Chaser: An Enhanced Fault Injection Tool for Tracing Soft Errors in MPI Applications

Qiang Guan*, Xunchao Hu †, Terence Grove¶, Bo Fang‡, Hailong Jiang*, Heng Yin§, Nathan DeBardeleben¶
* Deparment of Computer Science, Kent State University
Email: {qguan, hjiang13}@kent.edu
†DeepBits Technology
Email: xchu@deepbitstech.com
‡University of British Columbia
Email:bof@ece.ubc.ca
§Department of CSE, University of California, Riverside
Email:heng@cs.ucr.edu
¶ Los Alamos National Laboratory
Email:{tagrove, ndebard}@lanl.gov

***Abstract*—Resilient computation has been an emerging topic in the field of high-performance computing (HPC). In particular, studies show that tolerating faults on leadership-class supercomputers (such as exascale supercomputers) is expected to be one of the main challenges. In this paper, we utilize dynamic binary instrumentation and virtual machine based fault injection to emulate soft errors and study the soft errors' impact on the behavior of applications. We propose Chaser, a fine-grained, accountable, flexible, and efficient fault injection framework built on top of QEMU. Chaser offers just-in-time fault injection, the ability to trace fault propagation, and flexible and programable interfaces. In the case study, we demonstrate the usage of Chaser on Matvec and a real DOE mini MPI application[1].**

***Index Terms*—soft error; MPI; fault injection; resilience; vulnerability; High Performance Computing.**

I. Introduction

Resilient computation has been an emerging topic in the field of high-performance computing (HPC) for several years. In particular, studies show that tolerating faults on leadership-class supercomputers (such as exascale supercomputers) is expected to be one of the main challenges. Due to high error rates, soft errors [36] pose a serious threat to the prospect of exascale systems.

The HPC community often lumps all reliability under the umbrella of "resilience." However, there are many ways that failures can affect supercomputers and the applications running on them. Faults can be transient, intermittent, or permanent. They can cause silent data corruptions (SDC), crash the application, or cause hardware components to fail. Faults can come from any range of events including (but not limited to) aging [9], [33], poorly designed components [13], material impurities/packaging [32], [41], [29], neutron ("cosmic") radiation [12], [7], electromagnetic interference [30], electromigration [24], temperature [6], [23], voltage [11], [37] extremes/fluctuations, and nefarious tampering/espionage [31]. Furthermore, faults are not relegated to only being driven by hardware influences; software plays at least some non-trivial part in the rate of faults on computing systems today [28], [15], [19], [16]. The lack of tools for characterizing and studying these types of faults was brought up in a recent governmental position paper on the challenges of resilient exascale computing[10]. Without proper tools it is unlikely that key advancements in HPC resilience will occur in time for next generation systems.

Researchers have built various fault injection tools to characterize the error resilience of applications. Current fault injection tools typically rely on either source code instrumentation [38], [39], [8], [35] or dynamic binary instrumentation [18], [26], [25], [40]. Such tools provide the capability of emulating and injecting faults into targeted programs with a variety of fault models. However, nearly all fault injectors treat the application under test as a black box, as the main goal of fault injection is to characterize the impact of soft errors.

On the other hand, studying error propagation within a program is considered a critical task in the dependability community. When a fault occurs, understanding how it affects the program's state and final outcome can be valuable to guide the design and implementation of fault tolerant systems. Prior studies [20], [27] have found that with more detailed information about error propagation, applications can be optimized and enhanced in robustness against soft errors. Traditionally, tracing faults during the execution of a program requires additional computational/memory resources, which inevitably incurs significant performance overhead. This problem can be exacerbated in the context of MPI applications, as the fault might pass the process boundaries, requiring the tracer to consider both the depth of the program (i.e. the complexity in the single process), and the width of the program (i.e. across multiple processes/nodes).

Statistical-based fault injection campaigns generally con-

[1]This manuscript is submitted for DSN2020 tool category
[1]LA-19-27437

978-1-7281-5809-9/20/$31.00 ©2020 IEEE
DOI 10.1109/DSN48063.2020.00051

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 30,2020 at 22:41:12 UTC from IEEE Xplore. Restrictions apply.

sume a large amount of cycles to reach a statistically significant estimate of the impact of faults. Additionally, such campaigns can take hours or days to finish [10]. For HPC applications, performance has become a major concern when running a large number (i.e. thousands) of fault injections. As such, minimizing the overhead introduced by the fault injector is critical. Therefore, a flexible, lightweight injection framework with a dynamically configurable error tracer is needed. Such a framework must also be able to record the footprint of errors in memory and between MPI processes.

In this paper we utilize dynamic binary instrumentation and virtual machine based fault injection to emulate soft errors. We can then study the impact that such errors have on the behavior of targeted applications. We propose Chaser, a fine-grained, accountable, flexible, and efficient soft error fault injection framework built on top of DECAF [14]. Chaser provides just-in-time fault injection, fault propagation tracing, and flexible fault injection interfaces. More specifically, Chaser allows the user to define which application to inject soft errors into, as well as when and where these soft errors will be injected, with various levels of granularity. Researchers can also build different fault injectors using the interfaces exported by Chaser. We demonstrate the use of Chaser against applications from Matvec [2] and an MPI-based DOE mini-app CLAMR [1]. Because of the exported fault injection interfaces, the effort of developing new fault injectors is substantially reduced, only requiring around 100 lines of code.

