Investigating How Software Characteristics Impact the Effectiveness of Automated Software Fault Tolerance

Benjamin James, *Graduate Student Member, IEEE*, Michael Wirthlin, *Senior Member, IEEE*, and Jeffrey Goeders, *Member, IEEE*

***Abstract*— A number of publications have examined automated fault tolerance techniques for software running on commercial off-the-shelf microcontrollers. Recently, we published an automated compiler-based protection tool called COmpiler Assisted Software fault Tolerance (COAST), a tool that automatically inserts dual- or triple-modular redundancy into software programs. In this study, we use COAST to explore how the effectiveness of automated fault protection varies between different benchmarks, tested on an ARM Cortex-A9 platform. Our hypothesis is that certain benchmark characteristics are more likely than others to influence the effectiveness of automated fault protection. Through neutron radiation testing at the Los Alamos Neutron Science Center (LANSCE), we show that cross section improvements vary from 1.6× to 54× across eight benchmark variants. We then explore the characteristics of these benchmarks and investigate how properties of these benchmarks may impact the effectiveness of automated fault protection. Finally, we leverage a novel fault injection platform to isolate two of these benchmark characteristics and validate our hypotheses.**

***Index Terms*— Silent data corruption (SDC), single-event upset (SEU), soft errors, software fault tolerance.**

I. Introduction

RADIATION-HARDENED processors are typically much more expensive and offer lower performance than commercial off-the-shelf (COTS) equivalents. This provides an incentive for finding software-based techniques that increase the fault tolerance of COTS microprocessors, so they can be used in high radiation environments, such as space.

Recent studies have explored different methods for providing programs with fault tolerance through pure software approaches. A common way of providing fault tolerance is through replicating program instructions and/or variables. By inserting one or two replicas of every software operation, faults can be detected and reported or corrected at runtime. There is certainly a performance cost for this type of protection, but this kind of approach has been shown to be successful at reducing the overall error rate and increasing the mean work to failure (MWTF) [1]–[4].

There are different ways of adding duplicated and triplicated instructions, such as modifying the architecture assembly code by hand [1], [2]. As this is not the most ideal solution, other works have studied automated methods of applying mitigation techniques [1], [5]–[10]. However, to the best of our knowledge, none of these works have provided a publicly available, open-source tool that others can use to replicate the work, or to use on new projects.

In July 2018, we released COmpiler Assisted Software Fault Tolerance (COAST), a compiler-based tool that *automatically* applies existing software mitigation techniques to user software. The tool is open-source and publicly available at https://github.com/byuccl/coast. Since COAST is much more automated and flexible than previous work in this area, it is suitable as a tool to explore the effectiveness of software protection on different processing platforms and benchmarks. In recent work, we showed how software protection could be applied to several different architectures (MSP 430, ARM 32-bit and 64-bit, and RISC-V) [11], [12]. Across different target architectures, we saw decreases in cross section ranging from 4× to 106×.

Most of the previous works that explored automated protection provided experimental results on just a couple key benchmarks. The automated nature of our work allows us to explore the effectiveness of software protection on a wide range of benchmarks. In this work, we aim to explore and understand what characteristics of particular benchmarks make them more apt for protection through data and instruction replication. Rather than varying the *platform*, we vary the *program*. The results from testing multiple benchmarks will allow us to better understand how the effectiveness of automated protection changes from benchmark to benchmark and ideally help designers to understand why automated protection may or may not provide substantial reliability improvements.

While we would ideally apply our tool to tens or hundreds of different C programs and build an accurate predictive model, this is not feasible. Limited access to radiation testing facilities combined with the relatively low frequency of errors means

Manuscript received March 19, 2021; accepted April 8, 2021. Date of publication April 14, 2021; date of current version May 20, 2021. This work was supported in part by the Industry/University Cooperative Research Centers (I/UCRC) Program of the National Science Foundation under Grant 1738550 and in part by the Los Alamos Neutron Science Center (LANSCE) under Grant NS-2019-8294.

The authors are with the Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT 84602 USA, and also with the NSF Center for Space, High-Performance, and Resilient Computing (SHREC), Provo, UT 84602 USA (e-mail: b_james@byu.edu; wirthlin@byu.edu; jgoeders@byu.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNS.2021.3073259.

Digital Object Identifier 10.1109/TNS.2021.3073259

we must restrict the number of benchmarks to a small sample set.

The main contributions of this article are:

1) Experimental testing of multiple C programs at the Los Alamos Neutron Science Center (LANSCE). The platform under study, the 32-bit Xilinx ZYNQ ARM Cortex-A9, had eight benchmarks tested on it. Across all these benchmarks, we saw reduction in cross sections from 1.6× to 54×.
2) An analysis of the experimental results, from which we draw insights into benchmark characteristics that may commonly impact software-based fault protection.
3) A fault injection framework that is used to isolate and validate two benchmark properties that impact fault protection effectiveness.

The article is organized as follows: Section II gives more background on related work and the COAST tool. Section III outlines the way in which we tested our benchmarks in a radiation beam and shows the results thereof. Section IV analyzes the radiation test results and benchmark characteristics. Section V details our subsequent fault-injection study, and Section VI provides conclusions.

II. Background

A. Related Work

There have been several works which have investigated adding fault mitigation to software programs. Error Detection by Duplicated Instructions (EDDI) [5] first presented techniques for fine-grained duplication of instructions. These techniques duplicate all instructions and variables while maintaining a single control flow, which requires synchronization of data-flows before any control-flow branching or function calls. This technique is also known as Duplicate With Compare (DWC) and allows for detecting errors at the cost of increased code size and execution time.

