `check()` function exhibit similar simulation accuracy (83% for serial execution for both, ∼86% at 128 MPI processes for the previous version of xSim, and

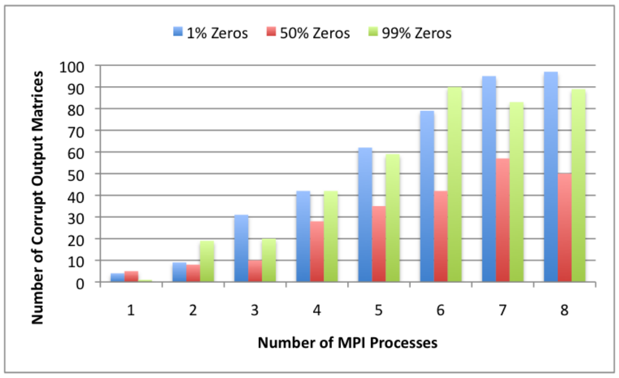


(a) Matrix A injections

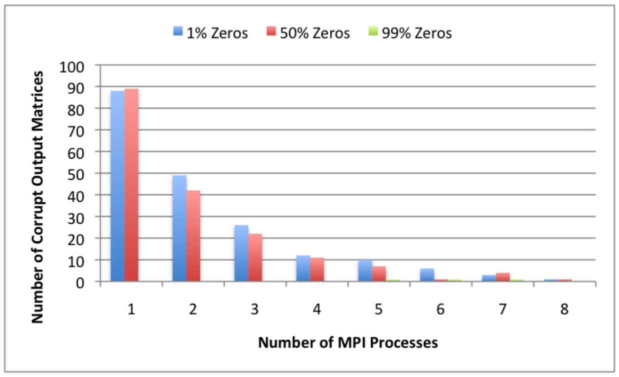


(b) Matrix C injections

Figure 5. The number of output matrices C corrupted by fault injection with calling the `check()` function



(a) Matrix A injections



(b) Matrix C injections

Figure 6. The number of output matrices C corrupted by fault injection without calling the `check()` function

$\sim$74% for the new version). Due to the disabled file system model, the application executes faster in simulated time as the time needed for file I/O is not simulated. The new version of xSim with calling the `check()` function shows again a significant simulation accuracy decrease that scales with the amount of computation time ($\sim$224% for serial execution and $\sim$77% at 128 MPI processes). The faster execution in simulated time due to the disabled file system model is partially offset by the slower execution in simulated time due to the `check()` function.

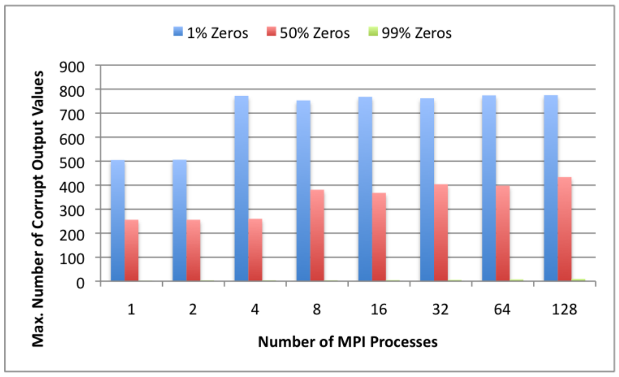
4.4 Fault Injection Campaign

To demonstrate the usefulness of the new feature, bit flip fault injection experiments were performed using the new version of xSim with POSIX file I/O, with and without calling the `check()` function for 512×512 matrices containing double-precision values. A random single SECDED ECC protection failure is injected in the form of a triple-bit flip in an 8-byte word. The injection is performed in the input matrix A or the output matrix C at a random location, a random point in time and at a random MPI process during the execution. The experiment is conducted from 1 to 128 MPI processes with 100 runs at each scale. As the injection is random, its impact depends on the time frame of vulnerability of the particular injection location. With a large enough number of runs with different injection times and locations within the same data structure on different MPI processes, the overall vulnerability of that particular data structure is revealed. As the impact also depends on the data itself, input matrices containing 1%, 50% and 99% zero values are used to demonstrate algorithmic masking. A total of 9,600 runs were executed.

