
Xunchao Hu*, Qiang Guan†, Heng Yin‡, Nathan DeBardeleben†
*Department of EECS, Syracuse University
Email: xhu31@syr.edu
†Ultrascale Systems Research Center
Los Alamos National Laboratory, Los Alamos, New Mexico
Email: {qguan, ndebard}@lanl.gov
‡Department of CSE, University of California, Riverside
Email:heng@cs.ucr.edu

Abstract—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. 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 DECAF-FSEFI, a fine-grained, accountable, flexible, and efficient soft error fault injection framework built on top of QEMU. DECAF-FSEFI provides just-in-time fault injection, fault propagation trace, and flexible fault injection interfaces. In the case study, we demonstrate the usage of DECAF-FSEFI on fault injection experiments. While armed with so many features, the experiments illustrate that DECAF-FSEFI only introduces 2.48x performance overhead in the worst case and almost 0 overhead in the best case.

Keywords—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 [16], pose a serious threat to the prospect of exascale systems.

In order to study the soft errors, researchers have built different fault injection systems. The current systems rely on either source code instrumentation [3], [17], [18], [6], [15] or dynamic binary instrumentation [7], [13], [12], [19]. They provide the capability of emulating and injecting faults into targeted programs. However, they are designed for different injection tasks and performance overhead is usually high. Also, they do not provide user-friendly interfaces for researchers to develop new fault injectors.

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 DECAF-FSEFI, a fine-grained, accountable, flexible, and efficient soft error fault injection framework built on top of QEMU [4]. DECAF-FSEFI provides just-in-time fault injection, fault propagation trace, and flexible fault injection interfaces. More specifically, DECAF-FSEFI allows to control what application, when and how to inject soft errors with different granularities, without interference to other applications. Researchers can build different fault injectors using the interfaces exported by DECAF-FSEFI.

We implemented and evaluated DECAF-FSEFI against real world programs. The performance overhead ranges from 0 to 2.48x. For Rodinia’s BFS benchmark [2], the performance overhead is 8.8% without fault propagation trace, and 61.1% with fault propagation trace. For CLAMR [1], the performance overhead is 2.2% without fault propagation trace, and 15.7% with fault propagation trace. Because of the exported fault injection interfaces, the development effort of new fault injector is substantially reduced. It only needs about 100LOC to develop a new fault injector. We also conducted the case study on CLAMR to demonstrate the usage of DECAF-FSEFI.

Our main contributions are summarized as follows:

1) We propose DECAF-FSEFI, a fine-grained, accountable, flexible, and efficient soft error fault injection framework built on top of QEMU.
2) We implement and evaluate DECAF-FSEFI’s performance and flexibility.
3) We demonstrate the usage of DECAF-FSEFI with case study on CLAMR.

II. DECAF-FSEFI DESIGN

A. Design Goal

Generally speaking, DECAF-FSEFI is running on top of DECAF (an "enhanced" QEMU). It is designed to achieve the following goals.

a) Fine-grained: The fault injector should be able to inject faults into designated application and instruction at the specific condition. This can allow researchers to construct different complex fault models to study the soft errors’ impact on the behavior of applications.
b) Accountable: The fault injector should be able to trace how the injected faults are propagated within the application. So researchers can analyze how the faults impact the behavior of the application and propose new resilient algorithms.

c) Flexible: The fault injector should be easy to customize and construct different fault models. This allows researchers to design their own fault injection experiments without building the fault injection system from scratch.

d) Efficient: The fault injector should not cause much additional performance overhead since the faults injection experiments usually requires thousands of runs.

