Understanding the Propagation of Transient Errors in HPC Applications

Rizwan A. Ashraf
University of Central Florida
Orlando, FL
rizwan.ashraf@ucf.edu

Roberto Gioiosa
Pacific Northwest National Lab
Richland, WA
roberto.gioiosa@pnnl.gov

Gokcen Kestor
Pacific Northwest National Lab
Richland, WA
gokcen.kestor@pnnl.gov

Ronald F. DeMara
University of Central Florida
Orlando, FL
Ronald.Demara@ucf.edu

Chen-Yong Cher
IBM T.J. Watson Research Center
Yorktown Heights, NY
chenyong@us.ibm.com

Pradip Bose
IBM T.J. Watson Research Center
Yorktown Heights, NY
pbose@us.ibm.com

ABSTRACT

Resiliency of exascale systems has quickly become an important concern for the scientific community. Despite its importance, still much remains to be determined regarding how faults disseminate or at what rate do they impact HPC applications. The understanding of where and how fast faults propagate could lead to more efficient implementation of application-driven error detection and recovery.

In this work, we propose a fault propagation framework to analyze how faults propagate in MPI applications and to understand their vulnerability to faults. We employ a combination of compiler-level code transformation and instrumentation, along with a runtime checker. Using the information provided by our framework, we employ machine learning technique to derive application fault propagation models that can be used to estimate the number of corrupted memory locations at runtime.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: [Reliability, Testing, and Fault-Tolerance]; C.4 [**Computer Systems Organization**]: Performance of Systems—*Fault tolerance*

Keywords

Fault Injection, Fault Propagation, Distributed Applications, Soft Errors, Application Vulnerability, Resiliency

1. INTRODUCTION

Exascale systems promise to deliver 1000-fold more computing capability than current petascale supercomputers. New low-power and near-threshold voltage (NTV) technologies, along with higher temperature tolerance, may be employed in some systems to address the power challenges of exascale computing. Without additional mitigation actions, these factors, combined with the sheer number of components, will considerably increase the number of faults experienced during the execution of parallel applications, thereby reducing the mean time to failure (MTTF) and the productivity of these systems. Moreover, power constraints will limit the amount of energy and resources that can be dedicated to detect and correct errors [31], thus silent data corruptions (SDCs) are expected to become more common. Finally, transistor device aging effects will amplify the significance of these conditions, further reducing the return of investment of expensive exascale systems.

Despite the importance and the risks associated with SDCs, there are limited studies that analyze the impact of transient errors on high performance computing (HPC) applications and how these errors propagate in the application data structures (*application state*) [10, 14, 17, 23, 27]. Generally, these studies rely on statistical analysis of the application's outputs performed by injecting faults in the application's state at random points during the execution. Although these works provide important information on the vulnerability of parallel applications, this "black-box" approach does not provide insights on the impact of injected faults on the application's internal state. Thus, it becomes difficult to assess the extent to which a fault has contaminated the application's data structures, the speed at which it propagates, or which resilience mechanism to employ. In particular, the analysis of the application's output state does not distinguish cases in which the application's state is corrupted even if the final results are correct. This happens, for example, if the acceptable residual error of a scientific simulation is large enough to accommodate for the variance introduced by a transient error. The outcome of the same execution would be different with stricter error bounds.

We argue that without a comprehensive knowledge of how transient errors propagate in the application's state during its execution, the results of a fault injection analysis may be incomplete and inaccurate. This may lead to wrong resilient mechanism to be employed. In order to collect such comprehensive knowledge, we propose a new fault propagation framework that accurately tracks the propagation of faults in distributed MPI applications. Our framework provides internal application's vulnerability information beyond what is generally provided by statistical analysis based on output variation. Specifically, our fault propagation frame-

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807670

work provides information on the *speed* (i.e., how quickly a fault propagates into a process state) and *depth* (i.e., how many processes are affected) with which a fault propagates throughout the execution of the application. This information is essential to understand the impact of transient errors in HPC applications and, thus, to select the most appropriate resilience mechanism to employ. We believe that deeper insights provided by our framework can expose vulnerabilities in the application's algorithm and implementation that are oblivious to a "black-box" analysis.

Our framework consists of an LLVM-based instrumentation component and a runtime checker that tracks the propagation of faults into the application's state. Although conceptually straightforward, accurately tracking the propagation of a fault requires a comprehensive and thorough methodology along with properly-implemented tools. In fact, the general assumption that the output of an instruction becomes corrupted, i.e., a fault propagates, if at least one of the inputs is corrupted could lead to large overestimation of the number of corrupted memory locations. To avoid such overestimation and to precisely track faults in a generic operation, we replicate the stream of instructions to compute both the *potentially corrupted* outputs and the *pristine* outputs. The former are the outputs of the instructions that may use input values corrupted by an injected fault or contaminated by previous operands. The latter are the outputs of instructions that only use non-contaminated operands, which are not impacted by the error. At store operations we compare the potentially-corrupted and the pristine value to determine if a fault propagates to memory.

We use our new framework to analyze the impact of faults in several commonly-used parallel applications from different scientific domains, such as hydrodynamics and molecular dynamics, taken from various benchmark suites [1] and DOE proxy applications [3]. We perform our tests on a 32-node cluster with a total of 1,024 cores. First, we show that the outcomes of statistical analysis based on output variation may lead to erroneous conclusions. For example, we show that such "black-box" analysis would conclude that faults injected in *LULESH* are masked in over 90% of the cases. However, a deeper analysis reveals that the faults often propagate and may corrupt up to 25% of the application memory state. Second, we show that, given the iterative nature of most HPC applications, faults propagate linearly into the application's states. We then employ machine learning techniques to derive application fault propagation models that can be used to estimate the number of corrupted memory locations, once a fault is detected. These models can be used to estimate the number of corrupted memory locations and to understand if a roll-back to a previous checkpoint should be triggered. From the fault propagation models, we extract the fault propagation speed (FPS) factor, a measure that indicates how quickly a transient error propagates into the application's state. We argue that the FPS is an insightful metric to express the vulnerability of HPC applications and can be combined with architectural vulnerability metrics [32, 36] to assess the system resilience. Finally, we measure the extent to which transient errors propagate to MPI processes through message passing.

In summary, this paper makes the following contributions:

- We propose a novel fault propagation framework that accurately tracks faults within a process and across MPI processes. To the best of our knowledge, our work is the first to provide detailed information about fault propagation in distributed MPI applications.
- We show that, without a comprehensive fault propagation analysis, the conclusions driven from statistical output variation analysis may be inaccurate.
- We derive application fault propagation models and compute the fault propagation speed factors.

The rest of this paper is organized as follows: Section 2 describes the fault model. Section 3 describes our fault propagation framework. Section 4 shows our experimental results. Section 5 describes our fault propagation models. Section 6 analyzes previous work. Section 7 presents the conclusions.