Our main contributions are summarized as follows:

1) We propose Chaser, a fine-grained, accountable, flexible, and efficient soft error fault tracer framework built on top of DECAF.
2) We implement Chaser and evaluate its performance and flexibility. Chaser can supervise the process of parallel fault injection as well as provide tracing functions. Users can trace the error propagation between parallel MPI processes as well as between computing nodes.
3) We demonstrate Chaser with CLAMR, a DOE mini application, to study error propagation using trace logs.

The rest of paper is organized as follows: Section II introduces the design requirements and goals for Chaser and explains its major components and functions; Section III presents the implementation of key functions and prototype of Chaser. Next, evaluation results are presented in Section IV, and the state-of-the-art studies related to this work are discussed in Section V. Finally, we present our conclusions in Section VI.

II. Chaser Design

A. Design Requirement

There are challenges in tracing errors in parallel computing environments dynamically. For instance, the tracing overhead can also significantly lower performance. Therefore, an error tracer should be able to mark the errors and specifically track the trajectory of errors in memory space across the execution of an application. Secondly, current fault injection tools often hypervise the injection locally in a single process of MPI applications due to the performance and management cost. It is possible for most fault injection tools to locally monitor error propagation, however, they may be unable to track error-corrupted messages shared between MPI ranks on different physical nodes. The tracer tool should be able to coordinate all MPI processes and notify the MPI processes of the incoming errors synchronized by other MPI processes.

These incoming errors behave like “injected errors“ and manifest locally again. An error propagation example is shown in Fig 1.

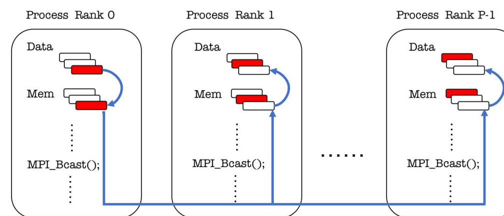

Fig. 1: Error propagation example between MPI ranks. The error injected in rank 0 will begin by propagating locally and then contaminate the memory of other ranks.

B. Design Goal

The design of Chaser is based on the Parallel Fine-grained Soft Error Fault Injector (PFSEFI) [18], [17] as an add-on function. It is designed to achieve the following goals.

a) Fine-grained: The fault tracer should be able to inject faults into a designated application and instruction after specific conditions are met. This allows researchers to construct various complex fault models to study soft errors' impact on the behavior of applications.

b) Accountable: The fault tracer should be able to trace how injected faults propagate within the application. By providing detailed information about traces across all MPI ranks, researchers are able to analyze how faults impact the behavior of an application in order to propose new resilient algorithms.

c) Flexible: The fault tracer should allow users to customize and construct different fault models. This allows researchers to design their own fault propagation experiments without building the entire system from scratch.

d) Efficient: The fault tracer should introduce minimal performance overhead to the system, since fault injection and propagation experiments usually requires a large number of runs.

C. Chaser Architecture

We propose Chaser, a new fault injection framework, to satisfy the above design goals. Figure 2 shows the overall architecture of Chaser. Inside the virtual machine, we run the target application and conduct fault injection on it externally via fault injection interfaces. To provide various fault injection capabilities, Chaser is extensively involved with the dynamic binary translation process, which is detailed in Section III-A. Chaser has the following key components:

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 30,2020 at 22:41:12 UTC from IEEE Xplore. Restrictions apply.

a) Just-in-time Fault Injection: This component is able to inject faults into the target process when the predefined conditions are met. Unlike F-SEFI [18] which rewrites the dynamic binary translation process for every instruction to allow fault injection, Chaser only inserts fault injection logic when that instruction is marked as a targeted instruction by the user. Since only a tiny portion of targeted processes are instrumented, this design significantly reduces the performance overhead as demonstrated in Section IV-D. Meanwhile, the previous works on fault injection [18], [4], [38], [26], [25] target a specific task or fault model. This makes customization of the fault injector very difficult for users. To address this, Chaser exports its fault injection capabilities as interfaces. Using the exported interfaces, users can then define their own fault models by setting the injection location, target instruction, etc. Details are further discussed in Section III-B.

b) Fault Propagation Tracing: Chaser traces the propagation of faults via the dynamic tainting technique [42]. It leverages DECAF's bitwise tainting [21], [22] and extends its tainting for floating point instructions. While instruction level traces can record the most complete information about fault propagation, the performance penalty is unacceptable in practice. In contrast to instruction level tracing, Chaser records tainted memory access activity only. This design sacrifices the completeness of fault propagation tracing to an acceptable degree while incurring a reasonable performance overhead. Details are discussed in Section III-C.

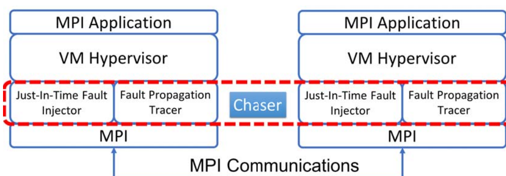

Fig. 2: The overall architecture of Chaser.