B. DECAF-FSEFI Architecture

We propose DECAF-FSEFI, a new fault injection framework, to satisfy the above design goals. Figure 1 illustrates the overall architecture of DECAF-FSEFI. Inside the virtual machine, we run the program and conduct fault injection on it externally via fault injection interfaces. To provide various fault injection capabilities, DECAF-FSEFI gets involved extensively with the dynamic binary translation process, which is detailed in Section III-A. DECAF-FSEFI has the following key components:

  a) Just-in-time Fault Injection: This component is able to inject the fault into target process when the predefined condition is satisfied. Unlike F-SEFI [7], which rewrites the dynamic binary translation process for every instruction to allow fault injection, DECAF-FSEFI only inserts fault injection logic when that instruction is marked as targeted instruction by the user. Since only a tiny portion of targeted process are instrumented, this design significantly reduces the performance overhead as demonstrated in Section IV-A.

  b) Fault Propagation Trace: DECAF-FSEFI traces the fault propagation via the dynamic tainting technique [22]. It leverages DECAF’s bitwise tainting [9], [10] and extends its tainting for floating point instructions. While instruction level trace can record the most complete information about the fault propagation, the performance penalty is not acceptable in practice. In contrast to instruction level trace, DECAF-FSEFI records tainted memory access activity only. This design sacrifices the completeness of fault propagation trace on an acceptable degree while incurs a reasonable performance overhead. Details are discussed in Section III-B.

  c) Flexible Fault Injection Interfaces: The previous works on fault injection [7], [3], [17], [13], [12] target on the specific task or fault model. They are not friendly to the users who are willing to customize their fault injector. To solve this, DECAF-FSEFI exports the fault injection capability as interfaces. Using the exported interfaces, the users can define their fault model via setting the injection location, target instruction, etc. Details are discussed in Section III-C.

C. Sample Fault Injector

Figure 2 presents a sample plugin that injects faults for designated instructions into the target application. When this plugin is loaded into the fault injection framework, its init_plugin function is called to initialize the plugin and returns a pointer to plugin_interface_t, which specifies a new terminal command (inject_fault) defined by the plugin. In addition, the plugin registers one callback (fi_process_creation) for process creation event.

When the user enters the inject_fault command in the terminal, the registered callback do_inject_fault (line 17-31) is called. This callback function sets the targeted program (line 18), targeted instruction (line 22-24), injection condition (line 28), and self-defined fault injector (line 25). Once there is a newly created process, the callback fi_process_creation (line 41-46) is called to check if it is targeted program. If yes, it enables the fault injector (line 43) and registers two callbacks - tainted_mem_write_cb and tainted_mem_write_cb (line 44-45), to log the fault propagation process.

During the runtime of the targeted program, fault_injector (line 8-14) is invoked before the targeted instruction (fadd, mov, etc.) is executed. In this sample, it updates the executed times of the targeted instruction (line 9-10). Once the number of instruction’s executed times reaches the injection condition, the actual fault injection is called to emulate different kinds of soft errors. (line 11-13).

It is worth noting that this sample plugin demonstrates that DECAF-FSEFI satisfies our design goals mentioned in Section II-A. The user can customize the fault injector (flexible) to inject faults into target program and instruction (fine-grained) under customized injection condition (flexible). And since only targeted program and instruction are instrumented, the performance overhead should be lower (efficient) than solutions like F-SEFI [7] which conducts heavy instrumentation on the whole program.

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 to be executed on the host. Figure 3(a) and 3(b) illustrate how fadd is translated into TCG instructions.
b) Placement of Fault Injector: Fault Injector is placed when the target process starts, and the interested instruction is translated. DECAF-FSEFI 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, DECAF-FSEFI flushes the code translation cache and triggers a new round of binary code translation process. During the translation process, the fault injector is injected into interested instructions. Figure 3 illustrates a demo of how the fault injector is injected into fadd. (a) is the original instruction fadd. (b) is the generated TCG IR by QEMU without fault injector. If fadd is labeled as interested instruction, DECAF-FSEFI generates a callback function DECAF_inject_fault and invokes it before fadd is executed. As shown in (c), this is achieved via inserting TCG IR code at the beginning of fadd’s translated TCG IRs.