2. FAULT MODEL

Large supercomputers are mainly built out of commodity off-the-shelf (COTS) components [2]. Although COTS components may be fairly reliable, for example the MTTF of a single memory module with double-bit error-correction capability is over 100 years [16], the sheer number of components assembled in current supercomputers dramatically reduces the net MTTF to a few days. For exascale systems, the expected MTTF is in the order of hours [22]. Faults occur at hardware level as the result of physical phenomena such as exposure to alpha particles, transient timing violations, or localized temperature variations. Generally, faults are categorized into two main categories: hard and soft faults. Hard faults are either permanent or intermittent; they are typically the result of aging effects or malfunctioning devices. Soft faults are transient faults typically caused by environmental conditions, such as radiation effects [13], that manifest in the form of bit flips. With the development of NTV technology [15] and the need for higher temperature tolerance required to achieve exascale efficiency [22], transient errors are becoming predominant. This work focuses on transient faults that escape hardware correction and detection and that propagate to the architectural state of the processor. We analyze the characteristics of how such faults propagate through the application's memory state. Transient errors may occur any time during the execution of an application and can result in a variety of outcomes. We extend previously proposed classification [5, 13, 40, 41] for HPC systems and classify the experiment outcomes in the following categories:

Vanished (V): Faults are masked at the processor-level and do not propagate to memory, thus the application produces correct outputs and the entire internal memory's state is correct.

Output Not Affected (ONA): Faults propagate to the application's memory state, but the final results of the computation are still within the acceptable error margins and the application terminates within the number of iterations executed in fault-free runs.

Wrong Output (WO): Faults propagate through the application's state and corrupt the final output. The application may take longer to terminate.

Prolonged execution (PEX): Some applications may be able to tolerate corrupted memory states and still produce correct results by performing extra work to

refine the current solution. These applications provide some form of inherent fault tolerance in their algorithms, though at the cost of delaying the output.

Crashed (C): Finally, faults can induce application crashes. We consider "hangs", i.e., the cases in which the application does not terminate, in this class.

Compared to previous classification, SDCs are identified as ONA, WO and, PEX, depending on their effects on the application's state and output. Analysis based on output variation, such as fault injection analysis, cannot distinguish V from ONA, thus we introduce the class Correct Output (CO) to indicate the sum of V and ONA when the application produces correct results within the expected execution time. Correct results with longer executions are in PEX.

Given that exascale machines do not yet exist and that analyzing real faults on current systems would require long periods of time, thus, in this paper, we follow a methodology compatible with previous work and perform accelerated fault injection [23, 42]. This approach allows us to perform a large number of tests in a relatively short time and explore a considerable part of the application code and result space. We randomly inject single-bit flips at the register-level during the execution of an application. The faults are injected into the source register of both arithmetic and load/store operations, which is the most accurate high-level error injection model [13], besides circuit-level fault injection. Since our target is to understand the vulnerability of HPC applications, we only inject faults in the application source code but not in the MPI and system libraries.

Our primary goal in this work is to analyze the vulnerability and sensitivity of HPC applications to transient errors. It must be remarked that this information alone will not provide a comprehensive understanding of the resilience of the entire system. Our analysis focuses on what happens after a fault, undetected by the hardware, contaminates the processor registers. We refer to previous work on understanding how transient errors occurring at circuit level eventually propagate to architectural level [13, 36]. In order to assess the resilience of the entire system (hardware and application), the user needs to combine the expected hardware failure in time (FIT) rate with the information provided in this work. Our work is complementary to the resiliency studies of hardware systems [4, 5, 24, 32, 36] and is essential to understand the resilience of the entire system. As we will explain in Section 6, our work is similar to the *program vulnerability factor (PVF)* [39] and the *data vulnerability factor (DVF)* [43], in that it examines the application's sensitivity to faults. However, our fault model considers both architectural level, i.e., how faults propagate in processor registers, not considered by DVF metric, and the MPI communication level, i.e., how faults propagate among MPI processes, not considered by PVF metric.

3. FAULT PROPAGATION FRAMEWORK

This section describes our fault propagation framework for MPI applications. The framework consists of the fault injection and the fault propagation modules.

3.1 Fault Injection

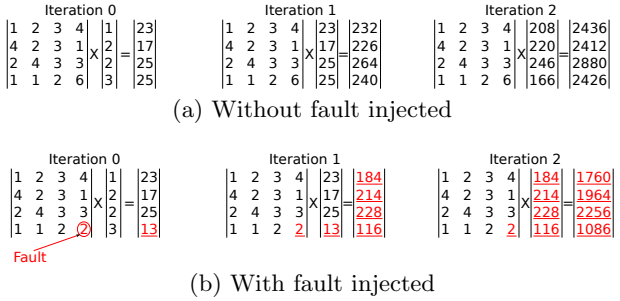
To understand the propagation of faults in parallel HPC applications we need to inject faults into the application's state. Since we target large parallel applications running on a cluster of nodes and fault injection at the circuit level is prohibitively slow and would limit the exploration space, in this work we opted for accelerated software fault injection [13]. We also opted for a compiler-level fault injection strategy because we assume that undetected transient errors will propagate to processor register or functional units but may be masked at processor level before contaminating memory locations. Software fault injection tools based on binary instrumentation, such as [23], instead, directly inject faults into memory locations. Previous work showed that this form of fault injection may inaccurately model transient errors occurring in the hardware [13, 31]. Moreover, injecting faults directly into the application memory state has an impact when assessing the resilience of a system. In fact, it is not possible to use the architectural FIT rate, usually known, but the user needs to estimate an application-specific FIT rate that takes into account faults masked at architectural level and that have not propagated to the application's memory state. Previous work [5] showed that the application-specific FIT rate is a dominating factor when assessing overall system resilience and it is not trivial to estimate. Injecting faults at register level, on the other hand, still allows us to use the micro-architectural FIT rate to evaluate the resilience of a system.

To inject faults at processor level we use LLFI [42], an LLVM-based fault injection tool that injects faults into the LLVM Intermediate Representation (IR) of the application source code. LLFI injects a single fault into live register at every run of a sequential application in specific program points, which allows the user to track the effects of the fault back to the source code. We extend LLFI in two directions: First, we add the necessary support to inject faults in multiple MPI processes at different times during the execution. Second, we extend LLFI to inject zero or more faults into each MPI process during each execution of the application. This means that only some MPI processes may experience direct (injected) faults, while others may experience indirect faults caused by receiving messages containing errors. In the rest of this work, we refer to LLFI++ as the extended version of LLFI for MPI applications. Using LLFI as our software fault injector has its drawbacks. First, faults are only injected into live registers, which limits our fault model to transient errors occurring during an instruction's operation, e.g., flip-flop errors in execution units. As shown in Section 4.3, this has an impact on the percentage of vanished faults. Second, as other software fault injectors, LLFI does not inject faults into non-programmable registers.

3.2 Fault Propagation

Once an error occurs in a hardware register, it might propagate and contaminate other registers or memory locations in the address space of same process or, in case of distributed parallel applications, in the address space of other processes. Understanding the *speed* (in terms of time) and *depth* (in terms of number of contaminated processes) at which a fault propagates is essential to understand the vulnerability of an application to faults.

Figure 1 shows an example of how an injected fault may propagate and contaminate a large part of the memory state of an application. The example in Figure 1 is an iterative Matrix-Vector multiplication program that, at each iteration, performs $Ax_i = b_i$, where A is a constant input matrix,



(a) Without fault injected

(b) With fault injected

Figure 1: Fault propagation in Matrix-Vector multiplication.

Table 1: Assume a and b to be 8-bit values and initially $a = 19$ (00010011) and $b = 5$ (00000101) and that the second least significant bit of a flips from 1 to 0.