III. Implementation

A. Fault Injection Component

a) Dynamic binary translation in QEMU: To support multiple architectures, QEMU makes use of a compiler backend called Tiny Code Generator (TCG) as its dynamic binary translation engine. QEMU translates each guest instruction into a series of architecture-independent TCG instructions grouped together as a TCG translation block (TB). The TCG compiler translates each TB into a piece of native code that can be executed on the host.

b) Placement of Fault Injector: The fault injector is placed where the target process starts and the interested instruction is translated. Chaser relies on DECAF's built-in Virtual Machine Introspection (VMI) technique to retrieve the process's states. Once the target process creation event is captured, Chaser flushes the code translation cache and triggers the next round of binary code translation. During this translation process, the fault injector is injected into the instruction of interest. Figure 3 illustrates a demonstration of how the fault injector is injected into an *fadd* instruction. (a) is the original instruction *fadd* while (b) is the generated TCG IR by QEMU without the fault injector. If the *fadd* is labeled as the targeted instruction, Chaser generates a callback function *DECAF_inject_fault* and invokes it before the *fadd* is executed. As shown in (c), this is achieved by inserting the TCG IR code at the beginning of the *fadd*'s translated TCG IRs.

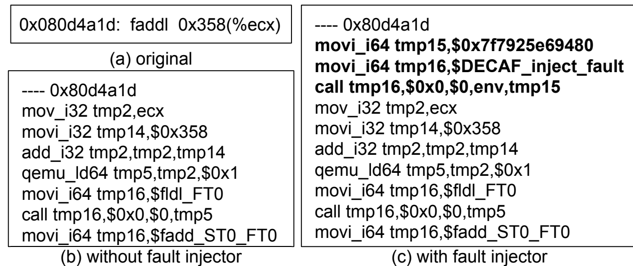

Fig. 3: Demonstration of fault injection for an *fadd* instruction.

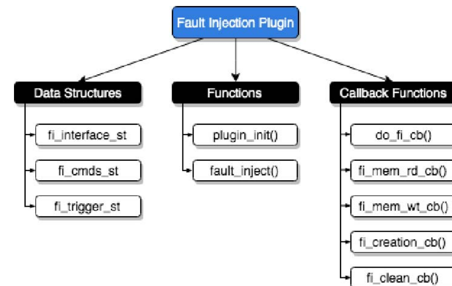

Fig. 4: Overview of Chaser's fault injection plugin.

c) Software Structure of Fault Injection Plugin: Figure 4 presents the software structure of the fault injection plugin of Chaser, which injects faults for designated instructions into the target application. When this plugin is loaded into the fault injection framework, its `plugin_init()` function is called to initialize the plugin and returns a pointer to `fi_interface_st`, which specifies a new terminal command `inject_fault` defined by the plugin. Additionally, the plugin registers callback `fi_creation_cb()` for the process creation event.

When the user enters the `inject_fault` command in the terminal, the registered callback `do_fi_fault()` is called. This callback function sets the targeted program, targeted instruction, injection condition, and self-defined fault injector, and saves this information into `fi_cmds_st`. Once there is a newly created process, the callback `fi_creation_cb()` is called to check if it is the targeted program. If so, it enables the fault injector and registers two callbacks - `tainted_mem_wt_cb()` and `tainted_mem_rd_cb()`, to log the fault propagation process.

During the execution of the targeted program, `fault_injector()` is invoked before the targeted instruction (*fadd*, *mov*, etc.) is executed. The instructions

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 30,2020 at 22:41:12 UTC from IEEE Xplore. Restrictions apply.

executed counter is then updated. Once the value of the instructions executed counter reaches the injection condition, the actual fault injection is called to emulate different kinds of soft errors, which are defined in `fi_trigger_st`. After the injection is finished, the callback function `fi_clean_cb()` will turn off the screening on incoming new processes and detach the injector.

It is worth noting that the design of Chaser satisfies our design goals mentioned in Section II-B. The user can customize the fault injector (**flexible**) to inject faults into a target program and instruction (**fine-grained**) under customized injection condition (**accountable**). And because only the targeted program and instruction are instrumented, the performance overhead is minimized (**efficient**). Detailed performance overhead results are discussed in IV-D.

B. Fault Injection Interfaces

Chaser exports its fault injection capabilities as interfaces for users. These interfaces allow users to customize the fault injector in the following ways.

a) what application: Leveraging DECAF's Virtual Machine Introspection technology, users can specify the targeted application at runtime. Using the callback function `VMI_CREATEPROC_CB`, users can retrieve the information of a created process to determine if it is the application targeted for fault injection.

b) when to inject: Chaser allows the user to define different injection conditions for every X86 instruction. At runtime, these conditions are checked by the user to determine when to inject faults. For example, to inject a fault to *fadd* after it is executed 1000 times, the user can define when to inject faults and and how many bits to flip through the data structure template defined in Chaser. This design makes Chaser both highly customizable and extendable. Chaser, by default, provides three types of fault models: *probabilistic fault model, deterministic fault model and group fault model*. Table I shows the definitions of the supported fault models.