Later work introduced SWIFT-R [9], which extended this type of technique to triplication, allowing for not only error detection, but also correction. This is similar to triple modular redundancy (TMR) in hardware and is often referred to by this name in software as well. Software TMR has an even higher cost in code size and execution time than DWC, but with the added benefit of being able to tolerate errors without a reset or rollback.

There have been other works which explored different replication and synchronization rules, as this is an important factor when evaluating tradeoffs between increased run time and fault coverage. Chielle *et al.* [13] presented a set of rules that can be used to guide decisions about replication and synchronization. The COAST tool implements software protection based on some of these rules.

Although there have been many recent works exploring different variations and optimizations on these basic DWC/TMR techniques [1]–[4], [7], [8], [14]–[19], there is no other current tool that offers the automation and flexibility of COAST. These previous works have used hand-modified assembly code, relied on specific architectures or assembly code formats, or leveraged specific processor features to obtain fault tolerance. In addition, none of these works are available as open-source tools, and very few have been tested in an actual high-radiation environment.

```
do:
  ld r0 = i

  r1 = sub r0, 1

  r2 = cmp r1, 0

  br neq r1 do
```

(a)

```
do:
  ld r0 = i
  ld r10 = i_copy
  ld r20 = i_copy2
  r1 = sub r0, 1
  r11 = sub r10, 1
  r21 = sub r20, 1
  r2 = cmp r1, 0
  r12 = cmp r11, 0
  r22 = cmp r21, 0
  r3 = cmp r2, r12
  r4 = select r3, r2, r22
  br neq r4 do
```

(b)

Fig. 1. Code before and after TMR mitigation, from [11]. (a) Original code. (b) TMR code.

B. COAST

Our software protection tool, COAST, automatically adds data-flow protection to arbitrary user programs. The default configuration (and the configuration used in our experiments) is based on the *VAR3* scheme from [6], which is to replicate all compute and memory load/store instructions and to synchronize as necessary before control flow instructions. However, the COAST tool is very configurable; it supports both DWC and TMR modes, as well as changing some of the replication and synchronization rules. Synchronization consists of (for DWC) a comparison of the two data flows, or (for TMR) a voter which determines the correct value based on the three copies. The replication of existing instructions and insertion of synchronization instructions is fully automated as part of the program compilation.

Fig. 1 shows an example of what some assembly code would look like before and after it is run through COAST. The bold text shows the changes made by our compiler pass.

In our past work, we focused on proving the usefulness of COAST [20], or showing its usefulness on multiple target architectures [11]. In this work, we aim to show which *types* of benchmarks can benefit the most from being protected with software techniques.

III. Radiation Test

Radiation testing offers a realistic view into the effectiveness of fault mitigation techniques; however, the high cost and relatively low availability of testing facilities often means that only a few system configurations can be evaluated. In the past work, we observed that fault mitigation was much more effective on some benchmarks than others.

The goal of this test was to evaluate *several* benchmarks on a single platform, to gain an understanding of what software characteristics are present in benchmarks which benefit more from software-based fault protection. We tested eight different benchmarks executing on three identical Xilinx PYNQ development boards. Testing was performed at the LANSCE over the span of 5 days.

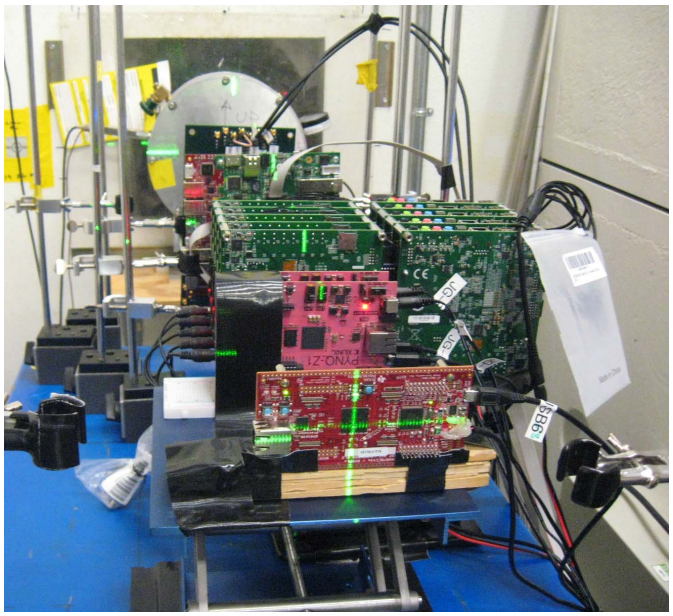

Fig. 2. Neutron beam test setup.

A. Methodology

1) Device Under Test: The device under test (DUT) was a ZYNQ XC7Z020 SoC field-programmable gate array (FPGA), which contains an embedded dual-core 32-bit 667-MHz 28-nm ARM A9 processor. There is a 32-KB instruction cache, 32-KB data cache, and a 512-KB unified cache per core (non-ECC). Only one core of the processor was used in the test, and the FPGA fabric was not utilized or tested. The platform was configured as a bare-metal system, with only essential board support package (BSP) software. The 30-L flight path (Ice House) was used, and the three boards were placed 99, 101, and 106 cm from the neutron detector. These distances were accounted for when determining the fluence received by each board during the experiments. Each board was placed, so the A9's external DRAM chip was outside of the 2" diameter neutron beam. Fig. 2 provides a photograph of the setup, which includes boards from several other experiments.

During the experiment, each benchmark runs some computation operation and then checks the result against a known golden value. In cases where the program output is a small value, such as crc32, the golden value is the exact program output. For other benchmarks, the golden value is a hash of a larger output (the $M \times M$ matrix multiplication benchmark hashes the resultant matrix), or in the case of the *qsort* benchmark, the code ensures the result is sorted. This approach to output validation aims to minimize cases where the golden value, which is not protected, can be corrupted while the program executes.