N	Op	Result (b)	Faulty Result (b')	Cont.?
1	$b = a + 5$	24	**22**	Yes
2	$b = 13$	13	13	No
3	$b = a >> 1$	9	**8**	Yes
4	$b = a >> 2$	4	4	No

$x_i = b_{i-1}$ is the input vector ($x_0 = [1 \quad 2 \quad 2 \quad 3]$ is a program input) and b_i is the iteration output. Figure 1a shows three iterations of the program when no fault is injected. Let's now assume that, during the execution of iteration 0, a fault occurs and the third least significant bit of $A[3,3]$ flips from 1 to 0, inducing a change of value in $A[3,3]$ from 6 to 2. The corrupted value in A is then used to compute $b_0[3]$ which, in turn, becomes corrupted. Since b_0 is used as input vector in iteration 1, the fault keeps propagating and corrupting other values in the application's state. As shown in Figure 1b, in three iterations one single bit flip contaminates 37.5% of the application's memory state, 100% of the application's output state b_2 and 100% of the read/write state x_2 and b_2. In fact, in just two iterations, 25% of the application's state, 100% of the output state and 62.5% of the read/write state have already been corrupted. This rudimentary example shows how quickly a fault can propagate and contaminate part of the application's state and outputs.

The fault propagation module (FPM) for MPI applications tracks the fault injected by LLFI++ into a register as it progressively contaminates the application's memory state. The FPM consists of two components: a compiler-level translation/instrumentation and a runtime checker/tracker. This double-approach is necessary because accurately tracking faults on real applications running on distributed systems is not as simple as it may appear at first. Many false positives may produce inaccurate results and lead to erroneous conclusions. Table 1 shows several examples in which the propagation of a fault depends on the particular operation and the operands involved. As Table 1 shows, whether or not the fault introduced in a propagates to b depends on the subsequent operations performed on both a and b. Understanding whether a fault propagates with a pure compiler-level tool is complicated, thus we integrate it with a runtime checker/tracker. Table 1 shows that a fault propagates only if the output of an operation diverges from its equivalent output when all inputs are pristine. We use this observation to understand, at runtime, how fault propagates into the application memory state. This means that we need to compute both the *potentially-corrupted* results

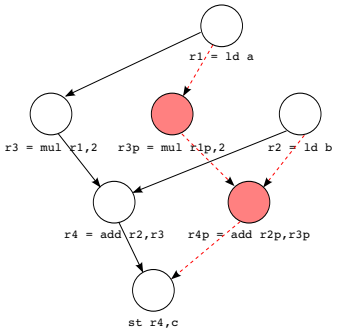


Figure 2: Primary and secondary chain of instruction for the statement `c = 2*a + b`.

of an operation (b in Table 1), computed with inputs that might have been contaminated, and the *pristine* results of the same operation (b' in Table 1), computed with inputs that have not been contaminated. The instructions that contain potentially-corrupted results are part of the *Primary Chain* of instructions, i.e., the original program instructions chain augmented with LLFI++ fault injection instrumentation, while the replicated original program instructions are part of the *Secondary Chain*, as shown in Figure 2. The pristine values associated with corrupted memory locations are stored in a hash-table structure in the FPM runtime.

The FPM translation and instrumentation of the statement `c = 2*a + b` is depicted in Figure 3. Figure 3a shows an LLVM-like intermediate representation of the code. The program loads two input values from addresses `a` and `b` and stores the final result at the address `c`. Figure 3b shows the first step (fault injection): the code is instrumented with fault injection functions (`fim_inj(x)` at lines 3, 5, and 6).[1] At runtime, the `fim_inj(x)` function checks if a fault should be injected and eventually flips a random bit in register `x` (hence the term "potentially-corrupted"). The result is a potentially-corrupted value stored in register `xf`, which will then be used in the primary chain of instructions. The second step (fault propagation) produces the code shown in Figure 3c. All arithmetic instructions are replicated by the source-to-source translator (lines 7 and 11). Hence, the original instructions in the primary chain use potentially-corrupted registers (`r1f`, `r2f`, `r3`, `r3f`, and `r4`), whereas, the replicated instructions in the secondary chain use pristine registers (`r1p`, `r2p`, `r3p`, and `r4p`). Load and store operations are instrumented with runtime functions as follows: at each load operation the runtime system checks whether the target memory location has been previously contaminated and, if so, also fetches the pristine value for that memory location (`fpm_fetch()` at lines 2 and 4 in Figure 3c and also the red dashed lines in Figure 2). If the target memory location has not been contaminated, the `fpm_fetch()` returns the same value for both the potentially-corrupted and the pristine registers. Store instructions are instrumented to compare the potentially-corrupted value that has to be stored in memory to the corresponding pristine value computed by the secondary chain of instruction (`fpm_store()` at line 13). If the two values differ, the runtime checkers adds the memory location address to the list of memory locations contaminated and stores its pristine value (this value will be

[1] In this example, LLFI++ instruments only arithmetic operations, but other class of instructions can be considered.

```
1: r1 = ld a
2: r2 = ld b
3: r3 = mul r1, 2
4: r4 = add r2, r3
5: st r4, c
```

(a) LLVM IR

```
1: r1 = ld a
2: r2 = ld b
3: r1f = fim_inj(r1)
4: r3 = mul r1f, 2
5: r2f = fim_inj(r2)
6: r3f = fim_inj(r3)
7: r4 = add r2f, r3f
8: st r4, c
```

(b) LLFI++ code

```
1: r1 = ld a
2: r1p = fpm_fetch(a)
3: r2 = ld b
4: r2p = fpm_fetch(b)
5: r1f = fim_inj(r1)
6: r3 = mul r1f, 2
7: r3p = mul r1p, 2
8: r2f = fim_inj(r2)
9: r3f = fim_inj(r3)
10: r4 = add r2f, r3f
11: r4p = add r2p, r3p
12: st r4, c
13: fpm_store(r4,r4p,c)
```

(c) FPM code

Figure 3: FPM transformation and instrumentation of the statement `c = 2*a + b`.

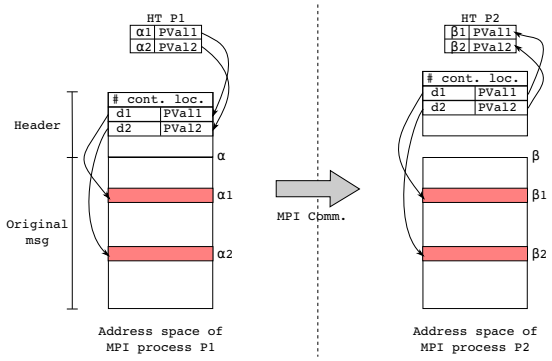


Figure 4: MPI message handling.

fetched from the next load instructions). Notice, we do not need to compare the potentially-corrupted and the pristine value produced by every single instruction but only when a value is stored to memory. In other words, we maintain a local representation of both potentially-corrupted and pristine registers until the final result is stored to memory.

In the following, we analyze the main challenges.

