Modeling Application Resilience in Large-scale Parallel Execution

Kai Wu
University of California, Merced
kwu42@ucmerced.edu

Wenqian Dong
University of California, Merced
wdong5@ucmerced.edu

Qiang Guan
Kent State University
qguan@kent.edu

Nathan DeBardeleben
Los Alamos National Laboratory
ndebard@lanl.gov

Dong Li
University of California, Merced
dli35@ucmerced.edu

ABSTRACT

Understanding how the application is resilient to hardware and software errors is critical to high-performance computing. To evaluate application resilience, the application level fault injection is the most common method. However, the application level fault injection can be very expensive when running the application in parallel in large scales due to the high requirement for hardware resource during fault injection.

In this paper, we introduce a new methodology to evaluate the resilience of the application running in large scales. Instead of injecting errors into the application in large-scale execution, we inject errors into the application in small-scale execution and serial execution to model and predict the fault injection result for the application running in large scales. Our models are based on a series of empirical observations. Those observations characterize error occurrences and propagation across MPI processes in small-scale execution (including serial execution) and large-scale one. Our models achieve high prediction accuracy. Evaluating with four NAS parallel benchmarks and two proxy scientific applications, we demonstrate that using the fault injection result to predict for 64 MPI processes, the average prediction error is 8%. While using the fault injection result to make the same prediction for eight MPI processes, the average prediction error decreases to 7%.

ACM Reference Format:
Kai Wu, Wenqian Dong, Qiang Guan, Nathan DeBardeleben, and Dong Li. 2018. Modeling Application Resilience in Large-scale Parallel Execution. In *Proceedings of 47th International Conference on Parallel Processing (ICPP 2018).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225119

1 INTRODUCTION

High-performance computing (HPC) applications are at a significant risk of being hit by hardware and software errors in the future extreme-scale systems. Understanding how the application is resilient to those errors is critical to ensure computation result correctness and design efficient fault tolerance mechanisms. The challenge of *application resilience* is one of the grand challenges facing the future extreme-scale systems.

To evaluate application resilience, fault injection at the application level is the most common method. The application level fault injection usually triggers random bit-flip in an input or output operand of a random instruction as a fault injection test. Typically, the statistical result of many fault injection tests is used to evaluate application resilience. It has been shown that HPC application resilience could vary in different execution modes. In particular, given a fixed input problem for an application, application resilience can be different from the execution in a small scale with less MPI processes (small-scale execution) to the execution in a large scale with more MPI processes (large scale execution) [37].

In addition, the application level fault injection can be expensive: one has to perform a large number of fault injection tests to ensure statistical significance, hence consuming a lot of hardware resource. This is particularly problematic for evaluating the resilience of the application running in large scales. In a large-scale execution of the application, one has to ask for many nodes to deploy application execution and perform fault injection. However, given the limited hardware resource in a supercomputer, a job using many nodes to deploy a large-scale execution of the application is usually assigned with low priority by the scheduler system in HPC. It is therefore difficult to obtain enough hardware resource to evaluate the resilience of the application running in large scales.

Furthermore, to study the resilience of a given application, the large-scale execution can significantly increase fault injection time, (compared with the case in small-scale execution), making it intractable to perform a comprehensive analysis. In particular, some fault injection tools, such as F-SEFI [13], BIFIT [21] and pinFI [36], are based on binary instrumentation. The overhead of those tools is directly related to the number of instructions. A large-scale execution can increase the number of instructions, hence increasing the overhead of those fault injection tools. Our study on one NAS parallel benchmark [3] CG reveals that the execution with just four MPI processes increases the number of instructions by 74.5% than serial execution. Using F-SEFI as our fault injection tool, the fault injection time with four MPI processes increases by 58% than with serial execution, although the serial execution and parallel execution use the same number of fault injection tests and the two executions without F-SEFI only differ by 15% in execution time. Such execution time overhead will even grow drastically as we use a larger number of MPI processes for fault injection.

LA-UR-17-29747

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225119

In this paper, we introduce a new methodology to evaluate the resilience of the application running in large scales. Instead of injecting errors into the application in large-scale execution, we inject errors into the application in small-scale execution and serial execution to model and predict the fault injection results for the application running in large scales.

Our models are based on a series of empirical observations. These observations reveal that there exists a correlation between small-scale execution (including serial execution) and large-scale one. Such correlation provides a foundation to construct our models. In particular, we observe that without the consideration of MPI communication in parallel execution, the computation difference (in terms of code structures) between small scale and large scale executions are often small. For the common computation in serial and parallel executions, injecting multiple errors into serial execution can be used to emulate error contamination in multiple MPI processes in a larger scale. Furthermore, we observe that error propagation across MPI processes on a small scale often gives indications on how error propagation across MPI processes happens on a larger scale. Based on the above observations, we introduce models that use serial execution to capture the effects of error contamination on multiple MPI processes and use a small scale execution to capture error propagation across MPI processes, for modeling the resilience of the application running in larger scales.

The future HPC applications are expected to be deployed in larger system scales to improve simulation speed and accuracy. We vision that evaluating application resilience in large-scale execution with fault injection will become more and more challenging. Our modeling methodology avoids high requirement on hardware resource to evaluate application resilience in large-scale execution. Hence our modeling methodology makes the evaluation more feasible and helps future HPC applications survive ever-increasing threats from the future HPC resilience challenge.

Furthermore, our work introduces a fundamentally new methodology to study application resilience. For the first time, we reveal that application resilience in different execution modes is correlated. Such correlation allows us to do fault injection and study application resilience in an easy and controllable environment.

The contributions of this paper are summarized as follows.

- A new methodology to model and predict the resilience of the application running in large scales;
- The characterization of error occurrences and propagation across MPI processes in small and large scales;
- The evaluation of the modeling methodology and demonstrating high modeling accuracy.

The remainder of the paper is organized as follows. Section 2 provides background information on fault models, fault injection and our assumptions on common HPC applications. Section 3 characterizes application resilience in small scale and large scale executions, and examine application resilience difference in the two scales from the perspective of error occurrences and propagation. Section 4 explains our models in details. Section 5 evaluates our modeling accuracy and further studies the correlation between the modeling accuracy, sampling granularity and fault injection execution time overhead. Section 6 gives detailed descriptions of prior work, and Section 7 summarizes the contribution of this work.

2 BACKGROUND AND EXPERIMENT DEPLOYMENT

In this section, we introduce fault models used for modeling and fault injection. We explain how to deploy fault injection tests and what kind of HPC applications we are addressing.

Fault models and fault injection. The fault injection is the most common methodology to study application resilience. In our fault injection tests, we randomly choose an instruction during application execution and then randomly trigger a bit flip in one of the instruction operand(s) as an error. We use single-bit flip for fault injection tests in this paper, similar to the existing work [6, 7, 15, 21, 30]. The single-bit flip is also the most common fault pattern in large-scale HPC data centers [32, 33]. However, our modeling methodology is general and does not make any assumption that the injected error must be single-bit flip.