The benchmarks repeatedly execute the same computation operation, periodically printing a heartbeat message via UART, which is monitored by a computer that is outside the path of the neutron beam. If the computed value does not match the golden value, the program immediately prints an error message to the monitoring computer. In these cases, the controlling system power cycles the board and reprograms the software. Other events can also trigger a reprogramming, including a malformed output, or a heartbeat timeout.

In the experiments described in this article, we employ the TMR option of COAST, which inserts voter operations at branch points in the program. We also enable the `-countErrors` option which allows for enhanced voter code that tracks whether any fault was detected and corrected. While this introduces some extra overhead that would not be used on a deployed system, it provides useful data for our experiment. When faults are detected, they are reported to the controlling system, and they also trigger a power cycle.

B. Benchmarks

We used eight different benchmarks in our test, as outlined below:

- crc32: A 32-bit Cyclic Redundancy Check. This computes the hash of a statically defined table of 32-bit values.
- dijkstra: From the MiBench test suite, this finds the shortest path between a predefined set of nodes.
- matrixMultiply: Matrix multiplication, tested with two sizes: *Fit L1*, where the matrices were sized to all fit in the L1 cache (when triplicated), and *Fit L2*, where they likewise fit in the L2 cache.
- nanojpeg: A simple JPEG decoder[1]; input data is a JPEG image converted to a C array.
- qsort: Sort an array of floating-point numbers. Tested in two configurations: *Library*, where we use the C standard library implementation of `qsort`, which is notably not protected by our tool; and *Custom*, which uses our own code for the sorting kernel, which allows protection to be enabled on the entire algorithm.
- sha256: Computes the SHA-256 hash of a statically defined array.

Each benchmark was compiled and tested using an original unmitigated version and a TMR'd version produced by COAST.

C. Radiation Test Results

The results from our experiment are shown in Table I. The first column lists the benchmark and protection configuration. The next column lists the total Fluence received for each benchmark configuration, which was calculated by correlating timestamps for when each benchmark was running with timestamped flux measurement logs from LANSCE. The next set of columns list the different abnormal statuses encountered during repeated benchmark execution. The *Faults* column lists the number of times the TMR voters in the code detected and corrected a fault. Errors are the number of times the benchmark computed a result which did not match the golden value. A *Hang* was recorded when the benchmark heartbeat stopped responding for a significant amount of time (about 10× the

[1]based on https://keyj.emphy.de/nanojpeg/

TABLE I
NEUTRON BEAM TEST RESULTS

Configuration (Bench, Options)	Fluence (n/cm2)	Faults (TMR Fixed)	Errors (SDC)	Hangs/ Invalid Status	Code Size (KB)		Runtime (ms)		Cross Section (cm2)		MWTF
crc32, Unmit	2.41×10^7	N/A	5	1/1	159	-	936	-	2.08×10^{-7}	-	-
crc32, TMR	2.6×10^8	20	0	11/1	191	↑ **1.201x**	1162	↑ **1.241x**	$^{**}3.84 \times 10^{-9}$	↓ **53.99x**	↑ **43.49x**
dijkstra, Unmit	1.14×10^9	N/A	0	76/2	171	-	478	-	$^{**}8.81 \times 10^{-10}$	-	-
dijkstra, TMR	6.25×10^9	13	0	356/1	191	↑ **1.117x**	2414	↑ **5.05x**	$^{**}1.6 \times 10^{-10}$	↓ **5.51x**	↑ **1.09x**
MxM, Unmit, L2	1.23×10^8	N/A	24	12/0	307	-	212	-	1.95×10^{-7}	-	-
MxM, TMR, L2	4.97×10^8	101	7	47/0	536	↑ **1.75x**	640	↑ **3.02x**	1.41×10^{-8}	↓ **13.85x**	↑ **4.58x**
MxM, Unmit, L1	8.06×10^8	N/A	3	36/0	209	-	1528	-	3.72×10^{-9}	-	-
MxM, TMR, L1	1.14×10^{10}	14	1	519/3	228	↑ **1.09x**	2897	↑ **1.9x**	8.8×10^{-11}	↓ **42.28x**	↑ **22.3x**
nanojpeg, Unmit	5.85×10^9	N/A	47	324/1	187	-	324	-	8.04×10^{-9}	-	-
nanojpeg, TMR	7.27×10^9	119	22	329/1	241	↑ **1.29x**	2503	↑ **7.73x**	3.02×10^{-9}	↓ **2.66x**	↓ **2.91x**
qsortLib, Unmit	8.62×10^9	N/A	2	464/4	290	-	77	-	2.32×10^{-10}	-	-
qsortLib, TMR	6.85×10^9	13	0	333/2	429	↑ **1.48x**	189	↑ **2.45x**	$^{**}1.46 \times 10^{-10}$	↓ **1.59x**	↓ **1.54x**
qsortCustom, Unmit	5.25×10^9	N/A	10	255/0	290	-	277	-	1.9×10^{-9}	-	-
qsortCustom, TMR	2.29×10^{10}	22	0	1119/0	429	↑ **1.48x**	880	↑ **3.18x**	$^{**}4.37 \times 10^{-11}$	↓ **43.61x**	↑ **13.73x**
sha256, Unmit	5.21×10^7	N/A	4	2/241	138	-	14	-	7.68×10^{-8}	-	-
sha256, TMR	2.13×10^8	30	2	10/0	215	↑ **1.56x**	57	↑ **4.07x**	9.37×10^{-9}	↓ **8.19x**	↑ **1.95x**

**No errors observed, so this is calculated given one error (assuming the worst-case, where an error could be observed on the next neutron).

expected heartbeat interval). An *Invalid Status* was recorded any time the UART message from the benchmark did not match the expected regular expression format. Any of these unsuccessful runs triggered a reset of the board.