Store addresses: In the example in Figure 3, transient errors are only injected into registers that contains variables' values. However, instructions can also manipulate addresses and use registers to indirectly access memory. A fault propagating to a register that contains a memory address which is used by a store operation produces a duplicate effect. First, the actual memory location modified becomes corrupted because the address used by the store was not supposed to be written. Second, the memory location that was supposed to be written is not modified, hence contains a corrupted value. Consider the following instrumented code:

```
r1f = fim_inj(r1)
st 5, (r1f)
```

if a fault is injected in `r1` (thus, `r1 != r1f`), the value `5` is written to a memory location `r1f` that was not supposed to be written and becomes corrupted. To record this contamination, the FPM runtime adds the pair `<r1f,x>` to the runtime hash-table, where `x` is the original value of `r1f` before the store. The memory location `r1` was supposed to contain the value `5` after the store but it has not been overwritten, thus FPM also adds the pair `<r1,5>` to the hash-table.

Function Calls: The input parameters passed to a function may be contaminated by a previously injected fault and affect the result computed and returned by the function. For pure functions it would be enough to execute the function twice, once with the potentially-corrupted input parameters and once with the corresponding pristine values. The former case would produce a potentially-corrupted output while the latter a pristine output. We use this approach for library function calls (such as `sin()` from the math library). However, in general, a function may also access global variables during their execution. This means that a generic function could contaminate a much larger application state than the returned value. To address this issue, we follow the same dual-chain approach described previously, but we also modify the description of each function to accommodate one extra parameter (the pristine value) for each input parameter. Moreover, we modified the exit point to return a struct that consists of two values, the potentially-corrupted computed by the primary chain and the pristine value computed by the secondary chain of instructions. Finally, additional code is inserted to properly retrieve the pristine values associated to each input parameter and to store the pristine result.

Some functions, such as memory management or I/O operations, impact the address space structure or the interaction with the external world. We do not replicate them to avoid side effects, such as output values printed twice.

MPI communications: A fault can propagate from the address space of an MPI process `P1` to the address space of another MPI process `P2` if `P1` sends a message to `P2` containing contaminated data. Neither the sender nor the receiver process have enough information to accurately track the propagation of faults through inter-process communication. The main problem is that a contaminated memory location in the virtual address space of the sender may be stored in a completely different memory location in the virtual address space of the receiver. Since neither the sender nor the receiver has access to each other's address space, we embed extra information about the contaminated data in the message together with the message itself.

Figure 4 illustrates our approach in detail: Assume that an MPI process `P1` sends a message `msg` to a destination process `P2`. In the address space of the sender process `msg` is stored at address α, while in the address space of the destination process the message will be copied to address β, which, in general, is different from α. Also assume that N memory locations in `msg` are contaminated and that their pristine values are stored in the FPM hash-table in the address space of `P1`, `HT1`. Given that $\alpha \neq \beta$, we cannot use the addresses of the two contaminated memory locations in the address space of `P1` ($\alpha 1$ and $\alpha 2$) to derive the addresses of the memory locations in the address space of `P2` ($\beta 1$ and $\beta 2$). We use the *displacements* with respect to the beginning of the `msg`, which remains constant regardless of the initial address at which the `msg` is stored, to communicate to the receiver which memory locations are contaminated in the message. Before sending the message, the FPM runtime routine intercepts the MPI communication functions and analyzes the message. For each contaminated memory location, FPM computes the displacement with respect to the initial address α of the original message and retrieves the corresponding pristine values from the hash table. FPM then adds an extra `header` to the original `msg`, containing the number of memory locations contaminated in the message and one record `<displacement,pristine value>` for each

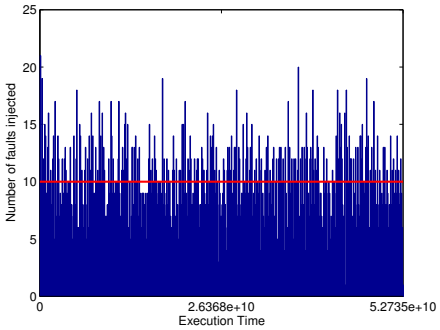


Figure 5: Fault injection coverage. Faults are injected uniformly throughout the execution of *LULESH*.

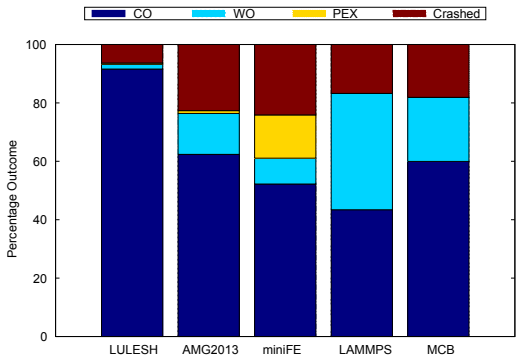


Figure 6: Outcome of fault injection with single fault into a single MPI process.

contaminated memory location. On the receiver side, the destination process extracts the `header` and uses the first entry to determine how many contaminated memory locations are present in the message. The receiver then extracts the original message `msg` and stores it at address β. At this point, the displacements in each record in the `header` are used to compute the addresses of the contaminated memory locations in the destination address space. Finally, the FPM runtime on the receiver side adds the addresses of the contaminated memory locations and their corresponding pristine values to the `HT2` in its address space.

4. EXPERIMENTAL RESULTS

In this section, we analyze the impact of faults in important HPC applications and how faults propagate into the applications' state. We performed our experiments on a 32-node cluster equipped with two AMD Interlagos [8] 16-core sockets, for a total of 32 cores per node and 1,024 cores per system. We selected several applications from different benchmark suites: *LULESH2* [20] from the ASCR Exascale Co-Design Center [3], *LAMMPS* [33], *AMG2013* and *MCB* from the CORAL program [1], and *miniFE* from the DOE proxy applications. All applications are compiled with `LLVM` version 3.4, which internally use `GNU gcc` 4.8.2, and `OpenMPI` 1.7.4. All applications use their default input set.

4.1 Fault Injection Coverage

First we analyze the coverage of our fault injection technique. Ideally, we would like to analyze how faults propagate in the application's memory state when injecting faults at every cycle. This approach, however, is impractical for large applications such as the ones tested in this work. Herein statistical fault injection is performed and therefore it is important to verify that we uniformly inject faults throughout the execution of the applications. Figure 5 shows that, indeed, we inject faults uniformly during the application execution. The Figure shows the results of 5,000 injections for *LULESH*:[2] the x-axis represents time in cycles divided into 500 bins. The bars represent the number of faults injected in each bin and the red line represents an ideal uniform distribution. As evident in the plot, the actual distribution of injected faults closely matches the ideal uniform distribution. We also verified the approximation of an ideal uniform distribution through χ^2 tests.

[2]The plots for the other applications follow the same structure and we omit them because of space constraints.