TABLE I: Chaser supported fault models

Fault Model	Functions
Probabilistic	fault injection location is based on a predefined probability distribution function.
Deterministic	fault injection location is the exact predefined location.
Group	multiple faults are injected.

c) how to inject: For every X86 instruction, the user can define custom fault injectors. Chaser maintains a function pointer to the fault injector for every instruction. If the injection condition is satisfied, the corresponding fault injector is invoked. Chaser also provides functions such as `CORRUPT_REGISTER` and `CORRUPT_MEMORY` to ease the injection process. These functions can write to any user specified registers and memory locations.

TABLE II: Lines of code (LOC) required to develop injectors

InjectorName	**LOC**
Probabilistic Injector	97
Deterministic Injector	100
Group Injector	98

d) Flexibility : We discuss the flexibility of the injection interfaces in terms of lines of code (LOC) and time required to develop new fault injection models. We implemented three fault injectors described in F-SEFI - a probabilistic injector, which injects faults at a predefined probability; a deterministic injector, which injects faults into target instruction at certain condition; and a group injector, which injects faults into all floating point instrcutions. Table II summarizes the LOC and time required to develop these fault injectors. As shown in the table, it takes about 2 hours and 100 LOC to develop a new fault injector, which is a relatively small task for researchers. This demonstrates that with Chaser's injection interfaces, it is much easier to construct new fault injection models compared to starting from scratch, which may require deep systemic knowledge and tedious engineering work.

C. Fault Propagation Tracing

a) Dynamic Taint Analysis in DECAF: Dynamic taint analysis runs an application and observes which computations are affected by predefined taint sources such as user input [34]. The purpose of dynamic taint analysis is to track the flow of information between sources and sinks. This technique allows Chaser to trace the propagation of faults by marking the injected faults as sources.

DECAF implements its lightweight, bitwise taint propagation mostly at the TCG instruction level. To achieve bit-level precision, DECAF propagates tainted bits through CPU registers, memory, and IO devices. TCG translates a basic block of guest instructions into a translation block (TB) of TCG instructions (a). The taint propagation rules are enforced with TCG instructions (b). More details can be found at [21].

b) Taint Propagation for MPI applications: In MPI applications, a fault can propagate from MPI process P1 to MPI process P2. Neither the sender nor the receiver process have enough information to accurately track the propagation of faults through inter-process communication. [4] tackles this problem by adding an extra header to the original MPI message, and then propagating the faults with the assistance of this extra header. This solution requires the modification of source code.

Chaser takes a different approach. Figure 5 illustrates the overview of our solution. We create a `TaintHub` module to store and share the tainting status of MPI messages with different MPI processes. Each MPI process can send and receive messages using standard MPI interfaces. Chaser hooks these functions, extracts the message information from the stack, and then shares them with `TaintHub`.

To ensure the error related information is synchronized between MPI ranks, on the sender side, Chaser hooks the MPI message sending functions to collect the taint status and broadcasts that status to `TaintHub` before sending

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 30,2020 at 22:41:12 UTC from IEEE Xplore. Restrictions apply.

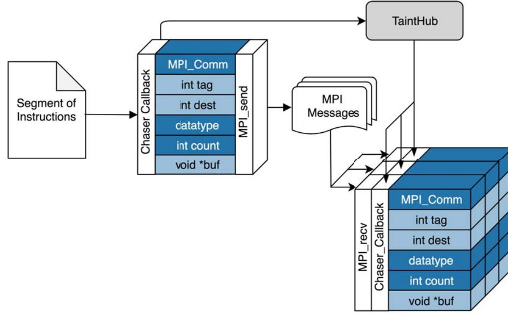

Fig. 5: Design of taint propagation for MPI applications

out the MPI messages to other ranks. When the function `MPI_send` is invoked, Chaser first extracts the `buf`, where the actual message is stored, to check if it is tainted. If it is not tainted, Chaser simply returns without any operation. Otherwise, Chaser extracts `(tag, dest)` as the ID of this MPI message. The taint status of this message will be read using information `( count, datatype, buf)`. Chaser then shares the ID and taint status with `TaintHub`.

On the receiving side, Chaser hooks the MPI message receiving functions to retrieve tainting status of the current MPI message from `TaintHub`. When `MPI_recv` function is invoked. Chaser first extracts `(tag, source)` and uses them to poll `TaintHub` so that the tainting status of the current MPI message can be retrieved. Based on the retrieved information, if the current MPI message is not tainted, Chaser simply returns without any operation. Otherwise, it will mark the MPI message as tainted with the taint status information and the taint propagation continues from one MPI process to the next.

c) Fault Propagation Log: Chaser uses callbacks `DECAF_READ_TAINTMEM_CB` and `DECAF_WRITE_TAINTMEM_CB` from DECAF to record propagation of fault. These two callbacks are invoked when the targeted program reads/writes the tainted memory. Chaser logs the *eip (instruction pointer)* , virtual memory address, physical memory address, tainted value and current value in this memory location for post analysis. We believe that this detailed information will provide us with new ways to analyze and evaluate soft errors' impact on applications.