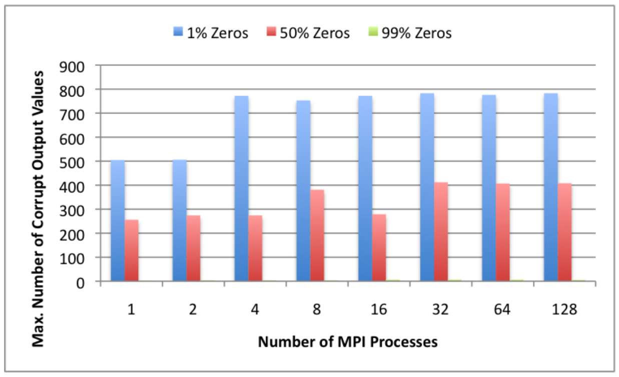
Figure 5 shows the number of output matrices C corrupted by a single triple-bit 8-byte flip fault injection in the input matrix A or C with calling the `check()` function. For all cases (1%, 50% and 99% zero values), the number of output matrices C corrupted by matrix A fault injections initially decreases and then increases with the number of MPI processes (Figure 5(a)). The minimum is at 2 MPI processes for inputs containing 1% zero values, at 4 MPI processes with 50% zeros, and at 2 with 99% zeros. At 1 and 2 MPI processes, inputs containing 99% zero values produce the least amount of corrupt outputs, while inputs containing 50% zeros produce the least amount of corrupt outputs at higher scales. At 1 and 2 MPI processes, inputs containing 50% zeros produce the most number of corrupt outputs, while inputs containing 1% zeros produce the most number at higher scales. For input matrices containing 1% and 50% zeros, the number of output matrices C corrupted by matrix C fault injections initially decreases, then increases, and then decreases again with the number of MPI processes (Figure 5(b)). The amount of corrupt outputs is close between both cases at each scale. For inputs containing 99% zeros, at most 1 corrupt output was produced.

Figure 6 illustrates the number of output matrices C corrupted by a single triple-bit 8-byte flip fault injection in the input matrix A or C without calling the `check()` function. In comparison to Figure 5, the impact of not calling the `check()` function is clearly visible. The initial high vulnerability of matrix A at lower scales when calling the `check()` function is not present without calling it. The initial trend of decreasing vulnerability of matrix A is not visible either. Only the overall trend of higher vulnerability of matrix A when scaling up is similar, as well as the differences between inputs containing 1%, 50% and 99% zeros. While inputs containing 1% and 99% zeros produce a somewhat similar amount of corrupt outputs with injected faults, inputs containing 50% zeros produce less. Also, the initial lower vulnerability of matrix C at lower scales when calling the `check()` function is not present without it. The increase in vulnerability of matrix C at 4, 6 and 16 MPI processes is also not displayed. The differences between inputs containing 1%, 50% and 99% zeros are similar, as inputs with 1% and 50% zeros produce about the same number of corrupt outputs and inputs with 99% zeros create at most 1. This result clearly shows the need for the `check()` function to get more insights about the vulnerability of data structures at runtime.

Figure 7 shows the maximum number of output values in matrix C corrupted by fault injection in matrix A in a single run, with and without the `check()` function. There is no impact for calling the `check()` function. Less corrupt output values are being created at 1 and 2 MPI processes. Inputs with 1% zeros produce the most corrupt output values, while inputs containing 50% zeros create about half, and inputs with 99% zeros only 10 or less.



(a) With calling the `check()` function



(b) Without calling the `check()` function

Figure 7. The maximum number of output values in matrix C corrupted by fault injection in matrix A

5 Related Work

There have been several recent works that have used fault injection to study the resilience in HPC. We briefly highlight a few that are directly relevant to the current work, namely: MPI, interposition libraries, and soft error studies.

In the FAIL-FCI/FAIL-MPI [7, 8] work, a language was developed for users to describe fault injection experiments along with supporting infrastructure for managing the distributed execution of fault injection campaigns. The distinction between the FCI and MPI variants had to do with management of task launching/attaching in distributed environments. The FAIL description language supported different error activation methods, these "fault triggers" were *onload*, *onexit* and *onerror* [7]. In contrast to the current work, xSim's fault injection method integrates with the simulator and therefore is focused on a "time based" trigger. The FAIL works used a debugger style mechanism to inject faults, i.e., debugger process launch/attach.

The library level fault injection (LFI) [13] work provided a general tool that used the `LD_PRELOAD` mechanism to interpose on application execution when calls are made to shared libraries. This is similar to the mechanism used by xSim for interposing on application routines and tracking variable usage (symbols), where no direct source code modifications[1] are necessary to enable/disable injection sites at runtime. However, the xSim fault injection mechanism is fully aware of the discrete event simulation and ensures proper causality for resilience experiments.