The columns *Code Size* and *Runtime* are for comparing the overhead required for COAST protection against the original version of the benchmark. *Code Size* is the size of the compiled ELF file, measured in kilobyte.

The column Cross Section measures the error rate according to the following equation:

$$\text{Cross Section} = \frac{\text{Errors (SDC)}}{\text{Fluence}}. \tag{1}$$

The results show that the COAST TMR protection reduces cross section by 1.0× to 54×, indicating that the characteristics of the benchmark significantly influence the effectiveness of the fault mitigation. The cross section results from Table I are summarized in Fig. 3, which shows the cross section for each of the benchmarks with 95% confidence error bars.

Along with cross section, we have the indicator mean work to failure (MWTF) that puts cross section in the context of the run-time overhead. In other words, benchmarks which run longer have more time during which they can be upset. The equation for calculating MWTF is given by the following equation:

$$\begin{aligned}\text{MWTF} &= \frac{\text{amount of work completed}}{\text{number of errors encountered}} \\ &= (\text{raw error rate} \cdot \text{AVF} \cdot \text{execution time})^{-1}. \end{aligned} \tag{2}$$

When taking run time into consideration, it can be seen that while most benchmarks improved in MWTF (1.1×–43×), there were a couple that degraded (*nanojpeg* and *qsortLib*), meaning that the improvement in cross section is not sufficient to overcome the increased fault rate due to the longer runtime.

IV. BENCHMARK ANALYSIS

When we began this test, we had hopes of using the data to construct a model from which to predict the fault coverage of future programs when protected by COAST. However, it quickly become apparent that there are simply too many factors at play to develop an accurate predictive model and doing so would require many more benchmarks and hours of radiation testing, which would be infeasible. However, it was still our intention to gather as much insights into the data we were able to collect, to help learn some lessons for future work, and gather insights that may be helpful for future engineers attempting to apply automated software protection.

The approach we took was to analyze the set of benchmarks we tested in radiation, to determine whether we could find benchmark properties that would correlate with the ***improvement in reliability*** when automated software protection was applied. More specifically, we tried to identify a set of benchmark properties that correlated with the ***factor decrease to cross section*** when automated protection was applied to our benchmark set. To do this, we identified a large set of benchmark characteristics and then determined which subset of these provided the best fit using multiple linear regression.

We recognize that our benchmark set is limited, and with small data sets it may be easy to infer correlation between data when it does not really exist, so we choose to use these identified characteristics to motivate further fault injection testing to validate that these characteristics impact the effectiveness of automated protection. These fault injection results are presented in Section V.

A. Characteristic Set

The set of characteristics we found to be most impactful were the following:

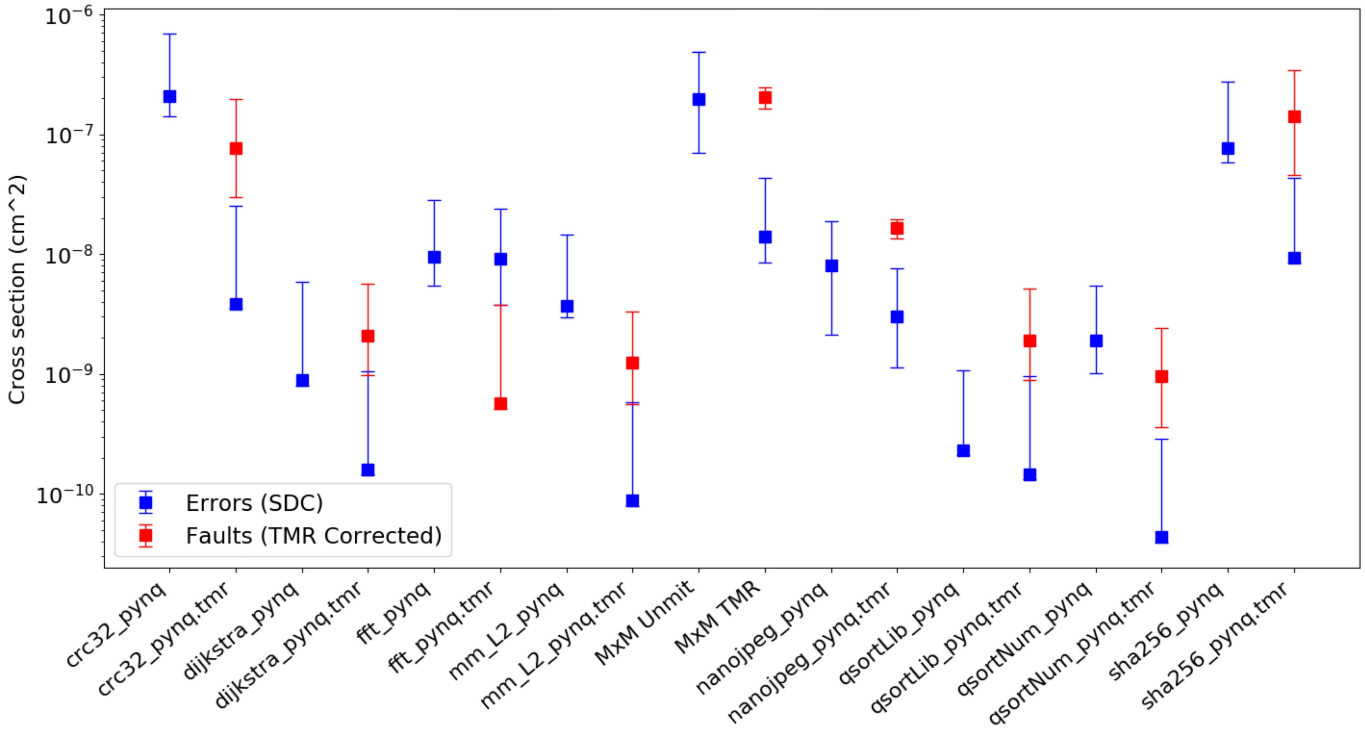

Fig. 3. Benchmark cross sections, 95% confidence interval.