When doing fault injection, we distinguish instructions by instruction type (e.g., floating point instructions, memory load/store instructions) to characterize application resilience. This is based on our observations that the fault injection result is sensitive to what type of instruction is randomly selected for fault injection [12]. In this paper, we always choose floating point instructions (floating point addition and multiplication) for fault injection, because those instructions are very common in HPC and errors in them are among the most difficult errors to detect in HPC. Errors in other common instructions, such as integer instructions, are often related to memory pointers, loop iterator and array index in HPC applications. Such errors are relatively easy to identify because of their big impact on the application (e.g., application crash). However, our modeling methodology is general and does not make any assumption on which specific instruction type should be considered for modeling.

For each *fault injection deployment*, we perform a large number of fault injection tests to establish the statistical significance of the fault injection result. A fault injection deployment uses a specific fault injection configuration. Such configuration includes a specific number of MPI processes to run the application and a specific fault pattern for fault injection (e.g., how many errors are injected? what type of instruction is selected for fault injection? Are we using a single-bit flip or multiple-bit flip?).

Each fault injection deployment, after a large number of fault injection tests are done, will generate a fault injection result. The fault injection result is a statistical summary of all fault injection tests. Each *fault injection test* can have at least three outcomes. The three outcomes are described as follows.

- Silent Data Corruption (SDC): The application output under a fault injection test is different from the fault-free run.
- Success: Two cases are considered "success". 1). The application output under a fault injection test is different from the fault-free run, but the output successfully passes the application "checkers" and shows valid. 2). The application output under a fault injection test is exactly same as the fault-free run.
- Failure: The application crashes or hangs.

The *fault injection result* for a specific outcome (SDC, success, or failure) is the percentage of fault injection tests that have the outcome. For example, given a fault injection deployment, the fault

injection result for the "success" outcome is 20%, which means 20% of all fault injection tests have "success" outcome. We also use the word, *success rate*, to refer to the success outcome.

To ensure statistic significance of the fault injection result, we must perform a sufficiently large number of fault injection tests [19]. The previous study has shown that the fault injection result without sufficient fault injection tests will be different from otherwise [14]. In our fault injection study, we perform a large number of fault injection tests (4000 tests) for each fault injection deployment, such that further increasing the number of fault injection tests does not cause big variation (less than 10%) in the fault injection result. This method ensures that our fault injection is sufficient and our fault injection result is statistically correct.

For a fault injection test in parallel execution, we inject an error into application computation, not MPI communication. Studying the impact of communication corruption is out of the scope of this paper, but the injected error in the computation of one MPI process may propagate to other MPI processes through MPI communication. Our models capture error propagation across MPI processes through MPI communication.

Fault injection tool. We use Fine-grained Soft Error Fault Injector (F-SEFI) [13] as our fault injection tool. F-SEFI is a software fault injection tool developed by Los Alamos National Laboratory. F-SEFI capitalizes on the open source QEMU virtual machine hypervisor and works as a pluggable module. F-SEFI utilizes the QEMU virtual machine hypercalls to inter-communicate with the guest virtual machines (VMs). QEMU is widely used for emulating guest architectures which are different from the host physical architecture. F-SEFI inherits this feature and allows fault injections into architectures that are prototypes or are physically unavailable.

F-SEFI supports serial execution of the application. We use an enhanced version of F-SEFI [12] that supports parallel execution of the application based on MPI. Furthermore, F-SEFI and its new version allow us to map the selected instructions for fault injection into the application, such that we can know where the error happens at the application level [4]. The new version of F-SEFI also allows us to track which MPI processes are contaminated during application execution after an error is injected into one MPI process.

HPC applications and their executions. Our study in this paper involves application execution at different scales: serial execution, small-scale execution, and large-scale execution. Given an HPC application, we assume that those executions at different scales use the same input problem size. This means that we consider the strong scaling execution in this paper.

Our model targets a set of *common HPC applications*. Those HPC applications have the following characteristics.

(1) Serial execution, small-scale parallel execution, and large-scale parallel execution must perform similarly in terms of computation (i.e., they must use the same numerical algorithms). If the three executions have very different computation, then using serial and small-scale parallel executions cannot predict application resilience in large-scale parallel execution.

(2) We assume that all MPI processes within the application are doing the same computation. We make such assumption because our fault injection tests randomly choose one MPI process for fault injection. If there is a difference between MPI processes, then depending on which MPI process is chosen for fault injection, we could have different fault injection result. We do not consider the difference between MPI processes to enforce full randomness in fault injection. However, if there is a difference between MPI processes, then the fault injection tests must consider which MPI process should have errors injected.

3 CHARACTERIZATION OF EXECUTION DIFFERENCE BETWEEN SMALL AND LARGE SCALE EXECUTIONS

In this section, we study execution difference between small-scale (including serial execution) and large-scale executions. Such execution difference includes computation, error propagation, and application resilience. The study in this section motivates our model construction in the next section.

3.1 Characterizing Computation Difference between Serial and Parallel Executions

Compared with serial execution, parallel execution can have slightly different computation, even though both executions use the same numerical algorithms. *Without the consideration of MPI communication in parallel execution*, computation difference between the two executions manifests as executing different code structures. Understanding such computation difference is important for our model construction. If there is a big difference in the computation between the two executions, then using serial execution to make the prediction on parallel execution may not be possible.

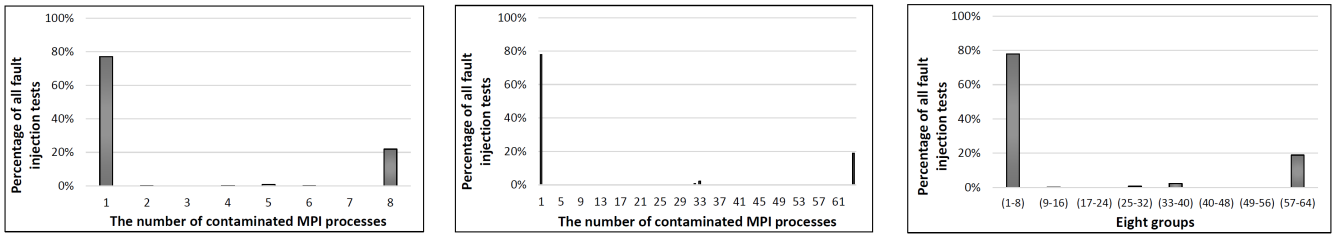
We study four benchmarks (CG, FT, MG, and LU) from NAS parallel benchmark suite (NPB) [3] and two proxy scientific applications MiniFE [17] and PENNANT [18]. We focus on the main computation loop in those benchmarks because the main computation loop dominates computation time and is highly possible to have errors injected. We have two observations on the computation difference.