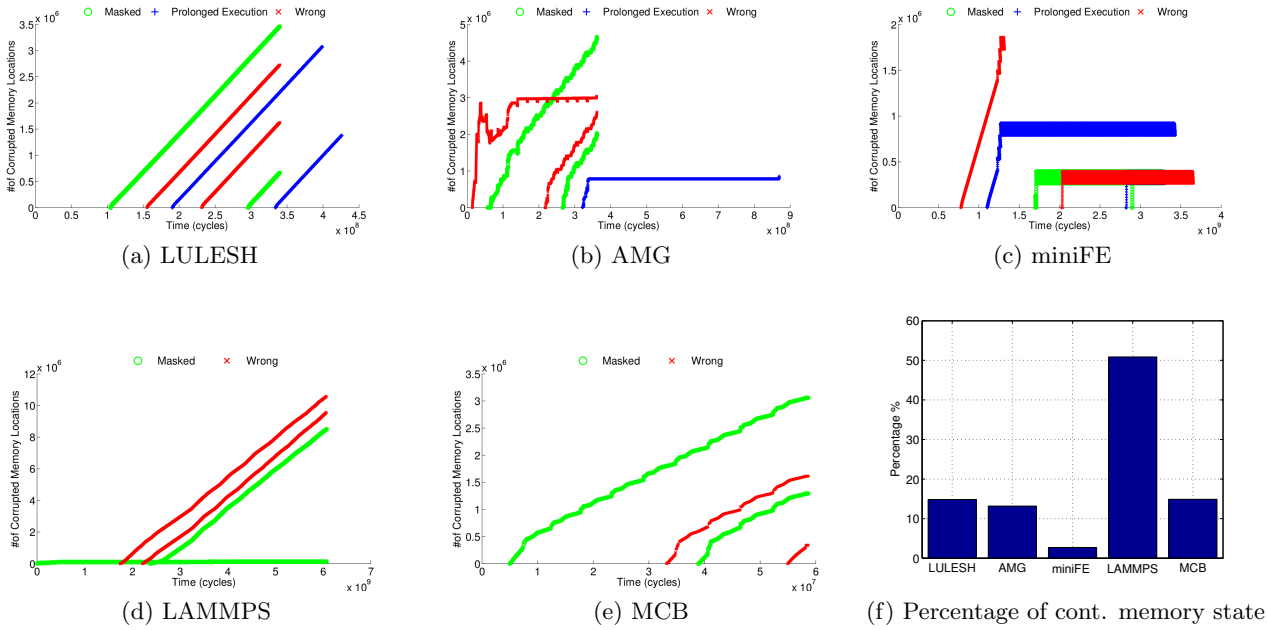
4.2 Fault Injection Analysis

In our second set of experiments, we analyze the impact of injecting random transient errors into registers. These experiments are similar to the "black box" approaches followed in previous work [23, 37, 42] and are based on the analysis of the application's output variation. We run each application 5,000 times and we inject a single fault in each execution into a randomly selected MPI process. In these experiments, we inject faults into registers utilized by arithmetic operations, but other kind of instructions can also be targeted by LLFI++. The application output is considered corrupted (WO) if it differs significantly from the fault-free execution (we use a 5% tolerance) or if the application itself reports results outside of the error boundaries.

Figure 6 reports the observed results. From the results, *LULESH* appears as a robust application with over 90% of cases resulting in correct results and no performance penalties. Only less than 10% of the executions result in crashes and less than 5% of the experiments produce wrong results. On the other extreme, *LAMMPS* appears to be the most vulnerable application: about 20% of the experiments result in crashes and in 40% of cases the result is corrupted by a single-bit fault (WO). Whereas, the final results are correct in only 40% of the experiments. *MCB* shows a behavior similar to *LAMMPS*, though 60% of experiments show correct results. For *miniFE*, we notice a considerable number of cases that produce a correct result, but take more time to converge to an acceptable solution (PEX). These are interesting cases, as they expose a particular characteristic of scientific applications that is not necessarily present in other domains: the user could trade-off the accuracy and correctness of the computed solution for performance. We believe that these kind of trade-offs will be more important in the exascale era, when SDCs will be more common. The crashes we observed are mainly due to bit flips in pointers that cause the applications to access a part of the address space that has not been allocated. For *LULESH*, we notice that some of the crashes occur because of an internal check on the partial result: if the energy computed at time step i is outside of the acceptable boundary, the application aborts the execution calling `MPI_Abort()` routine. This may explain why we observed a limited number of WO cases.

4.3 Fault Propagation Analysis

Statistical analysis of the vulnerability of parallel applications based on output variation provides useful high-level in-



(a) LULESH (b) AMG (c) miniFE

(d) LAMMPS (e) MCB (f) Percentage of cont. memory state

Figure 7: Fault propagation profiles. In these experiments we randomly inject a single fault per run into a randomly selected MPI process. Figure 7f shows the percentage of memory locations contaminated at the end of each applications (max).

formation but does not provide insights on the application's memory state. In this section, we examine how injected faults propagate and their characteristics. In particular, we are interested to discover how fast and with which profile faults propagate in the memory space of an MPI process (*propagation speed*) and how many MPI processes are contaminated (*propagation depth*). As in the previous set of experiments, we run each application 5,000 times and we inject a single fault per run. Since, it is not possible to show all the graphs, we selected representative fault propagation profiles for each application in Figure 7. We will show how to use the findings of these experiments in the next section. Already, the few cases plotted highlight the importance of the applications' structure and algorithm in the propagation of faults. As reported in the previous section, we notice that crashes generally occur immediately or in proximity of the injected fault, thus we do not report these cases in Figure 7. The plots in Figure 7 report two cases for each of the three remaining classes (CO, WO, and PEX), whenever possible. We also report the maximum percentage of contaminated memory state separately in Figure 7f.

We notice that the iterative nature of these scientific applications produces a deep contamination of their memory state. When faults contaminate the velocity or position of a particle P, the interaction of P with other particles and the forces induced on the latter by P will produce wrong movement or energy charges. The particles affected by P will also be contaminated and the process will repeat exponentially in the next time steps. Eventually, given enough iterations, all the memory state can become contaminated. In the following, we analyze each application separately.

Figure 7a shows how faults propagate in *LULESH* [20], a shock hydrodynamics proxy application developed by the ASCR ExMatEx Exascale Co-Design Center to model numerical algorithms, data motion, and programming style of typical scientific applications. The application solves a simple Sedov blast problem with analytical answers. As depicted in the plots, injected faults progressively propagate into the application's state. This is the result of the iterative structure of the application wherein the results of a time step i (speed and position of the fluid) are used as input of time step $i+1$. With a closer look at the graph, it is possible to identify the time steps. Within each time step the number of contaminated memory locations remains roughly constant while between one time step and the next the number of contaminated memory locations increases. Figure 7a also shows that the propagation of faults follow the same trend in all cases, regardless of the correctness of the final output or if the application takes longer to converge.

LAMMPS [33] is a molecular dynamics code that models an ensemble of particles in a liquid, solid, or gaseous state. The application computes Newton's equations of motion for system of interacting particles and can model atomic, polymeric, biological, metallic, granular, and coarse-grained systems using a variety of force fields and boundary conditions. We solve the Cu metallic solid with embedded atom method (EAM) potential which involves the dynamics of 32,000 atoms for 100 time steps. Figure 7d shows that faults injected in the application progressively propagate through the memory state at every time step. A fault that corrupts the velocity or the position of a molecule at time step i will induce wrong forces to the adjacent molecules at time step $i+1$. Within 100 time steps, more than half of the memory state becomes contaminated (see Figure 7f), which results in more than half the case of corrupted results in Figure 6. An interesting case is represented by the lower profile in Figures 7d. In this case, the injected fault corrupted a static data structure that is not used during the computation, thus

the fault does not propagate to the rest of the application's memory state. Note that this case was not identified in the previous experiments based on output variation analysis.

miniFE is a DOE proxy application that implements several kernels representative of implicit finite-element applications. In particular, the application assembles a sparse linear-system from the steady-state conduction equation on a brick-shaped problem domain of linear 8-node hex elements. Next, *miniFE* solves the linear-system using a simple unpreconditioned conjugate-gradient (CG) algorithm and compares the computed solution to an analytical model for steady-state temperature in a cube. Figure 7c presents fault propagation profiles for *miniFE*. We can distinguish in the graph the assembly of the linear system in the first part (which mainly consists of scattering element-operators into sparse matrix and vector) from the CG solving phase (sparse matrix-vector products). Faults injected in the initialization quickly propagate and contaminate the sparse matrix and vector (as in the dense example in Figure 1) and, reach a steady state maintained in the solving phase. Faults injected in the solving phase quickly reach a steady state. As we can see from the graph, the two cases with wrong results cause different behaviors: for the left-most case, the internal check on the sparse matrix and vector fails and the application aborts before starting the solving phase. In the right-most case, the application does not converge and terminates after reaching the maximum number of iterations. Given the sparsity of the matrix and vector, even a small percentage of contaminated memory locations (see Figure 7f) can lead to corrupted results or prolonged executions.