1) **Peak Heap Usage** (in kbytes). *Negatively correlated with effectiveness of fault protection.*
2) **Static Memory Size**. Size of global variables in memory (.data and.bss section counted, in kbytes) *Positively correlated with effectiveness of fault protection.*
3) **Sync Points/s**. How many times a synchronization voter was hit per second of program execution.*Positively correlated with effectiveness of fault protection.*
4) **Fault Tolerance of Unprotected Benchmark**. This characteristic measures the cross section (cm^{-2}) of the unmitigated program, determined from our experimental data. *Unmitigated cross section is positively correlated with effectiveness of fault protection.*

There were several other benchmark characteristics that we examined that either showed no meaningful correlation for our benchmark set or were redundant when considered with other properties. These characteristics included maximum resident set size (memory), read/write ratio, error rate from fault injection on the register file, and all combinations of cache access characteristics for each of the L1 and L2 caches. Although these were not influential for the data set we obtained from the radiation testing, it is certainly possible that some of these characteristics could affect the applicability of our fault mitigation techniques if other benchmarks were used, or if a more thorough regression was performed that included a larger data set or more characteristics.

Furthermore, our set of characteristics is not meant to be an exhaustive list of meaningful benchmark properties. It is very possible that there are other benchmark properties that we failed to identify that may serve as good predictors of the effectiveness automated fault protection. However, we feel that there are still meaningful design lessons to be learned from the characteristics we analyzed.

We now discuss each of these characteristics in greater detail.

B. Peak Heap Usage

Our results indicated that an increase in heap usage negatively correlated with the effectiveness of our automated fault protection. Peak heap usage was obtained using the dynamic analysis tool `massif`, from the `valgrind` tool suite. In our benchmark set, only a few programs used the heap, with `nanojpeg` and `qsort-Library` being the largest users. Since the *Static Memory Size* positively affected cross section performance, we concluded that it was not actually the memory usage itself that was the primary issue, but rather the calls to `malloc`, and the way that it manages heap memory. It seems that the more often `malloc` is called, the less effective COAST is at protecting the code. There are a couple of reasons we expect this is the case: 1) since `malloc` is a library function, it cannot be protected by COAST and 2) even when the memory regions are passed back to the protected code, `malloc`'d regions have special header/footer metadata sections that COAST cannot synchronize. These metadata sections are used by subsequent calls to `malloc` and `free` to determine how each block of memory should be managed. If a fault occurs in any of these special regions, it is likely unrecoverable. Based on this, we believe that it is best to avoid using dynamic memory allocation when wanting to perform software-based fault mitigation. As a second point of reference, the Jet Propulsion Laboratory (JPL) coding standard strongly discourages dynamic memory allocation.

C. Static Memory Size

Static memory usage was determined by inspecting the program executables using the `readelf` utility and observing the sizes of the `.data` and `.bss` sections. The positive correlation indicates that we expect fault tolerance effectiveness to increase as the amount of memory set aside for variables increases. In our test platform, the main memory consists of a large DRAM chip, which is outside of the beam path and naturally more resistant to radiation-based upsets than SRAM [21]. However, data in the processor caches is still highly susceptible to faults and our previous test results indicate that COAST is effective at protecting values that reside in caches [12]. Furthermore, COAST, and other tools like it that apply protection through data replication are inherently designed to protect against data upsets. These approaches do not target upsets that could happen to control-flow elements such as the PC register, return values on the stack, and so on. Since these data-replication approaches are designed to target upsets in data memory, it is not surprising that as programs become more data-heavy, and data sets increase in size, these tools are more effective at protecting against upsets.

To be more precise, although this characteristic is called "static memory size," it is actually the total size of the `.data` and `.bss` sections of the ELF file. These sections represent the majority of the data variables, besides those that are allocated from the `.heap` section, which is described in the previous characteristic. In summary, we believe that the more data variables there are, the better able COAST is at protecting the program.

Some of this behavior may be due to the cache configuration of the platform under test. In our previous radiation experiments [12], we tested the PYNQ board with and without caches enabled. The error rate with the caches enabled was noticeably higher than the error rate with the caches disabled. This means that the errors are more likely to occur in the caches than in main memory or the processor registers. So COAST is helpful in this case because the cross section is so much higher with the caches enabled that there's plenty of room for improvement. It is possible that a different memory hierarchy configuration would have different behavior under a high radiation environment. When application designers are considering using automated protection to provide fault tolerance, they will need to take these architectural features into account when anticipating whether automated fault protection will be effective.

D. Synchronization Points/s

Synchronization points, or voters, are locations in the code where the automated fault protection has inserted operations that inspect the redundant copies of a piece of data and vote on which value should be propagated into the future.

The number of synchronization points encountered during normal execution was determined by modifying our compiler tool to automatically instrument the code such that it would increment a global counter each time a synchronization point was encountered. This number was then divided by the program execution time.

Our results suggest that benchmarks that synchronize more often will see more benefit from our protection techniques. The reason is due to the granularity of replication afforded by COAST. The data flow is replicated at the instruction level, so any data errors could be checked for as often as every three cycles. Although this is somewhat extreme, the general rule is, the sooner the error is detected and corrected, the less chance it has of propagating through the system.

One thing to keep in mind is that it is possible to have too many sync points. Although synchronization allows the TMR'd code to detect/correct errors, it does introduce a potential single point of failure. The code that does the voting is vulnerable to upsets, which represents a failure mode that did not exist in the original, unmitigated version of the code. Analyzing these sync points would be very difficult, as it is not as straightforward as simply measuring the memory usage as in other predictors. The sync points can vary distinctly in quantity, type (data store vs branch comparison), and placement. However, it appears that with the benchmarks tested, we did not exceed the ratio of normal code to synchronization code that would cause it to perform worse.