B. Fault Propagation Trace

Whenever a fault is injected into the memory or register, DECAF-FSEFI taints the corresponding memory or register, and relies on DECAF’s bitwise tainting to trace the fault propagation. Floating point instructions are intensively used on HPC applications, but not supported by DECAF’s bitwise tainting. To support that, we extend DECAF’s tainting implementation by introducing additional shadow registers and tainting rules for floating point instructions. For the simplicity, if the source operand is tainted and there is information flow from source operand to destination operand, we mark all the bytes of destination operand as tainted. While this tainting rule is not as precise as DECAF’s bitwise tainting, it does not sabotage the soundness of DECAF’s tainting system.

DECAF-FSEFI uses callbacks DECAF_WRITE_TAINTMEM_CB and DECAF_READ_TAINTMEM_CB from DECAF to record the fault propagation. These two callbacks are invoked when the targeted program reads/writes the tainted memory. DECAF-FSEFI logs the eip, virtual memory address, physical memory address, tainted value and current value in this memory location for post analysis. We believe that these detailed information provide us a new way to analyze and evaluate soft errors’ impact on applications.

C. Fault Injection Interfaces

DECAF-FSEFI exports fault injection capability as interfaces for users. Those interfaces allow users to customize their fault injectors in the following ways.

Fig. 3: Demo of Fault Injection for fadd
a) what application: Leveraging DECAF’s Virtual Machine Introspection technology, users can specify the targeted application at runtime. Using VMI_CREATEPROC_CB callback, users can get the information of created process, then determine if it is targeted application to inject faults.

b) when to inject: DECAF-FSEFI allows the user to define different injection conditions for every X86 instruction. At runtime, those conditions are checked by the user to determine when to inject faults. For example, to inject a fault to *add* after it is executed 1000 times, the user can define the following struct.

```c
typedef struct InsCounterTrigger {
  // the executed times of the current ins
  uint64_t insCounter;
  // when insCounter hits InjectorTrigger
  // inject the faults.
  uint64_t InjectorTrigger;
  uint32_t BitsToFlip;
} InsCounterTrigger, * pInsCounterTrigger;
```

This struct is associated with *add* Every time *add* is executed, insCounter is increased by 1 and checked to determine if it reaches InjectorTrigger. And If yes, the fault injector is invoked to inject the faults.


c) how to inject: For every X86 instruction, the user can define different fault injectors. DECAF-FSEFI keeps a function pointer to the fault injector for every instruction. If the injection condition is satisfied, the corresponding fault injector is invoked. DECAF-FSEFI also provides functions like CORRUPT_REGISTER and CORRUPT_MEMORY to ease the injection process. These functions can write to any registers and memory locations specified by the user.

IV. Evaluation

We evaluated DECAF-FSEFI with respect to the performance under different configurations, the flexibility of the injection interfaces, and the fine-grained and accountable fault injection using case studies.

The hardware used for all evaluations is a 8-core 3.60GHz Intel Core i7-4790 CPU desktop with 32GB of RAM. The desktop uses Ubuntu 14.04 LTS as its OS. DECAF-FSEFI was executed on this server using an Ubuntu 12.04 LTS image with 512MB of allocated RAM.

A. Performance

We used Rodinia’s BFS benchmark [2] and CLAMR [1] to test the performance overhead of DECAF-FSEFI with and without fault propagation trace. Since the injected faults can change the behavior of application and cause unfair comparison of the performance, we inject original value into the memory or register instead of flipping the bits. The results are presented in Figure 4.

For Rodinia’s BFS benchmark, the faults are injected into *add* instruction when it is executed 1000 times. When fault propagation trace is enabled, it takes 145s and 161s (11% overhead) to finish the execution with and without faults injection, respectively. It takes 90s and 98s (8.8% overhead) if fault propagation trace is disabled. The performance overhead of fault propagation trace is about 61.1% (145s vs 90s).

Table I summarizes the LOC and time consumption used to develop injectors.