AMG2013 [19] is a parallel algebraic multi-grid solver for linear systems arising from 3-D problems on unstructured grids. The communications and computations patterns exhibit the surface-to-volume relationship. We use the default problem, a 3-D Laplace type problem on an unstructured domain with an anisotropy in one part. The fault propagation results for *AMG2013* are shown in Figure 7b. The application performs three different phases that can be identified in the figure, especially when the fault is injected early during the execution of the application: Initialization, Setup of the conjugate gradient pre-conditioner and the Solving phase. A close look at the data reveals that faults injected in the early initialization phase propagates slowly at first and then ramps up when starting the setup phase. During the setup, the amount of memory location contaminated remains roughly constant, which indicates that the unstructured grid becomes quickly and completely contaminated. Finally, in the solving phase, *AMG2013* allocates the data structures required to solve the Laplace problem: as we can see from the graph, faults quickly propagate in the memory state of the application contaminating more memory locations at every iteration of the solver. In two cases, the fault injected contaminates data structures not involved in the solving phase. In these cases, the amount of contaminated memory locations remains stable at the value reached during the setup phase.

MCB models the solution of a simple heuristic transport equation using a Monte Carlo technique. The application employs typical features of Monte Carlo algorithms such as particle creation, particle tracking, tallying particle information, and particle destruction. The heuristic transport equation models the behavior of particles that are born, travel with a constant velocity, scatter, and are absorbed.

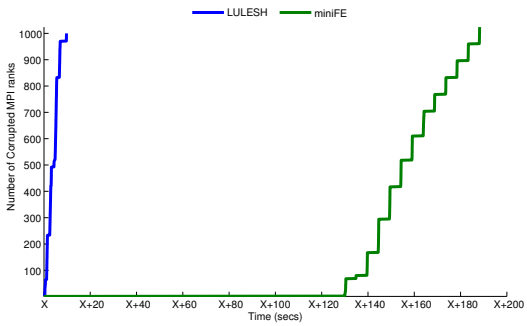


Figure 8: This graph shows how an injected fault propagates across different MPI processes for *LULESH* and *miniFE*.

MCB achieves parallelism through domain decomposition of the physical space and threading. When particles hit the boundary of a domain, they are buffered and then sent using a non-blocking MPI call to the processor simulating the domain on the other side of the boundary. Figure 7e shows the typical fault propagation profile that we have seen for other iterative applications. Faults propagate from one particle to the other during their movement and from one MPI process to another when a particle move across domains. Interestingly, even late-injected fault can still corrupt the output.

Propagation through MPI processes: We now analyze how faults propagate across different MPI processes. As we explained earlier, we inject a single fault into a randomly selected MPI process. However, that process may send contaminated data to other MPI processes and, thus, corrupt their address space. Figure 8 shows two examples, *LULESH* and *miniFE*, in which an initial fault injected at a certain time X into MPI process 4 and 6, respectively, propagates and contaminates all other MPI processes. For *LULESH* faults propagate immediately to all other MPI processes and spreads very quickly, as MPI exchange data at the end of an iteration. For *miniFE*, instead, the fault does not propagate until very late in the execution, but then spreads quickly to all other MPI processes.

Categorization Based on Fault Propagation: The results presented in Figure 6 and 7 are somehow contradictory. Figure 6 shows that the tested applications can tolerate the presence of faults during their execution and still produce correct results. For example, in 90% of the cases *LULESH* produces correct results in the presence of a randomly injected fault. Following this data, the user could decide to employ a light-weight resilience mechanism to protect *LULESH*, given the relatively robust nature of the application. In reality, however, the application is quite sensitive to transient errors, as *LULESH*'s memory state might be corrupted even when the final output is correct (Figure 7). This observation holds for the other applications as well. This means that only using the results of fault injection experiments may lead to incorrect conclusions and the deployment of resilience mechanisms that are not adequate.

Using our fault propagation framework, we are able to distinguish cases in which transient errors propagate through the application's memory state from the cases in which a fault is masked at processor level before contaminating any memory location. We analyzed the breakdown of the CO

Table 2: Fault propagation speed factors.

App.	LULESH	LAMMPS	MCB	AMG2013	miniFE
FPS	0.0147	0.0025	0.0562	0.0144	0.0035
SDev	1.48E-4	0.96E-4	26.7E-4	6.82E-4	2.89E-4

cases in Figure 6 and divide it into two categories: Vanished and ONA. A deeper analysis of the internal propagation of faults reveals that most cases (over 98%) identified as CO in Section 4.2 present corrupted memory states. The number of cases in which faults are masked at processor level before propagating to memory is surprisingly low. We believe that this may be due to the fact that LLFI injects faults into live registers and that these faults have a higher probability of propagating to memory than faults injected into dormant registers. Previous work has also identified similar discrepancies between circuit- and register-level fault injection [13]. Nevertheless, these results show that it would be dangerous to assume that the tested applications can tolerate the presence of faults while, in reality, they may produce incorrect results in a slightly different execution context.

5. FAULT PROPAGATION MODELING

In Section 4.3, we observed that faults generally propagate linearly in the application's state during the execution. In this section, we use this observation to build a fault propagation model that can be used to estimate the number of corrupted memory locations (CMLs) at a time t, once a fault is detected at time t_f. From the graphs in Figure 7, it is evident that each fault propagation profile can be expressed as a function of the execution time with a piece-wise equation that is linear in the first sub-domain and constant in the second. The linear part of the profile is the most interesting because the different profiles characterize the sensitivity of the applications to faults. We employ machine learning techniques to derive a generic close form of the fault propagation profile. For each experiment, we can express the specific fault propagation profile as

$$CML(t) = a \cdot t + b \tag{1}$$

where t is the time during the execution, a expresses how quickly a fault propagates in terms of memory locations corrupted per second, and b indicates the time t_f where the fault occurs. We employed standard validation techniques to verify the accuracy of each model. Our results show that the errors are within 0.5% of the actual CML values. The value of b in a particular execution can be derived from the time t_f in which the fault occurs.[3]

$$b = -a \cdot t_f \tag{2}$$

By applying machine learning techniques to each fault propagation experiment, we obtain a family of linear functions. If we abstract from the time t_f where a fault is injected, hence b, we can compute the fault propagation speed (FPS) factor for each application as the average of the a factors from each model. The FPS expresses the rate at which transient errors propagate into the application's state. The metric can be used operatively to estimate the number of CMLs within a time interval (t_1, t_2), even if the exact time at which the transient error occurred, t_f, is not known. For example, assume that no fault was detected at time t_1 and that a fault is detected at time t_2. The application FPS can then be used to estimate the maximum number of CMLs, as:

$$max(CML(t_1, t_2)) = FPS \cdot (t_2 - t_1) \tag{3}$$

The above formula is an upper-bound of the maximum number of CMLs that assumes fault time, t_f, is close to the lower extreme of the interval, t_1. On average, $t_f = (t_2 - t_1)/2$, hence the average number of CMLs in the time interval (t_1, t_2) is $avg(CML(t_1, t_2)) = max(CML(t_1, t_2)/2)$, as expected.[4] The estimation provided by our model can be used to decide, at runtime, if a roll-back should be triggered. For application with low FPS, i.e., relatively robust applications, the fault-tolerance system could decide to keep the application running if the CML at the end of the application is predicted to be below a safe threshold.