IV. Case Study

A. Setup and Benchmark

The testbed cluster used for testing Chaser includes four Dell Xeon servers with 16 cores 3.0 GHz CPU and 32GB of RAM. Servers are interconnected with a 10GB network. `TaintHub` is running on the head node of the cluster for fault injection experiments on MPI applications. The cluster is running Ubuntu 14.04 as the OS on each node.

To show the broad usage of Chaser, we evaluate it using applications from two different categories. The first category is common applications that are executed on a single machine. The second category is HPC applications. These two categories combined represent the majority of usage for fault injection techniques.

1) Rodinia : We select bfs, kmeans and lud from Rodinia [3]. During each run, Chaser randomly injects faults into the operands (fadd,fmul and mov) of the *and* instructions after it is executed *n* times and the faults are *x* bits flipped within the operand. We executed each application 5000 times.

2) Matvec: Matvec [2] uses MPI to compute a matrix-vector product $b = A*x$. We use this to illustrate our fault injection capabilities on MPI applications. Matvec is configured to use four MPI ranks running on four Chaser hypervised nodes. Faults are only injected into the master node. During each run, Chaser injects faults into the operands of the *mov* instruction after it is executed *n* times and the faults are *x* bits flipped within the operand. We executed Matvec 5000 times.

3) CLAMR: CLAMR [1] is a DOE mini app, which is used for testing new architectures and runtime systems. CLAMR is a cell-based adaptive mesh refinement application. It supports multiple platforms and provides the result check by applying domain specific mass conservation criteria. CLAMR generates results visualizations that can better help domain experts tune their codes and understand the insight of the physics behind the problem. CLAMR allows the users to define the size of problem, length of simulation, checkpoint frequency and correctness checker.

B. Fault Injection

Figure 6 summarizes the fault injection results for three benchmarks from Rodinia benchmark suite, namely bfs, kmeans and lud, Matvec and CLAMR. In this study, Chaser randomly chooses a floating point instruction or mov instruction at runtime to inject the fault (except for Matvec that only mov instructions are selected), and for each application we perform 3,000 to 5,000 fault injection runs (one fault per run), to reach a statistically significant estimate on the result. The types of failure outcomes include benign (the output data files compared bit-wisely the same as the files from the "golden" run of the benchmark), terminated (either the application crashes due to a OS signal such as SIGSEGV, or the application is terminated due to program level assertion), and silent data corruption (SDC) (the output data files differ bit-wisely from the "golden" run).

Rodinia's BFS contains frequent comparison operations. So we choose *cmp* as the targeted instruction to inject faults at runtime. Kmeans has a computation kernel for calculating the distance between data samples. We inject floating-point faults into Kmeans. For lud, we are using a combined of floating-point and "cmp" faults. Three Rodinia benchmark applications demonstrate the capability of injecting faults into target instructions.

We chose CLAMR as the target of our case study. CLAMR is a cell-based adaptive mesh refinement (AMR) hydrodynamics DOE mini-app that simulates the long range propagation of waves. The computation model uses the shallow water equation to simulate the fluid dynamics and harnesses the three conservation laws of mass, x momentum, and y momentum.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 30,2020 at 22:41:12 UTC from IEEE Xplore. Restrictions apply.

TABLE III: Termination breakdown for MPI application Matvec

Tests	OS Exceptions	MPI error detected	Slave Node failed
Total*	89.77%	9.94%	0.23%
Propagation§	72.77%	27.23%	0

*: The total runs of Matvec. §: The subset of Matvec runs in which faults propagated between master node and slave node.

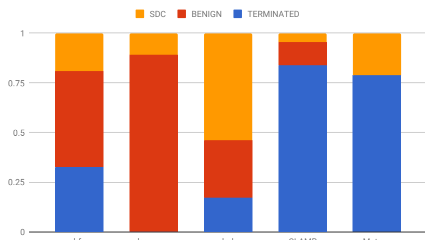

Fig. 6: Fault Injection Results.

We analyze the impact of injecting random transient errors into registers. We run CLAMR 5195 times and randomly inject a single bit error into the floating point instructions in each run. Within 5195 runs, CLAMR detected the injected faults in 4349 (83.71%) runs and 846 (16.28%) resulted in undetected faults. These undetected faults are of interest as they can result in silent data corruption. We further investigated the output of CLAMR with these faults and discovered that 618 (11.89%) of the undetected faults still produced the correct results while 228 (4.38%) of the runs produced incorrect results. This kind of analysis can be useful in discovering the vulnerability of an application to injected faults.

Analysis for terminated cases For MPI applications in our study, the dominant percentage of failures belong to application termination. Chaser is able to provide the understanding of the source of the terminated cases. For example, for Matvec, As shown in Table III, 89.77% of terminated cases are due to OS exceptions such as SIGSEGV, and nearly 9.94% are caused by the MPI runtime exceptions, and interestingly, 0.23% of terminations are due to the faults propagating from the master node to slave nodes. Table III also shows that for the cases where the fault propagate to the slave node, 72.77% of the terminations on the slave node are due to OS exceptions, while the rest are due to MPI runtime errors. Our observation shows that for MPI applications, the majority of the terminations of the application are due to the the faults occurring in the same node.