<table>
<thead>
<tr>
<th>InjectorName</th>
<th>LOC</th>
<th>Time(hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProbabilisticInjector</td>
<td>97</td>
<td>2</td>
</tr>
<tr>
<td>DeterministicInjector</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>GroupInjector</td>
<td>98</td>
<td>2.5</td>
</tr>
</tbody>
</table>

For CLAMR, the faults are injected into *add* instruction when it is executed 1000 times. We use (-n 250 -i 2 -t 20 -i 10) as CLAMR’s arguments. When fault propagation trace is enabled, it takes 103s and 103s (0% overhead) to finish the execution with and without faults injection, respectively. It takes 89s and 91s (2.2% overhead) if fault propagation trace is disabled. The performance overhead of fault propagation trace is about 15.7% (103s vs 89s).

It is worth noting that the performance overhead of fault injection fluctuates because of the different frequency of the injected instruction in the target application. The worst case is that into every executed instruction the faults are injected. We tested the worst case and the performance overhead is 2.48x. The performance overhead should range from 0 to 2.48x at most based on our experiments. In practice, only a very small portion of the code is inspected to inject the faults. If only 1% of the code is inspected, its performance overhead should be around 2.48% (2.48*1%). This is good enough for practical usage.

B. Flexibility

We evaluated the flexibility of injection interfaces in terms of lines of code (LOC) and time consumption used to develop a new fault injection model. We implemented three fault injectors described in F-SEFI - probabilistic injector, which injects faults at a predefined probability; deterministic injector, which injects faults into target instruction at certain condition; group injector, which injects faults into all floating point instructions. Table I summarizes the LOC and time consumption used to develop them. As shown in the table, it takes about 2 hours and 100 LOC to develop a new fault injector, which is a relatively small workload for researchers. This demonstrates that with DECAF-FSEFI’s injection interfaces, it is much easier to construct different new fault injection models compared with starting from scratch, which may require deep knowledge of the system and tedious engineering work.
TABLE II: Outcome of fault injection for CLAMR with faults into registers

<table>
<thead>
<tr>
<th>Total</th>
<th>Detected Faults</th>
<th>Undetected Faults</th>
<th>Correct Results</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>5195</td>
<td>4349(83.71%)</td>
<td>228(4.38%)</td>
<td>618(11.89%)</td>
<td>846(16.28%)</td>
</tr>
</tbody>
</table>

C. Case Study

We choose CLAMR as the target of our case study. It is executed with the arguments `-n 32 -i 100 -t 2000 -g 100`. Before each run, an injection description is generated to specify that faults will be injected into the operands of instruction $i$ after it is executed $n$ times and the faults are $x$ bits flip of operands. The injection description is generated for all the floating point instructions with randomly generated $n$ and $x$. At runtime, if the injection condition is satisfied, the corresponding faults will be injected.

a) Fault Injection Analysis: In the first set of experiments, we analyze the impact of injecting random transient errors into registers. We run CLAMR 5195 times and when the floating point instruction $i$ is executed $n$ times, inject $x$ bits transient errors into registers used by $i$.

Table II summarizes the observed results. Of 5195 runs, CLAMR detects the injected faults in 4349 (83.71%) runs. DECAF-FSEFI generates 846 (16.28%) undetected faults. These undetected faults are silent data corruption. We further investigated the output of CLAMR with these faults and discovered that 618 (11.89%) runs of CLAMR still produced the correct results while 228 (4.38%) runs produced incorrect results without being detected by CLAMR. We believe that the analysis of the injected faults for the 228 runs is valuable for the community to build a better resilient algorithm to defeat silent data corruption.

b) Fault Propagation Analysis: In this section, we examine how injected faults propagate and their characteristics. In particular, we are interested to discover the propagate characteristics from the perspective of memory operations. In the second set of experiments, we run CLAMR 2973 times and besides registers, also inject $x$ bits transient errors into memories accessed by instruction $i$.

Tainted bytes in the Propagation. Since it is not possible to show all the graphs, we selected 2 fault injection cases from 2973 runs randomly. 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. Figure 5 illustrates the results. 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. CLAMR does not access that memory region after some time. So that the injected faults are not propagated anymore.