Observation 1: *All computation that happens in serial execution also happens in parallel execution. We call such computation common computation. Some computation that happens in parallel execution does not happen in serial execution. We call such computation parallel-unique computation.*

The parallel-unique computation is typically used for preparing boundary data in the computation domain or preparing intermediate results before exchanging them between MPI processes. Fault injection that happens in the parallel-unique computation will not be able to be captured by serial execution. However, the parallel-unique computation usually takes a small portion of the total execution time in parallel execution, shown in Table 1. Table 1 shows the percentage of the parallel-unique computation in the total execution time of parallel execution (four MPI processes).

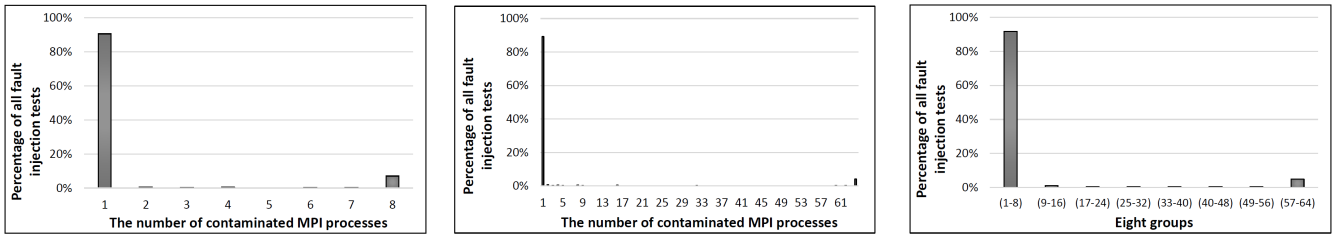
Table 1 reveals that except FT, all benchmarks have very small parallel-unique computation time (less than 2%). FT has relatively large parallel-unique computation time because of the computation in the transpose operation. The above observation lays the foundation for us to use serial execution to estimate application resilience of parallel execution.

Observation 2: *The parallel-unique computation usually takes a small portion of total execution time.*



(a) The profiling result for the small scale **(b) The profiling result for the large scale** **(c) Splitting 64 error propagation cases in the figure (b) into eight groups**

Figure 1: The profiling results for error propagation across MPI processes in a small scale (8 MPI processes) and a larger scale executions (64 MPI processes) for CG.



(a) The profiling result for the small scale **(b) The profiling result for the large scale** **(c) Splitting 64 error propagation cases in the figure (b) into eight groups**

Figure 2: The profiling results for error propagation across MPI processes in a small scale (8 MPI processes) and a larger scale executions (64 MPI processes) for FT.

Table 1: Percentage of the parallel-unique computation in the total execution time of parallel execution

Benchmarks	Percentage of the parallel-unique computation
CG (Class S)	1.6%
CG (Class B)	0.27%
FT (Class S)	10.4%
FT (Class B)	17.7%
MG	No parallel-unique comp
LU	No parallel-unique comp
MiniFE (nx=30 ny=30 nz=30)	1.54%
MiniFE (nx=300 ny=300 nz=300)	0.68%
PENNANT	No parallel-unique comp

3.2 Characterizing Difference in Error Propagation

Once an error is injected into one MPI process, the error can propagate to other MPI processes through MPI communication. Serial execution cannot capture how the error propagates to other MPI processes. We study error propagation across MPI processes in a small scale execution and study how it is correlated to error propagation across MPI processes in a larger scale execution.

We run all benchmarks with four or eight MPI processes as a small scale execution and with 64 MPI processes as a large-scale execution. We perform a large number of fault injection tests as described in Section 2, for both scales. In each fault injection test, we inject one error into *one MPI process*. We profile that how many MPI processes are contaminated in each fault injection test.

Figure 1.a shows the profiling result for CG in the small-scale execution (eight MPI processes). Among fault injection tests, 22% of fault injection tests has eight MPI processes contaminated; 77% of fault injection tests have only one MPI processes contaminated; the other error propagation cases with different numbers of contaminated MPI processes are rare. We have the similar observation for FT, shown in Figure 2.a.

Figure 1.b shows the profiling result for CG in the large scale (64 MPI processes). To reveal the similarity of error propagation between the small scale and large scale executions, we evenly split the 64 error propagation cases shown in Figure 1.b into eight groups (i.e., eight error propagation cases per group). Within each group, the percentages of eight error propagation cases are aggregated, shown in Figure 1.c. We find that Figure 1.a and Figure 1.c are quite similar. We perform the same analysis for FT and have the same observation, shown in Figure 2.a and c.

To quantify the similarity of Figure 1.a and Figure 1.c, we use a metric, the cosine similarity [31]. This metric is a measure of similarity between two vectors. The cosine similarity of two vectors is bounded to [0,1], with 1 for strong correlation and 0 for otherwise. We represent eight bars shown in Figure 1.a as a vector with eight elements, and represent eight bars shown in Figure 1.c as the other vector with eight elements. We calculate the cosine similarity of the two vectors. The cosine similarity value is 0.999, which is very close to 1.

We perform the same analysis for all benchmarks and calculate the cosine similarity values. Due to the space limitation, we cannot show the profiling results for all benchmarks, but we summarize the cosine similarity results in Table 2. The table shows that except CG and LU (particularly, in the case of comparing four MPI processes and 64 MPI processes), all cases have the cosine similarity value close to 1. For the special cases of CG and LU (comparing four MPI processes and 64 MPI processes), the cosine similarity value

Table 2: Cosine similarity values to quantify the similarity of error propagation across MPI processes between small scale and large scale executions. "4V64" means using four MPI processes as the small scale execution and 64 MPI processes as the large scale execution; "8V64" means using eight MPI processes as the small scale execution and 64 MPI processes as the large scale executions.

Benchmarks	Cosine similarity value
CG (Class S, 4V64)	0.122
CG (Class S, 8V64)	0.999
FT (Class S, 4V64)	0.905
FT (Class S, 8V64)	0.999
MG (Class S, 4V64)	0.999
MG (Class S, 8V64)	1.000
LU (Class W, 4V64)	0.638
LU (Class W, 8V64)	1.000
MiniFE (nx=30 ny=30 nz=30, 4V64)	0.981
MiniFE (nx=30 ny=30 nz=30, 8V64)	1.000
PENNANT (leblanc, 4V64)	0.979
PENNANT (leblanc, 8V64)	0.999

is not close to 1, because the execution of four MPI processes (the small-scale execution) has error propagation across MPI processes in almost every fault injection test, while the execution of 64 MPI processes (the large-scale execution) has many fault injection tests without error propagation (i.e., the error is limited to only one MPI process).