C. Fault Propagation Analysis

In this section, we examine the characteristics of injected faults and how they propagate. In particular, we are interested in discovering the propagation characteristics from the perspective of memory operations. In the second set of experiments, we run CLAMR 2973 times and in addition to registers.

Tainted bytes in the Propagation. Since it is not possible to show all of the graphs, we randomly selected two fault injection cases from 2973 runs. These two cases are executed again with the same injected faults as the first run. Once the faults are injected, the number of tainted bytes in memory is extracted every 100K executed instructions. The results are seen in Figure 7. The number of tainted bytes finally reaches a constant number in both case 1 and case 2. This is because the injected faults can only affect a fixed portion of the memory. When CLAMR does not access that memory region after some time, the injected faults cease to propagate.

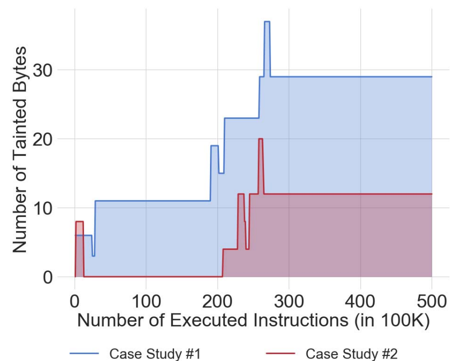

Fig. 7: Termination analysis.

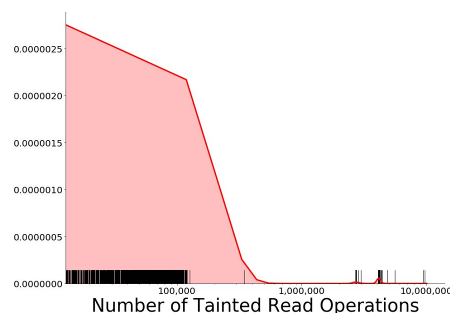

Fig. 8: Distribution of # of tainted memory reads across all MPI ranks over all fault injections runs. For example, most of injected faults will trigger the taint read operation maximum at 2500k times. Majority of the cases are under 800k times of taint read operations.

We also discover that the number of tainted bytes fluctuates during fault propagation. It even drops to zero at times. This is due to the fact that tainted bytes are overwritten by the program with clean data.

Memory Operations In the Propagation Chaser keeps track of two types of memory operations as described in Section III-C. The tainted memory read and tainted memory write operations represent how faults propagate through memory. We count the number of these two operations for every run of CLAMR and summarize them in Figure 8 and 9. The figures illustrate that the number of involved memory operations vary significantly between CLAMR runs. Out of the 2973 total runs, 1402 (47.1%) runs have more tainted memory read operations, 118 (3.97%) runs only have tainted memory read operations, and 444(14.93%) only have tainted memory write operations.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 30,2020 at 22:41:12 UTC from IEEE Xplore. Restrictions apply.

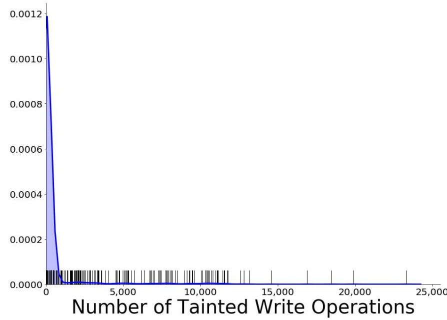

Fig. 9: Distribution of # of tainted memory writes within a single run across all MPI ranks over all fault injection runs. For example, most of the injected faults will trigger the taint write operation maximum at 12k times.Majority of the cases are under 1k times of taint write operations.

Intuitively, the injection points that resulted in higher tainted memory operations should considered candidates for further hardening via resilience techniques. Armed with fault injection information, researchers can design more resilient algorithms through the analysis of the relationship between different injection points and the propagation of faults.

D. Performance Overhead

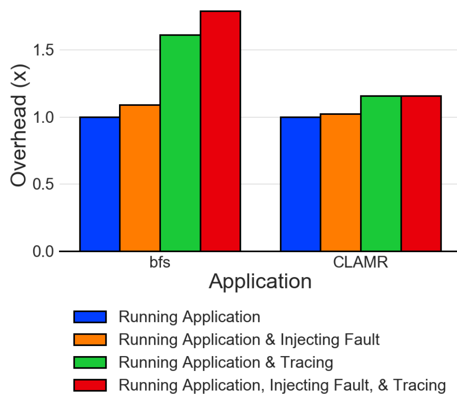

Fig. 10: The performance overhead of Chaser. Results are based on running the application under DECAF++ [14], which imposes only 4% overhead when compared to running the application natively on the host.

We used the Matvec and CLAMR to test the performance overhead of Chaser with and without fault propagation tracing. Since injected faults can change the behavior of an application and cause unfair comparisons of performance, we simply inject the original values into the memory or register instead of flipping any bits. The normalized overhead results are presented in Figure 10.