E. Fault Tolerance of Unprotected Benchmark

Our model is designed to measure improvement to cross section; however, it is important to note that if the benchmark was already inherently fault-tolerant, there may be fewer opportunities for COAST to improve its cross section.

We used the radiation test cross section results from the unprotected benchmarks to determine how naturally susceptible each benchmark was to upsets. Or put another way, the cross section of the unmitigated benchmark provides indication of how likely an upset will manifest as an error in the program output. The larger the cross section, the less fault-tolerant a benchmark is, and thus, there are more opportunities to improve reliability through automated fault protection. On the other hand, if a benchmark has very low cross section, it may already naturally mask faults, and the runtime and memory overheads of imposing automated fault protection may not be worth it.

An example of this is seen in the *quicksort* benchmarks: our golden checking code ensures that the values were sorted correctly; however, it does not actually check that no bits were flipped. Thus, many faults could be naturally masked. While this may not be desirable for an actual sorting benchmark, it would likely arise in other benchmarks, such as machine learning algorithms which have been shown to be somewhat fault tolerant [22].

V. Validating Characteristics Through Fault Injection Testing

In order to validate some of the trends we observed in our radiation testing, we devised a set of experiments to try and isolate a couple of particular benchmark characteristics and then use fault injection testing to determine how the changes impact the effectiveness of our automated protection scheme.

A. Fault Injection Experiments

We created two different fault injection experiments.

1) Matrix Multiply Size: We modified our matrix multiplication benchmark to vary the sizes of the input matrices. These matrices are stored entirely in static memory, so this was done to validate our observation that our automated protection approach is more effective on benchmarks with larger data sizes. In this experiment, we tested four different matrix sizes: 30×30, 75×75, 120×120, and 180×180. In each case, we created an unmitigated version of the benchmark and then a protected version where TMR protection was applied to the code.

In this experiment, we are interested in observing whether the decrease in error rate obtained by TMR protection does indeed improve as the matrix size increases.

2) Inherent Benchmark Fault Tolerance: The other experiment we performed is designed to explore our observation that benchmarks which already mask upsets will not see as much improvement with TMR protection.

In this case, we modified our *qsortLib* benchmark. In the original version, an unsorted array is input into the function, and the quick sort algorithm sorts the values. The golden checking code at the end of the benchmark checks that the values in the array are indeed in sorted order. It is important to recognize that this approach will inherently mask many upsets. This is the case for a couple of reasons. First, if an array entry is sorted into place and then a bit is flipped, it may often still be in sorted order (especially if a lower order bit was flipped). In addition, if a bit is flipped in an array entry before it is sorted into place, the algorithm may still produce a sorted array, even though the final array may be different than the original data set.

We then took this fault-tolerant version of quick sort and modified the golden checking code to instead produce a hash of the sorted values. If this hash did not exactly match a golden hash value, an error is reported. This removes the natural fault tolerance of the algorithm and will instead report an error if any single bit of the array data is modified.

B. Fault Injection Framework

To evaluate these benchmark characteristics, we performed fault injection using our own custom-designed fault injection platform "Platform for ACtive Injection of Faults In a Campaign" (PACIFIC). This framework, which we are publicly releasing as part of our COAST tool (https://github.com/byuccl/coast), approximates radiation testing using randomly injected faults into software while it is executing. Our fault injection tool uses QEMU, a popular machine emulator, to perform fault injections at random locations in memory, and at random points in time during program execution.

While many other fault injection tools exist [23]–[27], our fault injection framework is noteworthy for a few reasons: 1) It leverages custom QEMU plugins, rather than requiring modifications to the QEMU source code like previous tools; 2) it supports fault injection on bare-metal programs; 3) fault injections are granular to the processor-cycle level; and 4) it is specifically designed to allow simulating fault injections in the processor cache.

Testing is done in the form of a fault injection "campaign," where the user specifies 1) the executable to be tested; 2) the section to be targeted; and 3) the number of faults to inject. The campaign supervisor will manage then QEMU and GNU debugger (GDB) instances and inject the specified number of faults, randomly distributed across the bits in the desired section. This is done over multiple runs of the program, where on each execution, the processor is paused and GDB is used to flip a single bit before execution is allowed to continue. If execution of the program does not finish, there is a watchdog which will detect if the program has gone on too long, so it can be forcibly ended. The different possible results are: success, error detected, fault corrected, invalid output, and timeout. Fig. 4 provides a system diagram of our fault injection framework.

In our experiments, we specifically chose to target the processor caches, since the bits in the caches represent a significant target for radiation-induced upsets [28], [29], and our previous radiation testing on the same platform [12] indicated that cache upsets were responsible for a large fraction of our errors.

Our framework is able to specifically target caches by using a QEMU plugin that "subscribes" to execution of all data load and store instructions and will update an internal model of the processor caches. It maintains a model of what addresses in memory are resident in cache at any point of program execution, allowing us to inject faults specifically into these memory addresses.

The QEMU plugin system is also leveraged to enable cycle-accurate injection points. This second QEMU plugin subscribes to instruction execution events, allowing the plugin to monitor each time an instruction is executed. This means that we can randomly inject after any number of instructions and provides much finer control and better distribution than simply sleeping the process for a random amount of time and then pausing execution. This fined-grained approach does add significant runtime overhead and means that thorough fault injection campaigns can take hours or days to complete.

C. Fault Injection Results

The results of the fault injection experiments are provided in Table II.