Observation 3: *The error propagation across MPI processes in a small scale execution can often serve as a strong indication to the error propagation across MPI processes in a large-scale execution.*

3.3 Characterizing Difference in Application Resilience

In parallel execution, once an error is injected into one MPI process, the error may propagate into multiple MPI processes. Afterward, multiple MPI processes are contaminated. We collect the fault injection results for cases of n MPI processes being contaminated ($1 \leq n \leq p$ and p is the number of MPI processes in total). In addition, we collect the fault injection results for cases of injecting n errors into serial execution, and the n errors are injected into the common computation. We want to compare fault injection results between the above parallel and serial executions. Such comparison motivates the construction of our models.

Figure 3 shows the fault injection results for "success" outcome. Note that in the figure, some fault injection results for the parallel execution (e.g., the cases of 2-6 errors in LU) are missing, because we do not observe the injected error contaminates those numbers of MPI processes. Figure 3 reveals the following observation.

Observation 4: *In some of the cases (e.g., CG, MiniFE, and PENNANT), the fault injection result for serial execution with multiple errors injected is quite similar to the one for parallel execution with multiple MPI processes contaminated. In some cases (e.g., MG), the fault injection results of serial execution and parallel execution are different, but the variances of success rate are quite similar. In the rest of cases (e.g., FT and LU), serial and parallel executions are different in both fault injection results and the variance of success rate.*

The reasons that account for the difference in fault injection results between serial and parallel executions are two-fold. First, fault injection in serial execution is unable to capture *at what time* error propagates across MPI processes. Error propagation across MPI processes only happens after certain MPI communication phases. However, fault injection in serial execution injects errors completely randomly. Second, when injecting multiple errors into serial execution, those errors may be injected into the computation that only happens in one MPI process of parallel execution, while we expect those errors to be injected into the computation that happens in multiple processes of parallel execution. The above-unexpected error occurrence in serial execution is different from the execution where multiple MPI processes have errors in parallel.

4 RESILIENCE MODELING

Our models are based on the observations in Section 3. We explain our modeling in this section. We first give a general description of the models and then explain it in details.

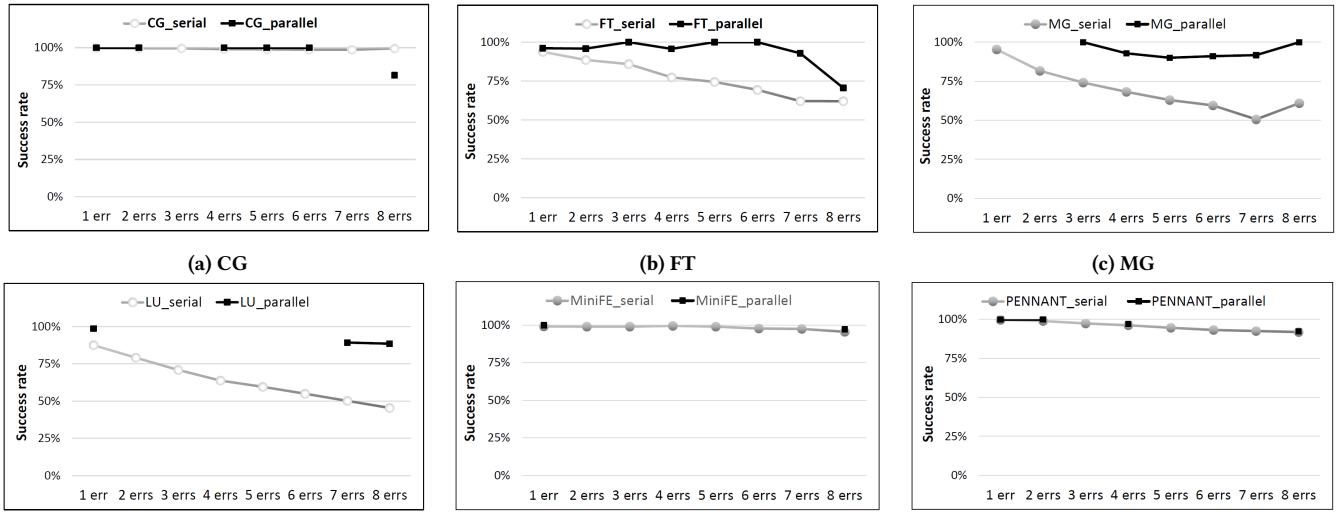
4.1 General Description

Our models aim at predicting fault injection results for large-scale parallel execution, using fault injection results for serial execution (based on Observation 4) and using limited profiling of smaller scale parallel execution (based on Observation 3). In essence, our models empirically quantify the correlation between the fault injection results of small-scale execution (including serial execution) and large-scale execution. We generally describe our models as follows.

Given an error that happens in an MPI process in parallel execution, the error can happen in the common computation and propagate to other MPI processes (or no propagation at all). If the error propagates to other n MPI processes, then $n+1$ MPI processes will have errors concurrently propagating. We model such error propagation behavior with a serial execution with $n+1$ errors injected into the common computation. Those $n+1$ errors occur in the serial execution, emulating the scenario where $n+1$ MPI processes are contaminated in the parallel execution.

To further explain the idea, we use an example shown in Figure 4. In the figure, we have a parallel execution of four MPI processes and a serial execution. During the parallel execution, an error occurs in one MPI process and propagates to other two processes through MPI communication and synchronization. Afterward, three MPI processes have errors. To emulate the impact of three contaminated MPI processes on the application in the parallel execution, we inject three errors in the serial execution. The three errors co-exist and propagate.

The above emulation approach is based on Observation 4. (i.e., using the serial execution with errors injected can give an indication of application resilience of a parallel execution). However, Observation 4 also reveals that sometimes the serial execution cannot provide good emulation. To decide that if the serial execution can provide good emulation or not, we perform fault injection in a small scale execution, and compare its fault injection result with the fault injection result of the serial execution. If the two results are quite different, then the serial execution cannot provide good emulation. For such case, we fine-tune the fault injection result of



(a) CG (b) FT (c) MG

(d) LU (e) MiniFE (f) PENNANT

Figure 3: Characterization of application resilience difference between serial and parallel executions (eight MPI processes). The x axis is the number of errors injected into the serial execution or the number of contaminated MPI processes in the parallel execution. The y axis is the fault injection result for success outcome (i.e., success rate).

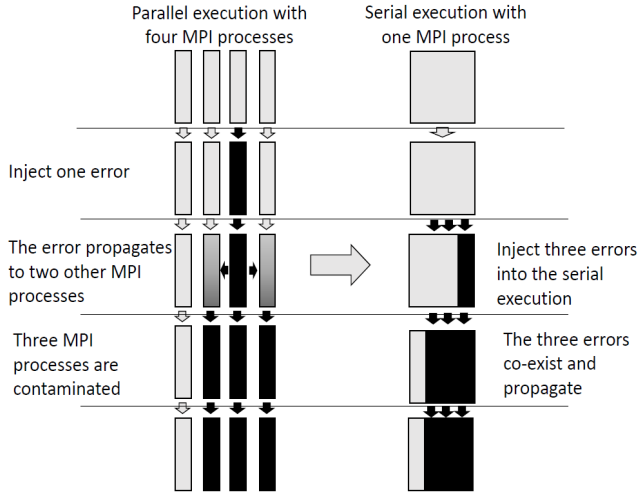


Figure 4: An example to explain the basic idea of using serial execution to emulate error effects in parallel execution.

the serial execution, using the fault injection result in the small-scale execution, to approximate application resilience in large-scale execution.