For CLAMR, the faults are injected into the *fadd* instruction after it has been executed 1000 times. We use *(-n 250 -l 2 -t 20 -i 10)* for our CLAMR arguments. With fault propagation tracing enabled, it takes 103s to finish the execution with and without fault injection (0% overhead). It takes 89s and 91s (2.2% overhead) if fault propagation tracing is disabled. The performance overhead of fault propagation tracing is about 15.7% (103s vs 89s).

V. Related Work

There is a long history of using fault injection technologies to profile the vulnerability of applications, but not for fault propagation study. The work from [5] proposes an error propagation framework which utilizes LLVM instrumentation to inject faults. They then instrument the MPI message "sender" to customize the message structure and format by adding more data corruption related tags and information. The "receiver" can then be notified with incoming corrupted data. Our solution is similar but we add an extra hub (`TaintHub`) to coordinate the information exchanges between the "sender" and "receiver". With `TaintHub`, the message "receiver" does not have to keep parsing each individual incoming message. Instead, only when informed by `TaintHub`, "receiver" will start to record data corruption in the message. Therefore, the overhead of turning on the tracing module in Chaser is much smaller. Still based on LLVM, Li [27] presents the study of error propagation. By dividing the memory into Total Memory(TM), Result Memory(RM) and Output Memory(OM), they traced the errors travels between these three layers at coarse grained level. No syntax is needed, but at the same time, the findings from the error propagation is not useful in guiding the resilience design at the software and algorithm levels. Guo et al. [20] propose FlipTracker, which uses PIN tool to inject errors into the instructions, trace the error propagation, and further analyze the resilience properties in HPC applications. Program codes are partitioned into multiple code regions. By monitoring the input and output of each code regions, the errors are marked and tracked through the execution. FlipTracker provides limited error tracing capability and the accuracy is largely dependent on the choice of the code region. Moreover, FlipTracker cannot support fault injection and analysis in MPI environments as the faults corrupt not only the local memory space but also the other parallel processes.

VI. Conclusions

In this paper, we utilized dynamic binary instrumentation and virtual machine based fault injection to emulate soft errors and error propagation. We proposed Chaser, a fine-grained, accountable, flexible, and efficient soft error fault tracing framework built on top of DECAF as an add-on to PFSEFI. Chaser provides fault propagation tracing and demonstrates the PFSEFI add-on interfaces built with DECAF. We demonstrate that the overhead is small and the tool can be used to study sequential parallel applications to discover how faults propagate through an application. We show that using Chaser, one can evaluate properties of an application such as the relationship of tainted reads and writes as well as how faults spread in a parallel application.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 30,2020 at 22:41:12 UTC from IEEE Xplore. Restrictions apply.

References

[1] Cell-based adaptive mesh refinement. https://github.com/losalamos/CLAMR.

[2] Mpi version of matrix-vectorproduct computation. https://people.sc.fsu.edu/~jburkardt/c_src/mpi/matvec_mpi.c.

[3] The rodinia benchmark suite. https://github.com/pathscale/rodinia/.

[4] Ashraf, R. A., Gioiosa, R., Kestor, G., DeMara, R. F., Cher, C.-Y., and Bose, P. Understanding the propagation of transient errors in hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), ACM, p. 72.

[5] Ashraf, R. A., Gioiosa, R., Kestor, G., DeMara, R. F., Cher, C.-Y., and Bose, P. Understanding the propagation of transient errors in hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 72:1–72:12.

[6] Bagatin, M., Gerardin, S., Paccagnella, A., Andreani, C., Gorini, G., Pietropaolo, A., Platt, S. P., and Frost, C. D. Factors impacting the temperature dependence of soft errors in commercial srams. In *2008 European Conference on Radiation and Its Effects on Components and Systems* (Sept 2008), pp. 100–106.

[7] Becker, H. N., Miyahira, T. F., and Johnston, A. H. Latent damage in cmos devices from single-event latchup. *IEEE Transactions on Nuclear Science 49*, 6 (Dec 2002), 3009–3015.

[8] Calhoun, J., Olson, L., and Snir, M. Flipit: An LLVM based fault injector for HPC. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I* (2014), pp. 547–558.

[9] Cannon, E. H., KleinOsowski, A., Kanj, R., Reinhardt, D. D., and Joshi, R. V. The impact of aging effects and manufacturing variation on sram soft-error rate. *IEEE Transactions on Device and Materials Reliability 8*, 1 (March 2008), 145–152.

[10] Cappello, F., Al, G., Gropp, W., Kale, S., Kramer, B., and Snir, M. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov.: Int. J. 1*, 1 (Apr. 2014), 5–28.

[11] Chandra, V., and Aitken, R. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale cmos. In *2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems* (Oct 2008), pp. 114–122.

[12] Chugg, A. M., Burnell, A. J., Duncan, P. H., Parker, S., and Ward, J. J. The random telegraph signal behavior of intermittently stuck bits in sdrams. *IEEE Transactions on Nuclear Science 56*, 6 (Dec 2009), 3057–3064.