The variations on the matrix multiplication benchmark confirm our hypothesis that COAST will provide more protection against errors when the program uses more static memory. The error rate decreases for the 30×30, 75×75, 120×120, and 180×180 matrix sizes are $12\times$, $354\times$, $1491\times$, and $2940\times$, respectively. These values show that COAST is highly effective at protecting against upsets in the cache and the effectiveness increases with data size. However, it is important to recognize that fault injection does not perfectly reflect real radiation effects and these results likely overestimate the effectiveness of protection. This is because the fault injection does not capture many upsets that COAST cannot fix, such as upsets in the program counter, control-flow structures, or internal processor state.

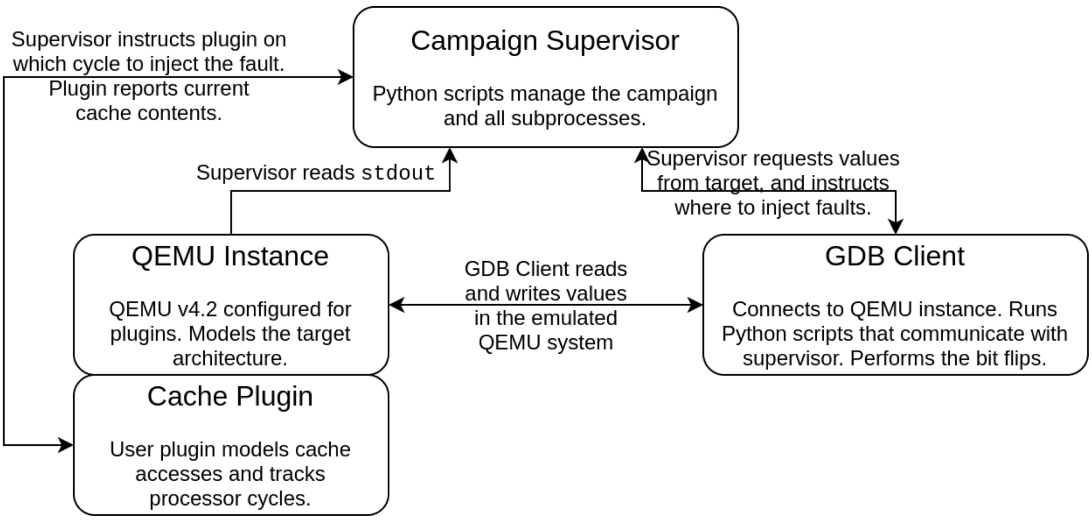

Fig. 4. PACIFIC fault injection framework.

TABLE II
FAULT INJECTION RESULTS

Configuration (Bench, Options)	# Runs	Faults (TMR Fixed)	Errors (SDC)	Hangs/ Invalid Status	Error Rate		MWTF
Matrix Multiply							
30x30, Unmit	4000	0	28	1/0	0.70%	-	-
30x30, TMR	35000	2473	20	126/1	0.06%	↓ **12.25x**	↑ **3.42x**
75x75, Unmit	1000	0	122	0/0	12.20%	-	-
75x75, TMR	58000	14964	20	49/3	0.03%	↓ **353.8x**	↑ **84.81x**
120x120, Unmit	1000	0	257	1/0	25.70%	-	-
120x120, TMR	116000	69276	20	74/2	0.02%	↓ **1490.6x**	↑ **169.57x**
180x180, Unmit	1000	0	490	1/0	49.00%	-	-
180x180, TMR	66000	53745	11	28/0	0.02%	↓ **2940x**	↑ **910.29x**
qsortLib							
Check Sorted, Unmit	2000	0	39	2/0	1.95%	-	-
Check Sorted, TMR	217000	16276	30	279/1	0.01%	↓ **141.1x**	↑ **60.2x**
Check Hash, Unmit	1000	0	162	1/0	16.20%	-	-
Check Hash, TMR	53000	17110	20	106/0	0.04%	↓ **429.3x**	↑ **265.58x**

The experiment on the quicksort algorithm also confirmed our hypothesis regarding algorithms that are inherently fault-tolerant. The TMR protection used by COAST provided a greater benefit on the hash-checking version, which reported an error whenever the simple XOR hash of the data detected a bit mismatch. The version that only checked that numbers were sorted (which could naturally mask upsets) did not achieve the same improvement. The difference was a 141× decrease in error rate versus 429×.

It is important to recognize that the fault-tolerant version still has a lower *raw* error rate, as it is more likely to mask upsets. However, the *improvement* provided by protection is not as significant. This is an important consideration for those looking to protect algorithms that may already naturally mask bit upsets, as the lower effectiveness offered by automated fault protection may not be worth the runtime and memory overheads.

VI. CONCLUSION

In this article, we have presented radiation test results of several benchmarks, tested both in their original form and with automated fault protection applied. The results demonstrate that the effectiveness of automated protection varies greatly from benchmark to benchmark, with cross section improvements ranging from 1.6× to 54×.

We analyzed several properties of the tested benchmarks to determine where correlations exist between the benchmark properties and the effectiveness of fault protection. While our data set is limited, it appears that some important benchmark characteristics include whether static or dynamic memory is used, the size of the data sets, how often replicated data is synchronized, and the inherent fault tolerance of the original algorithm.

Finally, we isolated and validated two of these properties (data size and inherent fault tolerance) through extensive fault

injection, leveraging our custom-designed QEMU-based fault injection framework. In both cases, the results validated what was observed in our original radiation testing.

The results of this work demonstrate how variations in algorithms and software workloads have a large impact on the effectiveness of automated fault tolerance. We hope that this work will spur further exploration into improving automated techniques for software reliability.

REFERENCES

[1] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran, "Software resilience and the effectiveness of software mitigation in microcontrollers," *IEEE Trans. Nucl. Sci.*, vol. 62, no. 6, pp. 2532–2538, Dec. 2015.