Besides the above discussion on the error occurrence in the common computation, it is possible that an error is injected into the parallel-unique computation. However, Observation 2 reveals that the parallel-unique computation is often small. Hence the chance to inject an error into it is small, and its contribution to the fault injection result is small. In the case that the parallel-unique computation is relatively large (e.g., FT), we model the impact of the parallel-unique computation on the application in a large scale execution by using the fault injection result for a small scale execution.

To use the models, we must know how many other MPI processes are contaminated after an error is injected into one MPI process, during parallel execution. We estimate the probability of error propagation across MPI processes using a small scale parallel execution. Such estimation is based on Observation 3. In particular, we perform fault inject tests in a small scale execution, and profile how often the error propagates across MPI processes. We call such profiling approach the *sampling-based approach*, because we use the error propagation probability collected in a small scale as samples to project the error propagation probability in a large scale.

To use the models, we must also perform fault injection in the serial execution to generate p fault injection results (p is the number of MPI processes in the large-scale execution for which we want to make the prediction). Each fault injection result corresponds to fault injection tests with x errors injected ($1 \leq x \leq p$). To avoid extensive fault injection, we only collect a sample set of fault injection results in the serial execution and use the sampling fault injection results to project the other fault injection results. Such sampling-based approach makes our modeling method more easily to be deployed, especially when p is large.

4.2 Model Details

We explain the models in details in this section. The model notation is summarized in Table 3.

Model formulation. Given a parallel execution, an error injected into one MPI process can happen in the common computation or parallel-unique computation. The fault injection results in a parallel execution is a weighted sum of the fault injection results in the common computation and parallel-unique computation, shown in Equation 1. We use the notation, FI_{par_common} and FI_{par_unique}, to represent the fault injection results for the common computation and parallel-unique computation respectively. The notation, $prob_1$ and $prob_2$, represents the probability of error injected into the common computation and parallel-unique computation, respectively. They are the weights of the weighted sum in the equation. $prob_1$ and $prob_2$ are calculated based on the execution time spent in the

common computation and parallel-unique code. For example, if a large scale parallel execution spends 90% of its execution time in the common computation and 10% of its execution time in the parallel-unique computation, then $prob_1$ and $prob_2$ are 90% and 10% respectively. The execution time of a large scale parallel execution can be predicted based on the execution time of a smaller scale execution [9]. Predicting the execution time of parallel execution is out of the scope of this paper.

$$FI_{par} = prob_1 \times FI_{par\_common} + prob_2 \times FI_{par\_unique} \quad (1)$$

To calculate FI_{par_common}, we use the fault injection results for serial execution. The general idea is discussed in Section 4.1 and shown in Figure 4. Equation 2 formalizes the idea.

$$FI_{par\_common} = \frac{n_1 \times FI_{ser_1} + n_2 \times FI_{ser_2} + \cdots + n_p \times FI_{ser_p}}{n_1 + n_2 + \cdots + n_p} \quad (2)$$

In Equation 2, the denominator $(n_1 + n_2 + \cdots + n_p)$ is the total number of fault injection tests to calculate FI_{par_common}. Among those tests, n_1 of them have one MPI process contaminated; or more generally speaking, n_x $(1 \le x \le p$, where p is the number of MPI processes) of them have x MPI processes contaminated. F_{ser_1} is the fault injection result for serial execution with one error injected; or more generally speaking, F_{ser_x} $(1 \le x \le p)$ is the fault injection result for serial execution with x errors injected. Hence, FI_{par_common} is a weighted sum of the fault injection results for serial execution with a different number of errors injected. The weight for serial execution with x errors injected is $n_x/(n_1 + n_2 + \cdots + n_p)$. We use notation r_x to represent such weight, shown in Equation 3. r_x is, in fact, the probability of x MPI processes contaminated after an error is injected into one MPI process. Based on r_x, Equation 2 is transformed into Equation 4.

$$r_x = \frac{n_x}{n_1 + n_2 + \cdots + n_p} \qquad (1 \le x \le p) \quad (3)$$

$$FI_{par\_common} = r_1 \times FI_{ser_1} + r_2 \times FI_{ser_2} + \cdots + r_p \times FI_{ser_p} \quad (4)$$

Equation 4 is constructed based on Observation 4 (i.e., serial execution with x errors injected can emulate parallel execution with x MPI processes contaminated). However, Observation 4 also reveals that there are some cases (e.g., LU and MG) where serial execution cannot provide good emulation for parallel execution. To identify those cases, we perform fault injection tests in a small scale (e.g., four MPI processes). We compare the fault injection result of the small scale execution with the one for serial execution. If the two results are quite different (larger than 20% difference), then we claim that serial execution with fault injection cannot provide good emulation to study application resilience in parallel execution.

To handle the case that serial execution cannot provide good emulation, we introduce a set of parameters α_x $(1 \le x \le p)$. Those parameters are used to fine-tune FI_{ser_x}, such that FI_{ser_x} can better approximate fault injection results in parallel execution. In particular, $FI'_{ser_x} = FI_{ser_x} \times \alpha_x$, where FI'_{ser_x} is the fault injection result in serial execution after fine-tuning. α_x are obtained from the fault injection in a small scale execution. For any x $(1 \le x \le S$, S is the number of MPI processes in the small scale execution), $\alpha_x = FI_{small\_par_x}/FI_{ser_x}$, where $FI_{small\_par_x}$ is the fault injection result for the small scale execution with x MPI processes contaminated. For any x $(x > S)$, $\alpha_x = \alpha_S$.

Table 3: Model notation

Variable	Definition
FI_{par}	Fault injection result for a parallel execution
p	The number of MPI processes in large scale parallel execution
S	The number of MPI processes in small scale parallel execution
FI_{ser_x} $(1 \le x \le p)$	Fault injection result for a serial execution with x errors injected
n_x $(1 \le x \le p)$	The number of fault injection tests with x MPI processes contaminated
$n_1 + n_2 + \cdots + n_p$	The total number of fault injection tests for large scale parallel execution
r_x $(1 \le x \le p)$	The probability of x MPI processes contaminated after an error is injected into one MPI process.

For FI_{par_unique}, we use our Observation 2 in Section 3. The observation suggests that in most cases, the parallel-unique computation is a small portion of total execution time. Hence, $prob_2$ in Equation 1 is small and FI_{par_unique} can be ignored. For the cases when the parallel-unique computation takes a relatively large portion of total execution time, we calculate FI_{par_unique} using the fault injection result in a small scale parallel execution. In particular, we perform fault injection tests in a small scale parallel execution, and each test has one error injected into the parallel-unique computation in one MPI process. We use the fault inject result as FI_{par_unique}.