[13] Coe, T., Mathisen, T., Moler, C., and Pratt, V. Computational aspects of the pentium affair. *IEEE Computational Science and Engineering 2*, 1 (Spring 1995), 18–30.

[14] Davanian, A., Qi, Z., Qu, Y., and Yin, H. DECAF++: Elastic Whole-System Dynamic Taint Analysis. *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2019).

[15] Fang, B., Guan, Q., Debardeleben, N., Pattabiraman, K., and Ripeanu, M. Letgo: A lightweight continuous framework for hpc applications under failures. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (8 2017).

[16] Fang, B., Wu, P., Guan, Q., DeBardeleben, N., Monroe, L., Blanchard, S., Chen, Z., Pattabiraman, K., and Ripeanu, M. Sdc is in the eye of the beholder: A survey and preliminary study. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)* (8 2016).

[17] Guan, Q., BeBardeleben, N., Wu, P., Eidenbenz, S., Blanchard, S., Monroe, L., Baseman, E., and Tan, L. Design, use and evaluation of p-fsefi: A parallel soft error fault injection framework for emulating soft errors in parallel applications. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques* (10 2016).

[18] Guan, Q., Debardeleben, N., Blanchard, S., and Fu, S. F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International* (2014), IEEE.

[19] Guan, Q., DeBardeleben, N., Blanchard, S., and Fu, S. Addressing statistical significance of fault injection: Empirical studies of the soft error susceptibility. *International Journal of High Performance Computing and Networking* (4 2017).

[20] Guo, L., Li, D., Laguna, I., and Schulz, M. Fliptracker: Understanding natural error resilience in hpc applications. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2018), pp. 94–107.

[21] Henderson, A., Prakash, A., Yan, L. K., Hu, X., Wang, X., Zhou, R., and Yin, H. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (2014), ACM.

[22] Henderson, A., Yan, L., Hu, X., Prakash, A., Yin, H., and McCamant, S. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering*.

[23] Jagannathan, S., Diggins, Z., Mahatme, N., Loveless, T. D., Bhuva, B. L., Wen, S. J., Wong, R., and Massengill, L. W. Temperature dependence of soft error rate in flip-flop designs. In *2012 IEEE International Reliability Physics Symposium (IRPS)* (April 2012), pp. SE.2.1–SE.2.6.

[24] Kahng, A. B., Nath, S., and Rosing, T. S. On potential design impacts of electromigration awareness. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)* (Jan 2013), pp. 527–532.

[25] Levy, S., Dosanjh, M. G., Bridges, P. G., and Ferreira, K. B. 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* (2013), ACM, pp. 21–26.

[26] Li, D., Vetter, J. S., and Yu, W. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 57.

[27] Li, G., Pattabiraman, K., Cher, C., and Bose, P. Understanding error propagation in gpgpu applications. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2016), pp. 240–251.

[28] Martino, C. D., Kalbarczyk, Z., Iyer, R. K., Baccanico, F., Fullop, J., and Kramer, W. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2014), pp. 610–621.

[29] May, T. C., and Woods, M. H. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices 26*, 1 (Jan 1979), 2–9.

[30] Nicolaidis, M. *Soft Errors in Modern Electronic Systems*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[31] Pellegrini, A., Bertacco, V., and Austin, T. Fault-based attack of rsa authentication. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)* (March 2010), pp. 855–860.

[32] Roberson, M. W. Soft error rates in solder bumped packaging. In *Proceedings. 4th International Symposium on Advanced Packaging Materials Processes, Properties and Interfaces (Cat. No.98EX153)* (Mar 1998), pp. 111–116.

[33] Rossi, D., Omaña, M., Metra, C., and Paccagnella, A. Impact of aging phenomena on soft error susceptibility. In *2011 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems* (Oct 2011), pp. 18–24.

[34] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, may 2010), IEEE Computer Society, pp. 317–331.

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

[36] Snir, M., Wisniewski, R. W., Abraham, J. A., Adve, S. V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications* (2014), 1094342014522573.

[37] Tan, L., DeBardeleben, N., Guan, Q., Blanchard, S., and Lang, M. Using virtualization to quantify power conservation via near-threshold voltage reduction for inherently resilient applications. *Parallel Computing* (2 2018).

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 30,2020 at 22:41:12 UTC from IEEE Xplore. Restrictions apply.

[38] THOMAS, A., AND PATTABIRAMAN, K. Error detector placement for soft computation. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2013), IEEE, pp. 1–12.

[39] THOMAS, A., AND PATTABIRAMAN, K., Eds. *LLFI: An Intermediate Code Level Fault Injector For Soft Computing Applications* (2013), Workshop on Silicon Errors in Logic – System Effects (SELSE).

[40] WEI, J., THOMAS, A., LI, G., AND PATTABIRAMAN, K. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on* (2014), IEEE, pp. 375–382.

[41] WILKINSON, J., AND HARELAND, S. A cautionary tale of soft errors induced by sram packaging materials. *IEEE Transactions on Device and Materials Reliability 5*, 3 (Sept 2005), 428–433.

[42] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 116–127.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 30,2020 at 22:41:12 UTC from IEEE Xplore. Restrictions apply.