The work by Li et al. [12] used binary instrumentation to introduce errors via the BIFIT tool to study soft errors in HPC applications. They gathered memory profiles to identify target locations for injections and then dynamically instrumented the binaries to inject bit flips into globals, heap and stack data. They observed that the time and location of the injection is significant for each application and that for all three applications (Nek50000, S3D, and GTC), global data was significant to influence the application's output and execution state. The fault injection mechanisms in xSim allow for very fine-grained control over the time and location of the injections. The memory profiling could also be employed with the current xSim mechanisms to tailor site selection.

6 Conclusion

This paper presented recent improvements to the Extreme-scale Simulator (xSim). A newly added bit flip fault injection feature allows an MPI application developer to observe application correctness and performance under vari-

[1]Note, the previously discussed xSim `check()` function can be added to the application's source code to reduce fault injection latency.

ous fault conditions with simulated HPC architectures. The novel bit flip fault injection feature permits the specification of injection location(s), allows the specification of fault activation time(s), and supports a significant degree of configurability to study different fault scenarios in detail. With this new feature, xSim is the very first simulation-based MPI performance tool that supports both, the injection of MPI process failures and the injection of bit flip faults.

The evaluation showed that the simulation overhead of the new feature is $\sim$2,325% for serial execution and $\sim$1,730% at 128 MPI processes for very fine-grain fault injection capability. The fault injection experiments demonstrated the usefulness of the new feature by injecting SECDED ECC protection failures in the input matrix A and the output matrix C of a basic parallel matrix-matrix multiply application. The results revealed the error vulnerability of these data structures, certain algorithmic masking effects and specific algorithmic error propagation effects.

Future work will focus on an efficient and accurate file I/O simulation implementation. Further planned work includes different fault injection features that mimic the behavior of faulty logic, such as of a faulty processor.

References

[1] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability (TDMR)*, 5(3):305–316, 2005.

[2] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of user-level failure mitigation support in MPI. In *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 193–203, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] G. Bronevetsky and B. R. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the* 21^{st} *ACM International Conference on Supercomputing (ICS) 2008*, Island of Kos, Greece, June 7-12, 2007. ACM Press, New York, NY, USA.

[4] Z. Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proceedings of the* 20^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2006*, page 10, Rhodes Island, Greece, Apr. 25-29, 2006. IEEE Computer Society.

[5] C. Engelmann. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Generation Computer Systems (FGCS)*, 30(0):59–65, Jan. 2014.

[6] J. D. et al. Inter-agency workshop on HPC resilience at extreme scale, Feb. 21-24, 2012.

[7] W. Hoarau, P. Lemarinier, T. Herault, E. Rodriguez, S. Tixeuil, and F. Cappello. FAIL-MPI: How Fault-Tolerant Is Fault-Tolerant MPI? In *Proceedings of the International Conference on Cluster Computing*. IEEE, Sept. 2006.

[8] W. Hoarau, S. Tixeuil, and F. Vauchelles. FAIL-FCI: Versatile fault injection. *Future Generation Computer Systems*, 23(7):913 – 919, 2007.

[9] M. Hoemmen and M. A. Heroux. Fault-tolerant iterative methods via selective reliability. Technical report, Sandia National Laboratories, Albuquerque, MN, USA, 2012.

[10] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers (TC)*, C-33(6):518–528, 1984.

[11] M. M. Latif, R. Ramaseshan, and F. Mueller. Soft error protection via fault-resilient data representations. In *Proceedings of the 3rd Workshop on Silicon Errors in Logic - System Effects (SELSE) 2007*, Apr. 2007.

[12] D. Li, J. S. Vetter, and W. Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 57:1–57:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[13] P. D. Marinescu and G. Candea. LFI: A Practical and General Library-Level Fault Injector. In *Proceedings of the* 39^{th} *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09)*. IEEE, June 29 - July 2, 2009.

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

[15] T. Naughton, C. Engelmann, G. Vallée, and S. Böhm. Supporting the development of resilient message passing applications using simulation. In *Proceedings of the* 22^{nd} *Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP) 2014*, pages 271–278, Turin, Italy, Feb. 12-14, 2014. IEEE Computer Society, Los Alamitos, CA, USA. Acceptance rate 32.6% (73/224).

[16] US Department of Energy. DOE Exascale Strategy - Report to Congress. http://assets.fiercemarkets.net/public/sites/govit/perera_fgit_foia_doe_exascale%20report.pdf, June 2013.