Model usage. To use the models, we need to calculate FI_{ser_x} and r_x $(1 \le x \le p)$. When p is very large, we have to perform fault injection tests in serial execution for each case of x to get FI_{ser_x}, which is time-consuming. We introduce a sampling-based approach to approximate FI_{ser_x} and r_x using a few sample cases of x.

In particular, to calculate FI_{ser_x} $(1 \le x \le p)$, we do not perform fault injection tests for every case of x. Instead, we choose a few sample cases and use the fault injection results of those sample cases for the other cases. Suppose we use S sample cases. Those S cases are selected to evenly sample the whole space of x. The sample cases will be $x_1 = 1, x_2 = 2p/S, x_3 = 3p/S, \cdots, x_S = p$. Given any x, FI_{ser_x} is equal to the fault injection result of one of the sample cases, $x_{\lceil x/S \rceil}$.

To calculate r_x $(1 \le x \le p)$, we use a small scale parallel execution with S MPI processes based on Observation 3. Observation 3 suggests that using small-scale parallel execution can capture the trend of error propagation across MPI processes in large-scale parallel execution. Hence, we perform fault injection tests in a small scale, each of which has one error injected. With those fault injection tests, we calculate r_x $(1 \le x \le S)$ in the small scale. We use the notation $r'_{x'}$ $(1 \le x' \le S)$ in the small scale to distinguish it with r_x $(1 \le x \le p)$ in the large scale. We map r_x to $r'_{x'}$, simply based on the following equation.

$$r_x = r'_{x'}, \qquad \text{where } x' = \lceil x/S \rceil \quad (5)$$

An example to use the models. To further explain how the models work, we use an example. Suppose that we want to predict the fault injection result (e.g., success rate) for a parallel execution of 64 MPI processes (i.e., $p = 64$). We use a small scale execution (four MPI processes) and serial execution to make the prediction.

$$FI_{par\_common} = r_1 \times FI_{ser_1} + \cdots + r_{64} \times FI_{ser_{64}} \quad (6)$$

To approximate FI_{ser_x} ($1 \le x \le 64$), we use a sample size $S = 4$. In other words, we only measure $FI_{ser_1}, FI_{ser_{32}}, FI_{ser_{48}}, FI_{ser_{64}}$. Other cases are approximated with the four cases. For example, $FI_{ser_2}, FI_{ser_3}, \cdots, FI_{ser_{16}}$ are approximated with FI_{ser_1}, and $FI_{ser_{17}}, FI_{ser_{18}}, \cdots, FI_{ser_{31}}$ are approximated with $FI_{ser_{32}}$. Based on the above approximation, Equation 6 becomes the following Equation 7.

$$\begin{aligned} FI_{par\_common} = & \\ & FI_{ser_1} \times (r_1 + r_2 + \cdots + r_{16}) + \\ & FI_{ser_{32}} \times (r_{17} + r_{18} + \cdots + r_{32}) + \\ & FI_{ser_{48}} \times (r_{33} + r_{33} + \cdots + r_{48}) + \\ & FI_{ser_{64}} \times (r_{49} + r_{50} + \cdots + r_{64}) \end{aligned} \quad (7)$$

Note that if serial execution with fault injection cannot provide good emulation for application resilience in parallel execution, FI_{ser_x} ($x = 1, 32, 48$, and 64) should be fine-tuned based on the fault injection results in the small scale execution. FI_{ser_x} in Equation 7 should be replaced with the fine-tuning result FI'_{ser_x}, where $FI'_{ser_1} = FI_{small\_par1}, FI'_{ser_{32}} = FI_{small\_par2}, FI'_{ser_{48}} = FI_{small\_par3}, FI'_{ser_{64}} = FI_{small\_par4}$

To approximate r_x ($1 \le x \le 64$), we use the small-scale execution with four MPI processes (i.e., $S = 4$). In particular, we perform a sufficiently large number of fault injection tests with the small-scale execution. Each test has one error injected into one MPI process. We calculate among those tests, what are the percentages of fault injection tests that have one, two, three and four MPI processes contaminated. Those percentage numbers are r'_1, r'_2, r'_3, and r'_4. Using them, Equation 7 becomes the following Equation 8. We use Equation 8 to make the prediction.

$$\begin{aligned} FI_{par\_common} = & \\ & FI_{ser_1} \times r'_1 + FI_{ser_{16}} \times r'_2 + \\ & FI_{ser_{32}} \times r'_3 + FI_{ser_{64}} \times r'_4 \end{aligned} \quad (8)$$

5 EVALUATION

We evaluate our modeling accuracy in this section. Besides this section, Section 3 also has some evaluation results to characterize application execution in small and large scales.

5.1 Experiment Setup

We use four benchmarks (CG, FT, MG and LU) from NAS parallel benchmark suite (the version 3.3) [3] and two proxy scientific applications, MiniFE [17] and PENNANT [18]. For CG, FT and MG, we use CLASS S as input problem; for LU, we use CLASS W. For MiniFE, we use the default input (nx=30, ny=20, nz=30). For PENNANT, we use the leblanc input problem. We run the evaluation on Chameleon [23], which is an NSF funded HPC and Cloud testbed with more than 492 nodes.

As discussed in Section 2, we choose floating point instructions for fault injection. For each fault injection deployment, we perform 4000 fault injection tests. In each fault injection deployment, the success rate becomes stable after the first 1000 tests. Hence our fault injection results have sufficient statistical significance.

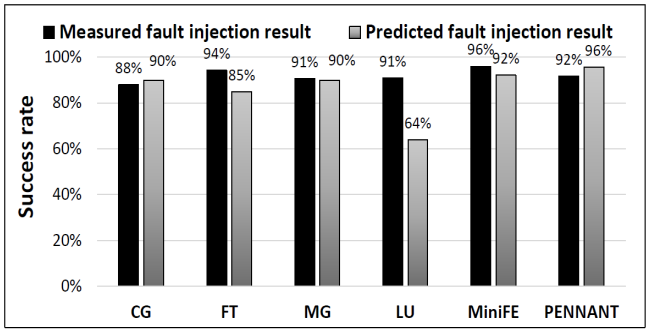


Figure 5: Modeling accuracy. We use a small scale execution (four MPI processes) and serial execution to predict a large scale execution (64 MPI processes).