Table 2 reports the FPS computed for each application. To the contrary of what is indicated in Figure 6, Table 2 shows that, when considering the CML in the application's state independently of the final output, *LULESH* is much more vulnerable than *LAMMPS*, as faults propagate at a rate of 0.0147 CML/sec in the former and 0.0023 CML/sec in the latter. *MCB* is the most vulnerable application among the ones tested. For this application faults propagate at a rate of 0.0531 CML/sec. We believe that this is a property of the Monte Carlo method used in the application, which is almost embarrassingly parallel. Interestingly, *LAMMPS* and *miniFE*, which are the applications with the largest percentage of wrong results in Figure 6, are the applications with the lowest FPS. This indicates that, compared to the other applications, faults propagate at a much lower rate in these two applications but the error margins used to accept the final solutions are stricter. We believe that FPS is a more precise way to assess the intrinsic vulnerability of an application because it takes into account the entire application's structure and not just the final outcome.

6. RELATED WORK

Fault Injection: Fault injection can be performed at various abstraction layers, from circuit to application level. Cho et al. [13] present a study of the accuracy of fault injection at higher abstraction layers. The authors report that single bit-flips at the circuit-level such as those in flip-flops are difficult to model at the register-file or architecture level. Fault injections at the circuit-level is considered the most accurate method but it requires sophisticated infrastructures, such as radiation-exposure to processor chips [11, 12], or processor RTL simulations [13, 30]. Next, cycle-accurate microarchitecture-level simulators [24] and architecture-level simulators [6, 17] have been proven reliable. Both options might limit the size and scale of the applications and systems under study. Software-Implemented Fault Injection (SWIFI) [9, 26, 23, 29, 37, 38, 42] can be used to perform accelerated fault injection at application level. Li et al. [23] propose a fault injection system based on Pintool [28] (a binary instrumentation tool for x86 systems), wherein a single fault is randomly injected into data-structures of parallel scientific applications and the correlation between the fault outcome and the location of the injected faults is analyzed.

[3]We assume that the fault is detected when it occurs. In reality, there might be a delay between the occurrence and the detection of the fault Δt that needs to be taken into account in the computation of b.

[4]$min(CML(t_1, t_2)) = 0$, assuming $t_f = t_2$.

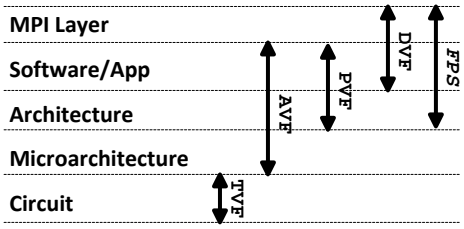


Figure 9: Fault-Masking levels investigated by related works as compared to proposed work.

Such fault injection is done directly in the application state, by corrupting global, heap or stack memory regions and does not consider faults that may be masked at the architecture level, before reaching memory.

On the other hand, compiler-level SWIFI tools such as LLFI [42] and KULFI [37] inject faults at register level, which provides a chance for the faults to be masked at the register level before being committed to memory. The accuracy of LLFI in assessing applications resiliency as compared to dynamic instrumentation based fault injection has also been evaluated [42]. LLFI is found to provide adequate information about applications vulnerability to soft errors. We adopt a compiler-level fault injection approach based on LLFI with extensions to inject multiple faults into an MPI parallel application. It is noteworthy to mention that similar extensions for KULFI are proposed by *FlipIt* [9], which was developed in parallel to this work.

Vulnerability Metrics: Different metrics at various abstracted layers have been presented in literature with distinct goals. Figure 9 summarizes some of the resilience and vulnerability metrics commonly used. At the lowest level of abstraction, circuit-level masking is quantified using the *timing vulnerability factor (TVF)* [36] which is used to determine the probability of a fault occurring within the setup and hold time windows of the Flip-Flops in the circuit. Moving upwards to assess the vulnerability of macro hardware structures inside of a processor like register files, arithmetic logic units, re-order buffer, the *architectural vulnerability factor (AVF)* [32] was proposed to determine the probability of a fault in each of these structures causing an error in the final application outcome. In a subsequent work, the *program vulnerability factor (PVF)* [39] was proposed to evaluate the software reliability independent of the underlying microarchitecture so as to propose changes at the software or application-level. Thus, AVF can be used to determine both the architecture and microarchitecture level masking effects on the application outcome, whereas PVF focuses on architecture-level masking effects. Both AVF and PVF are computationally-intensive, especially if applied to large parallel, distributed-memory applications, such as those tested in this work. Similarly, instruction categorization is done based on derating of a single bit flip in architectural registers [14] and the *application derating* metric is proposed [5]. PVF or AVF analysis can be done using architecturally correct execution (ACE) [32] analysis or using fault injection. On a side note, previous works [40] have pointed out inaccuracies of ACE analysis as compared to fault injection studies, which can lead to overestimation of protection mechanisms.

In a complementary work [18], fault propagation was studied across MPI boundaries. However as the faults are directly injected into data structures in distributed memories, the architecture-level masking effects cannot be quantified. Similarly, the *data vulnerability factor (DVF)* [43] is used to capture the vulnerability of data structures inside of an application by estimating the memory access patterns. While AVF, PVF and DVF are scalar metrics, the methodology and metric proposed herein provide detailed information about the internal application memory state within a single process and across multiple processes in distributed applications, while providing information about architecture- and software-level masking similar to PVF metric.

Fault Propagation: Li et al. [23] use application knowledge to analyze the results and qualitatively correlate the outcomes of injected faults to the locations where fault was injected. They assess that a fault has propagated if the final result of the application has been corrupted or the execution has been prolonged. In this work, we show that faults may propagate into the application's memory state even if they do not corrupt the final state of the application and we show the speed and depth with which faults propagate. This quantitative analysis is important to understand the underlying vulnerability characteristics of an application and to select the most effective fault tolerance system. Similar application-specific studies have been performed for multi-grid solvers [10] and iterative linear algebra methods [7]. For example, Casas et al. [10] study the effect of fault propagation across various phases of *AMG* by observing the deviation of known data structures from fault-free values with the final goal of protecting critical pointers. In contrast, we seek a generic methodology that allows the user to study a larger set of applications.