[2] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran, "Robust duplication with comparison methods in microcontrollers," *IEEE Trans. Nucl. Sci.*, vol. 64, no. 1, pp. 338–345, Jan. 2017.

[3] E. Chielle *et al.*, "Reliability on ARM processors against soft errors through SIHFT techniques," *IEEE Trans. Nucl. Sci.*, vol. 63, no. 4, pp. 2208–2216, Aug. 2016.

[4] D. S. Khudia, G. Wright, and S. Mahlke, "Efficient soft error protection for commodity embedded microprocessors using profile information," *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 99–108, May 2012.

[5] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 63–75, Mar. 2002.

[6] E. Chielle, R. S. Barth, A. C. Lapolli, and F. L. Kastensmidt, "Configurable tool to protect processors against SEE by software-based detection techniques," in *Proc. 13th Latin Amer. Test Workshop (LATW)*, Apr. 2012, pp. 1–6.

[7] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proc. Int. Symp. Code Gener. Optim.*, vol. 2005, 2005, pp. 243–254.

[8] M. Didehban and A. Shrivastava, "NZDC: A compiler technique for near zero silent data corruption," in *Proc. 53rd Annu. Design Autom. Conf. (DAC)*. New York, NY, USA: ACM, Jun. 2016, pp. 278–283.

[9] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2006, pp. 83–92.

[10] A. Martinez-Alvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F. R. P. Pinto, H. Guzman-Miranda, and M. A. Aguirre, "Compiler-directed soft error mitigation for embedded systems," *IEEE Trans. Dependable Secure Comput.*, vol. 9, no. 2, pp. 159–172, Mar. 2012.

[11] M. Bohman, B. James, M. J. Wirthlin, H. Quinn, and J. Goeders, "Microcontroller compiler-assisted software fault tolerance," *IEEE Trans. Nucl. Sci.*, vol. 66, no. 1, pp. 223–232, Jan. 2019.

[12] B. James, H. Quinn, M. Wirthlin, and J. Goeders, "Applying compiler-automated software fault tolerance to multiple processor platforms," *IEEE Trans. Nucl. Sci.*, vol. 67, no. 1, pp. 321–327, Jan. 2020.

[13] E. Chielle *et al.*, "Reliability on ARM processors against soft errors by a purely software approach," in *Proc. 15th Eur. Conf. Radiat. Effects Compon. Syst. (RADECS)*, vol. 2015, Sep. 2015, pp. 443–447.

[14] E. Chielle, F. L. Kastensmidt, and S. Cuenca-Asensi, "Overhead reduction in data-flow software-based fault tolerance techniques," in *FPGAs and Parallel Architectures for Aerospace Applications: Soft Errors Fault-Tolerant Design*, Cham, Switzerland: Springer, 2015, pp. 279–291.

[15] R. Vemu, S. Gurumurthy, and J. A. Abraham, "ACCE: Automatic correction of control-flow errors," in *Proc. IEEE Int. Test Conf.*, Oct. 2007, pp. 1–10.

[16] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *Proc. The 51st Annu. Design Autom. Conf. Design Autom. Conf. (DAC)*, Jun. 2014, pp. 65–70.

[17] C. Fetzer, U. Schiffel, and M. Süßkraut, "AN-encoding compiler: Building safety-critical systems with commodity hardware," in *Proc. Int. Conf. Comput. Saf., Rel., Secur.*, Berlin, Germany: Springer, 2009, pp. 283–296.

[18] C. Wang, H.-S. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2007, pp. 244–256.

[19] N. Nakka, K. Pattabiraman, and R. Iyer, "Processor-level selective replication," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2007, pp. 544–553.

[20] M. K. Bohman, "Compiler-assisted software fault tolerance for microcontrollers," M.S. thesis, Dept. Elect. Comput. Eng., Brigham Young Univ., Provo, UT, USA, 2018.

[21] T. Semiconductors. (Jan. 5, 2004). *Soft Errors in Electronic Memory—A White Paper*. Accessed: Feb. 12, 2020. [Online]. Available: http://www.tezzaron.com/media/soft_errors_1_1_secure.pdf

[22] F. Libano *et al.*, "Selective hardening for neural networks in FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 66, no. 1, pp. 216–222, Jan. 2019.

[23] W. Chao, F. Zhongchuan, C. Hongsong, and C. Gang, "FSFI: A full system simulator-based fault injection tool," in *Proc. 1st Int. Conf. Instrum., Meas., Comput., Commun. Control*, Oct. 2011, pp. 326–329.

[24] M. Heing-Becker, T. Kamph, and S. Schupp, "Bit-error injection for software developers," in *Proc. Softw. Evol. Week-IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng. (CSMR-WCRE)*, Feb. 2014, pp. 434–439.

[25] E. Carlisle, N. Wulf, J. MacKinnon, and A. George, "DrSEUs: A dynamic robust single-event upset simulator," in *Proc. IEEE Aerosp. Conf.*, Mar. 2016, pp. 3038–3048.

[26] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *Proc. 11th Eur. Dependable Comput. Conf. (EDCC)*, Sep. 2015, pp. 245–255. [Online]. Available: https://ieeexplore.ieee.org/

[27] L. Wanner, S. Elmalaki, L. Lai, P. Gupta, and M. Srivastava, "VarEMU: An emulation testbed for variability-aware software," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Oct. 2013, pp. 224–233.

[28] N. Wulf, G. Cieslewski, A. Gordon-Ross, and A. D. George, "SCIPS: An emulation methodology for fault injection in processor caches," in *Proc. Aerosp. Conf.*, Mar. 2011, pp. 2341–2349.

[29] H. Quinn, "Challenges in testing complex systems," *IEEE Trans. Nucl. Sci.*, vol. 61, no. 2, pp. 766–786, Apr. 2014.