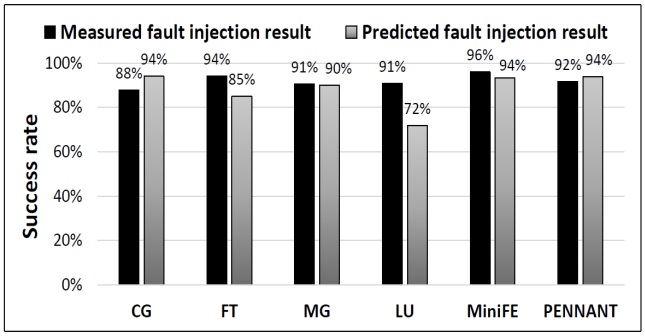


Figure 6: Modeling accuracy. We use a small scale execution (eight MPI processes) and serial execution to predict a large scale execution (64 MPI processes).

5.2 Evaluation Results

Prediction for 64 MPI processes. We first use serial execution (a single MPI process) and small-scale executions (4 MPI processes and 8 MPI processes) to predict the fault injection result of a large-scale execution (64 MPI processes). Figures 5 shows the modeling accuracy for using serial execution and four MPI processes to make the prediction; Figure 6 shows the modeling accuracy for using serial execution and eight MPI processes to make the prediction.

Using four MPI processes to make the prediction, the average success prediction error is 8% (27% at most). Using eight MPI processes to make the prediction, the average success prediction error is 7% (19% at most). We find that using more samples (i.e., using eight MPI processes) provide better modeling accuracy.

128 MPI processes prediction. To study how our modeling approach performs in a larger scale environment, we use the same small-scale executions and serial execution results as the input to predict the fault injection result of a larger scale execution (128 MPI processes). We cannot use any larger scale to verify our modeling accuracy, because a larger scale execution takes too much execution time for fault injection. We cannot get enough machine nodes to study application resilience in larger scales for such long time.

Figure 7 shows the results for CG and FT. Using serial execution and four MPI processes to make the prediction, the prediction error is no larger than 7%; Using serial execution and eight MPI processes, the prediction error is on larger than 6%.

Overall, our modeling method achieves high prediction accuracy.

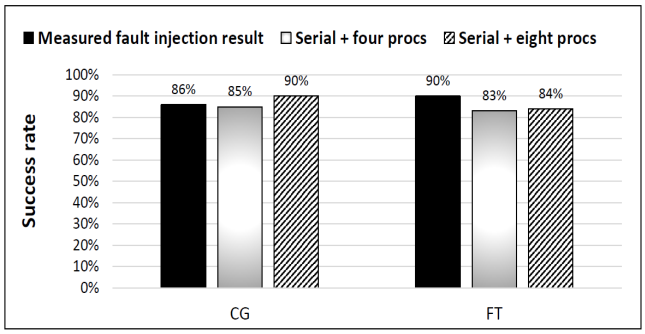


Figure 7: Modeling accuracy for a large scale execution (128 MPI processes). "Serial+four procs" means we use a small scale execution (four MPI processes) and serial execution to make the prediction; "Serial+eight procs" means we use a small scale execution (eight MPI processes) and serial execution to make the prediction.

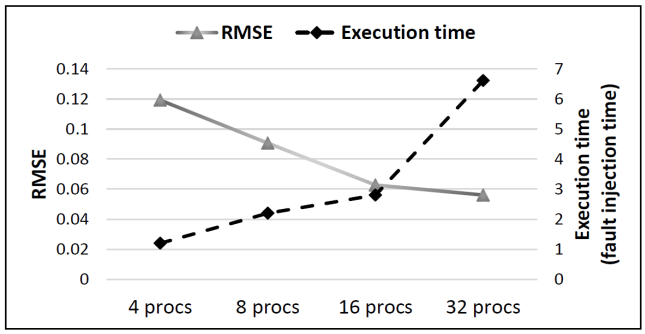


Figure 8: Study the tradeoff between modeling accuracy and execution time (fault injection time).

Sensitivity study. Generally speaking, if the scale of small-scale execution becomes larger, we can have better prediction accuracy, but have longer execution time for fault injection. To study the tradeoff between modeling accuracy and execution time (fault injection time), we perform the following study.

We change the scale of small-scale execution from 4, 8, 16 MPI processes to 32 MPI processes. We use those small-scale executions and serial execution to make the prediction for 64 MPI processes. We use all benchmarks for study and measure modeling accuracy and execution time. Figure 8 show the results.

The figure shows the root mean square error [8] of the prediction results of all benchmarks. The root mean square error is defined in Equation 9, where n is the number of benchmarks. The figure also shows the execution time (fault injection time) of small-scale execution. For each small-scale execution, the figure shows the average execution time of all benchmarks. The execution time is normalized by that of serial execution.

$$RMSE = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}\Big(measured\_rate - predicted\_rate\Big)^2} \quad (9)$$

The figure reveals that the accuracy is improved as we increase the scale of small-scale execution, but the execution time increases as well. When the scale of small-scale execution is 16 MPI processes, we reach a balance between execution time and modeling accuracy.

6 RELATED WORK

There exist some research efforts concentrating on developing and applying various kinds of techniques to study application resilience.

Application level random fault injection. Fault injection is a common method to evaluate application resilience. Casa et al. [7] study the resilience of an algebraic multi-grid solver by injecting faults into instructions' output based on LLVM. Similar work can be found in [6, 30]. Cher et al. [10] employ a GDB-like debugging tool to corrupt register states. Li et al. [21] build a binary instrumentation-based fault injection tool for random fault injection. Shantharam et al. [29] manually change the values of data objects to study the resilience of iterative methods. GOOFI [1] proposed by Aidemark et al is a pre-runtime software implemented fault injector, which injects faults into program and data area of the application. Ashraf et al. [2] and Wei et al. [36] use LLVM-based tools to inject faults, but they further introduce the functionality of tracking fault propagation. GemFI [24] proposed by Parasyris et al, is built on the full system simulators Gem5 [5]. GemFI can emulate the faults in registers within a processor. Levy et al. [20] and Wanner et al [35] leverage virtualization technology to emulate soft errors using QEMU, which are similar to our FSEFI fault injector used for the fault injection experiments. The existing work may face challenges to study application resilience in large-scale parallel execution.

Static and dynamic program analysis. Existing research uses compiler static and/or dynamic instruction analysis to detect application vulnerabilities. For example, Pattabiraman et al. use static analysis [26] and a data-dependence analysis [25] to determine the critical variables that are likely to propagate errors. Previous work also leverages the idea that some faults produce similar manifestations, thus they can be categorized into the same class in application resilience studies. For example, Hari et al. [15] use static and dynamic analysis to choose representative instructions for fault injection. They further leverage the equivalence of intermediate execution states to reduce the number of fault injections [28], and quantify the impact of single-bit errors in all dynamic instructions of an execution [34]. The above research efforts can be used in our work to reduce the number of fault injection tests, but those research efforts cannot fundamentally address the challenges to study application resilience in large-scale parallel execution.