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

Memory Operation In the Propagation. DECAF-FSEFI keeps track of two types of memory operations as described in Section III-B. The tainted memory read and tainted memory write operations represent how faults are propagated through memory. We count the number of these two operations for every run of CLAMR and summarizes them in Figure 6 and 7. The figures demonstrate that the involved memory operations’ number vary a lot for different runs of CLAMR. Figure 8 demonstrates the ratio of tainted memory read ratio in memory operations. 1402 (47.1%) of the 2973 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. Intuitively, the injection points that lead to more tainted memory operations should be better guarded with resilient algorithm. Combined with the injected faults information, researchers can build better resilient algorithms via the analysis of the relationship between different injection points and the faults propagation.

V. Future Work

In the future, we would like to continue our work in the following two directions. First, since the current HPC systems use more CPU architectures, we would like to take advantages of QEMU’s multi-platform support and extend DECAF-FSEFI to ARM, PowerPC and other architectures. Second, DECAF-FSEFI can do fine-grained fault injection and fault propagation trace. With these collected data, we would like to conduct research on the new fault models and resilient algorithms.

VI. Related Work

Virtualization provides an infrastructure-independent environment for injecting hardware errors with minimal modification to the system or application. In addition, Winter et al. [20] designed a software-implemented fault injector (SWIFI) based on the Xen Virtual Machine Monitor. Using Xen Hyperecalls, Xen SWIFI can inject faults into the code, memory and registers of Para-Virtualization (PV) and Full-Virtualization (FV) virtual machines. PV intrudes into the original system via modifications to kernel device drivers. Because the injection targets registers (i.e., EIP, ESP, EFL and EAX) that Xen SWIFI does not directly control, it is not possible to be certain that an injection affects the application of interest. SWIFI and the Palacios VMFI are both useful for studying whole systems but for our work we are focused on studying application responses to data corruption in an effort to quantifiably improve its resilience.

Levy et al. [12] propose a virtualization-based framework that injects errors into a guest’s physical address and evaluates fault tolerance technologies for HPC systems (e.g. Palacios [11]). Their Virtual Hardware Fault Injector (VMFI) is only able to inject crash failures and IDE disk failures.

CriticalFault [21] and Relyzer [8] are based on Simics, a commercial simulator. With static pruning and dynamic profiling of the application, instructions are categorized into several classes, i.e., control, data store and address. This process reduces the number of potential fault injection sites by pruning the injection space. Depending on the test scenario, soft errors are injected into different categories, which produce different faulty outputs, e.g., crash or SDC. However, CriticalFault and Relyzer cannot always establish the correlations between the instruction level fault injection and the faulty behaviors, because tracing back from instructions to high-level languages is difficult. Therefore, only coarse-grained injection is available. This makes it challenging to improve the resilience of the original program based on insights learned from the fault injection studies.

GemFI [14], proposed by Parasyris et al, is a fault injection tool based on the full system simulator Gem5 [5]. GemFI can emulate the faults in register files within a process to emulate the behaviors of a micro-component of process under the presence of soft errors. GemFI provides a solution from architectural level to inject the faults and study the propagation of faults, but by bringing in extra cycles in the application, the control of injection is operated inside of the source code to activate the corresponding injection functionality. These extra cycles are also vulnerable to the faults.

VII. Conclusions

In this paper, we utilized dynamic binary instrumentation and virtual machine based fault injection to emulate soft errors. We proposed DECAF-FSEFI, a fine-grained, accountable, flexible, and efficient soft error fault injection framework built on top of QEMU. DECAF-FSEFI provided just-in-time fault injection, fault propagation trace, and flexible fault injection interfaces. In the case study, we demonstrated usage of DECAF-FSEFI on fault injection experiments. While armed with so many features, the experiments showed that DECAF-FSEFI only introduced 2.48x performance overhead in the worst case and almost 0 overhead in the best case.

Acknowledgement

We would like to thank the anonymous reviewers for their constructive comments and suggestions. This work was
performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S.
Department of Energy contract AC52-06NA25396 and U.S.
National Science Foundation Grant #1054605. The publication
has been assigned the LANL identifier LA-UR-17-21172.
Any opinions, findings, and conclusions made in this material
are those of the authors and do not necessarily reflect the
views of the funding agencies.

REFERENCES