Fault Detection: Fault detection is orthogonal to this work, yet important for the design of resiliency systems [16, 25]. Faults are detected using a variety of techniques, including symptom-based, compiler-assisted, and simulation techniques [17, 21, 27, 34, 35]. These studies are mostly done at low-scale, while we target large-scale systems.

7. CONCLUSIONS

Power constraints of exascale systems will encourage use of technologies like massive parallelism and NTV, which may result in higher errors and impact the correctness of scientific applications. Despite its importance, the vulnerability of HPC applications has not received enough attention, primarily because of the lack of tools that allow researchers to perform accelerated fault injection and analyze how injected faults propagate in the application's state. In this work, we introduced a novel fault propagation framework that is capable of accurately tracking the propagation of faults into parallel MPI applications. The framework provides programmers with new insights about the vulnerability of applications to transient errors and the correlation with the application structure.

We present an analysis of several mission-critical HPC applications. Our results indicate that even a single fault introduced at register level can contaminate a consistent part of the application's state. We show that faults generally propagate linearly and progressively corrupt the address space and that errors contaminate other MPI processes through message passing communication. We used this observation to derive application fault propagation models that can be used at runtime to estimate the number of corrupted mem-

ory locations once a fault is detected. We also show that analyzing the vulnerability of parallel applications with a "black-box" approach based on output variability analysis resulting from statistical fault injection may lead to incorrect conclusions. In particular, our tool is capable of differentiating the cases in which a fault is completely masked at processor level and does not propagate to memory from those cases in which the application's memory state becomes contaminated, even though the final results appear correct.

Acknowledgements

This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under award number 59542 "Performance Health Monitor"; program manager Lucille T. Nowell.

8. REFERENCES

[1] Collaboration Oak Ridge, Argonne and Livermore. https://asc.llnl.gov/CORAL/.

[2] The top500 supercomputers list. http://www.top500.org.

[3] Advanced Scientific Computing Research (ASCR). Scientific discovery through advanced computing (SciDAC) Co-Design. http://science.energy.gov/ascr/research/scidac/co-design/.

[4] R. Balasubramanian, Z. York, M. Dorran, A. Biswas, T. Girgin, and K. Sankaralingam. Understanding the impact of gate-level physical reliability effects on whole program execution. In *the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, February 2014.

[5] C. Bender, P. Sanda, P. Kudva, R. Mata, V. Pokala, R. Haraden, and M. Schallhorn. Soft-error resilience of the IBM POWER6 processor input/output subsystem. *IBM Journal of Research and Development*, 52(3):285–292, May 2008.

[6] S. Bohm and C. Engelmann. xSim: The extreme-scale simulator. In *The Int. Conference on High Performance Computing and Simulation (HPCS)*, July 2011.

[7] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 155–164, 2008.

[8] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, 31(2):6–15, Mar. 2011.

[9] J. Calhoun, L. Olson, and M. Snir. FlipIt: An LLVM based fault injector for HPC. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 547–558. Springer International Publishing, 2014.

[10] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, 2012.

[11] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller. Understanding soft error resiliency of BlueGene/Q compute chip through hardware proton irradiation and software fault injection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC, 2014.

[12] C.-Y. Cher, K. Muller, R. Haring, D. Satterfield, T. Musta, T. Gooding, K. Davis, M. Dombrowa, G. Kopcsay, R. Senger, Y. Sugawara, and K. Sugavanam. Soft error resiliency characterization on IBM BlueGene/Q processor. In *the 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2014.

[13] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *the 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013.

[14] J. J. Cook and C. B. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *the International Conference on Dependable Systems and Networks (DSN)*, Anchorage, AK, 2008.

[15] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb 2010.

[16] C. Engelmann, H. H. Ong, and S. L. Scott. The Case for Modular Redundancy in Large-Scale High Performance Computing Systems. In *Proceedings of the 27th IASTED Int. Conference on Parallel and Distributed Computing and Networks (PDCN)*, Innsbruck, Austria, Feb. 2009.

[17] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *the 15th ACM Architectural Support for Programming Languages and Operating Systems*, ASPLOS, Pittsburgh, PA, 2010.

[18] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, 2012.

[19] V. E. Henson and U. M. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, Apr. 2002.

[20] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, May 2013.

[21] D. S. Khudia, G. Wright, and S. Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *the 13th ACM Int. Conf. on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES, Beijing, China, 2012.

[22] P. Kogge, K. Bergman, S. Borkar, D. Campbell,

W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. A. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report DARPA-2008-13, DARPA IPTO, September 2008.

[23] D. Li, J. S. Vetter, and W. Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC, Salt Lake City, Utah, 2012.

[24] M.-L. Li, P. Ramachandran, U. Karpuzcu, S. Hari, and S. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *the IEEE 15th Int. Symp. on High Performance Computer Architecture (HPCA)*, Feb 2009.

[25] C. Lu. Failure data analysis of HPC systems. *CoRR*, abs/1302.4779, 2013.

[26] C.-d. Lu and D. A. Reed. Assessing fault sensitivity in MPI applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, 2004.

[27] Q. Lu, K. Pattabiraman, M. Gupta, and J. Rivers. SDCTune: A model for predicting the SDC proneness of an application for configurable protection. In *the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10, Oct 2014.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conf. on Programming Language Design and Implementation*, PLDI, Chicago, IL, USA, 2005.

[29] R. Maia, L. Henriques, D. Costa, and H. Madeira. Xception - enhanced automated fault-injection environment. In *the International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[30] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris. Instruction-level impact analysis of low-level faults in a modern microprocessor controller. *IEEE Transactions on Computers*, 60(9):1260–1273, Sept 2011.

[31] S. Mitra, P. Bose, E. Cheng, C.-Y. Cher, H. Cho, R. Joshi, Y. M. Kim, C. R. Lefurgy, Y. Li, K. P. Rodbell, K. Skadron, J. Stathis, and L. Szafaryn. The resilience wall: Cross-layer solution strategies. In *Proceedings of International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, 2014.

[32] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, 2003.

[33] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[34] S. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *the International Conference on Dependable Systems and Networks (DSN)*, June 2008.

[35] S. Sastry Hari, S. Adve, H. Naeimi, and P. Ramachandran. Relyzer: Application resiliency analyzer for transient faults. *IEEE Micro*, 33(3):58–66, May 2013.

[36] N. Seifert and N. Tam. Timing vulnerability factors of sequentials. *IEEE Transactions on Device and Materials Reliability*, 4(3):516–522, Sept 2004.

[37] V. C. Sharma, A. Haran, Z. Rakamarić, and G. Gopalakrishnan. Towards formal approaches to system resilience. In *the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013.

[38] D. Skarin, R. Barbosa, and J. Karlsson. GOOFI-2: A tool for experimental dependability assessment. In *the International Conference on Dependable Systems and Networks (DSN)*, June 2010.

[39] V. Sridharan and D. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 117–128, Feb 2009.

[40] N. J. Wang, A. Mahesri, and S. J. Patel. Examining ACE analysis reliability estimates using fault-injection. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 460–469, 2007.

[41] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *the International Conference on Dependable Systems and Networks (DSN)*, 2004.

[42] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *the International Conference on Dependable Systems and Networks (DSN)*, June 2014.

[43] L. Yu, D. Li, S. Mittal, and J. S. Vetter. Quantitatively modeling application resilience with the data vulnerability factor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 695–706, 2014.