Resilience modeling. There are several existing studies investigating resilience modeling. Most of them [11, 16, 22, 27] focus on the system-level fault behavior modeling while we build the model on the application-level. Li et al. [38] introduce a new resilience metric, the data vulnerability factor (DVF), to analyze application vulnerabilities. However, such method requires knowledge of both the application and target hardware into the calculation. In this paper, we propose a sampling-based approach by analyzing application fault injection results in small-scale execution and serial execution to model and predict fault injection results for the application running in large scales. This approach is lightweight and does not need any algorithm or hardware knowledge.

7 CONCLUSIONS

Studying application resilience in large scales can be very challenging because of its high requirement on hardware resource. In this

paper, we introduce a new methodology to study the resilience of applications in large-scale parallel execution. Our methodology makes such study feasible and manageable. We reveal that application resilience in different execution modes (small scale and large scale parallel executions) is correlated. Based on such correlation, we introduce empirical models that use fault injection results from small-scale and execution and serial execution to model and predict application resilience in large-scale execution. Our work provides potential opportunities to study application resilience in the future extreme scale, hence helps future HPC applications survive ever increasing threats from the HPC resilience challenges.

Acknowledgement This work is partially supported by U.S. National Science Foundation (CNS-1617967 and CCF-1553645) and Kent State University (201065). We thank anonymous reviewers for their valuable feedback.

REFERENCES

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. 2001. GOOFI: Generic Object-oriented Fault Injection Tool. In *International Conference on Dependable Systems and Networks*.

[2] Rizwan Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the Propagation of Transient Errors in HPC Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[3] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. 1992. NAS Parallel Benchmark Results. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[4] Eli Bendersky. 2012. PYELFTOOLS. https://github.com/eliben/pyelftools. (2012).

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.

[6] Jon Calhoun, Luke Olson, and Marc Snir. 2014. FlipIt: An LLVM Based Fault Injector for HPC. In *Revised Selected Papers, Part I, of the Euro-Par 2014 International Workshops on Parallel Processing - Volume 8805*.

[7] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. 2012. Fault Resilience of the Multi-grid Solver. In *International Conference on Supercomputing (ICS)*.

[8] T. Chai and R. R. Draxler. 2014. Root Mean Square Error (RMSE) or Mean Absolute Error (MAE)? - Arguments Against Avoiding RMSE in the Literature. *Geoscientific Model Development* 7, 3 (2014), 1247–1250.

[9] G. Chapuis, D. Nicholaeff, S. Eidenbenz, and R. S. Pavel. 2016. Predicting Performance of Smoothed Particle Hydrodynamics Codes at Large Scales. In *2016 Winter Simulation Conference (WSC)*.

[10] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller. 2014. Understanding Soft Error Resiliency of BlueGene/Q Compute Chip Through Hardware Proton Irradiation and Software Fault Injection. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[11] A. Gainaru, F. Cappello, and W. Kramer. 2012. Taming of the Shrew: Modeling the Normal and Faulty Behaviour of Large-scale HPC Systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*.

[12] Qiang Guan, Nathan BeBardeleben, Panruo Wu, Stephan Eidenbenz, Sean Blanchard, Laura Monroe, Elisabeth Baseman, and Li Tan. 2016. Design, Use and Evaluation of P-FSEFI: A Parallel Soft Error Fault Injection Framework for Emulating Soft Errors in Parallel Applications. In *the 9th EAI International Conference on Simulation Tools and Techniques*.

[13] Qiang Guan, Nathan Debardeleben, Sean Blanchard, and Song Fu. 2014. F-sefi: A Fine-grained Soft Error Fault Injection Tool for Profiling Application Vulnerability. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 1245–1254.

[14] Luanzheng Guo, Jing Liang, and Dong Li. 2016. Understanding Ineffectiveness of Application-Level Fault Injection. In *Poster in International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[15] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[16] Eric Heien, Derrick Kondo, Ana Gainaru, Dan LaPine, Bill Kramer, and Franck Cappello. 2011. Modeling and Tolerating Heterogeneous Failures in Large Parallel Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[17] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. Improving Performance via Mini-applications. In *SANDIA REPORT*.

[18] Charles R. Ferenbaugh (Los Alamos National Laboratory). 2012. The PENNANT Mini-App. https://github.com/lanl/PENNANT. (2012).

[19] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. 2009. Statistical Fault Injection: Quantified Error and Confidence. In *Conference on Design, Automation and Test in Europe (DATE)*.

[20] Scott Levy, Matthew G.F. Dosanjh, Patrick G. Bridges, and Kurt B. Ferreira. 2013. Using Unreliable Virtual Hardware to Inject Errors in Extreme-scale Systems. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*.

[21] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. 2012. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[22] Yinglung Liang, Yanyong Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. 2006. BlueGene/L Failure Analysis and Prediction Models. In *International Conference on Dependable Systems and Networks (DSN'06)*.

[23] J. Mambretti, J. Chen, and F. Yeh. 2015. Next Generation Clouds, the Chameleon Cloud Testbed, and Software Defined Networking (SDN). In *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*.

[24] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas. 2014. GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[25] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2005. Application-Based Metrics for Strategic Placement of Detectors. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*.

[26] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. 2011. Automated Derivation of Application-Aware Error Detectors Using Static Analysis: The Trusted Illiac Approach. *IEEE Transactions on Dependable and Secure Computing* 8, 1 (Jan 2011), 44–57.

[27] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. 2003. Critical Event Prediction for Proactive Management in Large-scale Computer Clusters. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '03)*.

[28] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. 2014. GangES: Gang Error Simulation for Hardware Resiliency Evaluation. In *International Symposium on Computer Architecuture (ISCA)*.

[29] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. 2011. Characterizing the Impact of Soft Errors on Iterative Methods in Scientific Computing. In *International Conference on Supercomputing (ICS)*.

[30] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan. 2013. Towards Formal Approaches to System Resilience. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*.

[31] Amit Singhal. 2001. Modern Information Retrieval: A Brief Overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 24, 4 (2001), 35–43.

[32] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*.

[33] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. 2013. Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults. In *International Conference on High Performance Computing, Networking, Storage and Analysis*.

[34] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. 2016. Approxilyzer: Towards a Systematic Framework for Instruction-level Approximate Computing and Its Application to Hardware Resiliency. In *International Symposium on Microarchitecture (MICRO)*.

[35] Lucas Wanner, Salma Elmalaki, Liangzhen Lai, Puneet Gupta, and Mani Srivastava. 2013. VarEMU: An Emulation Testbed for Variability-aware Software. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '13)*.

[36] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. 2014. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *International Conference on Dependable Systems and Networks (DSN)*.

[37] Kai Wu, Qiang Guan, Nathan DeBardelebe, and Dong Li. 2017. Characterization and Comparison of Application Resilience for Serial and Parallel Executions. In *Poster in International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[38] Li Yu, Dong Li, Sparsh Mittal, and Jeffrey S. Vetter. 2014. Quantitatively Modeling Application Resiliency with the Data Vulnerability Factor. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.