Applying Compiler-Automated Software Fault Tolerance to Multiple Processor Platforms

Benjamin James, Heather Quinn, Michael Wirthlin, and Jeffrey Goeders

***Abstract*— Several recent works have explored the feasibility of using commercial off-the-shelf (COTS) processing systems in radiation-prone environments, such as spacecraft. Typically, this approach requires some form of protection to ensure that the software can tolerate radiation upsets without compromising the system. Our recent work, COmpiler Assisted Software fault Tolerance (COAST), provides automated compiler modification of software programs to insert dual- or triple-modular redundancy. In this article, we extend COAST to support several new processing platforms, including RISC-V and Xilinx, San Jose, CA, USA, SoC-based products. The automated software protection mechanisms are tested for a variety of configurations, altering the benchmark and cache configuration. Across the different configurations, the cross sections were improved by 4× to 106×. In addition, a hardware-mitigation technique is tested using dual-lock-step cores on the Texas Instruments, Dallas, TX, USA, Hercules platform, which is compared with the software-only mitigation approach.**

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

I. Introduction

RECENTLY, there has been a push to enable the use of commercial off-the-shelf (COTS) processing systems in radiation-prone environments, instead of using specialized, radiation-hardened systems. Using COTS systems provides the benefits of significantly reduced cost, and often access to more recent, higher performance technologies. However, using such systems typically require some way to mitigate the risk of silent data corruption (SDC).

One approach is to modify the software to make it more tolerant of upsets; this is typically done by duplicating or triplicating computation in order to detect errors or correct faults. Although this code duplication comes at a price (slower execution and increased memory usage), the approach can be successful at reducing error rate, and netting an increased mean work to failure (MWTF) [1]–[4].

Manuscript received July 6, 2019; revised October 2, 2019 and December 10, 2019; accepted December 11, 2019. Date of publication December 16, 2019; date of current version January 29, 2020. This work was supported by the I/UCRC Program of the National Science Foundation under Grant 1738550 and by the Los Alamos Neutron Science Center which provided beam time under Proposal NS-2018-7895-A.

B. James, M. Wirthlin, and J. Goeders are with the Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT 84602 USA, and also with the National Science Foundation, Center for Space, High-performance, and Resilient Computing (e-mail: b_james@byu.edu; wirthlin@byu.edu; jgoeders@byu.edu).

H. Quinn is with the Los Alamos National Laboratory, ISR-3 Space Data Systems, Los Alamos, NM 87545 USA (e-mail: hquinn@lanl.gov).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNS.2019.2959975

In the past, these replication techniques have been added manually, through the painstaking process of hand modifying the assembly code [1], [2]. Of course, this is not ideal; this labor-intensive process may be prone to errors, may miss the protection of key data elements, or maybe simply too much work for larger programs. While some previous works have investigated automatically applying mitigation techniques [1], [5]–[9], none of these works have provided a publicly available open-source tool that others can use to replicate their work or to use in their own projects.

Last year, we released COmpiler Assisted Software fault Tolerance (COAST), a compiler-based tool that automatically adds fault mitigation to user software through duplication or triplication of instructions and data. This tool is open-source and publicly available at https://github.com/byuccl/coast. We performed experimental testing of COAST on a single microcontroller (a Texas Instruments MSP430) and demonstrated a 4× to 7× increase in MWTF [10].

In this article, we demonstrate a key benefit of our COAST tool, which is that it can be applied to a wide variety of computing platforms. Since the COAST tool is implemented as a machine-independent compiler passes in the LLVM compiler framework, it can be extended to new architectures and platforms with minimal modifications. This has allowed us to perform experimental testing of software-mitigated designs on several different platforms and configurations.

The primary contributions of this article are as follows.

1) Extending the COAST tool to support the Freedom SDK RISC-V toolchain, and the Xilinx SDK toolchain, allowing us to generate software-protected binaries for RISC-V and Xilinx ARM-based SoCs.
2) Experimental testing of running COAST-protected software on multiple software platforms at the Los Alamos Neutron Science Center (LANSCE), Los Alamos, NM, USA. The tested platforms include a 32-bit Freedom RISC-V SoC, 32-bit Xilinx Zynq ARM A9, and 64-bit Xilinx MPSoC ARM A53. Different benchmark and cache configurations were tested and compared. Across the different configurations, the cross sections were reduced by 4× to 106×.

This article is organized as follows. Section II provides background information, describing previous software protection techniques, and a brief overview of the COAST tool. Section III describes how the COAST tool was extended to support new processing platforms. Section IV describes the experimental setup, including details of our test platforms, and

Section V describes the results of the experiments. Section VI provides conclusions.

II. Background

A. Software Protection Through Replication

Tolerating faults in software has been a goal of many different research works over the past couple of decades. Most commonly, these approaches exploit temporal and spatial locality to improve fault tolerance. Variables are stored at separate locations and updated at different times, reducing the probability that an upset will permanently corrupt data and result in an SDC.

Error detection by duplicated instructions [5] was an early work designed to protect the data flow of a program by performing fine-grained duplication of the program execution, duplicating each data-processing instruction. The two instruction flows are synchronized periodically, such as before branches or store operations. Although the program executes with duplicated data, synchronization before branching ensures that there remains only a single-control flow path. This technique also referred to as duplicate with compare (DWC), allows the software to detect errors, at a cost of code size and execution time.

While duplication can detect errors, triplication of data is required to correct errors and continue execution. This was first introduced as SWIFT-R [9] and is analogous to triple-modular redundancy (TMR) in hardware. This, of course, introduces even larger overheads in terms of program size, memory usage, and execution time.

Previous work has also explored how this program replication and synchronization should take place, specifically *what* instructions should be replicated, and *when* to perform synchronization. Chielle *et al.* [3], [11]–[13] present a set of duplication and synchronization rules that have been reused in many subsequent works, including our tool COAST.

B. Software Protection Tools

Although there is substantial previous work that has utilized these software protection techniques [1]–[8], [11]–[21], this article is not easily accessible or not suitable for the devices we are interested in targeting.

Some past work has used hand-modified assembly code rather than an automated process [1], [2], other works target only specific architectures [4], [5], [9], [21], assembly languages [3], [6], [11], [13], processor features [17], [20], or are dependent on multicore systems [18], [19], making them of limited use for future research or commercial projects. Other works have focused on protecting control-flow rather than data-flow [14]–[16]. Most of the works have focused on a server-like environment, targeting high-performance, superscalar processors, rather than embedded systems [4], [5], [7]–[9], [14]–[21]. Of the above-described tools, none of the works provided open-source, publicly available tools. Furthermore, only four of the articles [1]–[3], [12] present results tested in an actual high-radiation environment, the rest only have only simulated upsets with fault injection.

In contrast, our COAST tool [10], which we describe in Section II-C, is designed to be open-source, publicly accessible, and targetable to a wide range of architectures, from simple microcontrollers to more advanced multicore systems. While our tool does not introduce novel protection techniques, it does provide the opportunity for users to test many different platforms, which we demonstrate in the experiments in this article.

```
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 [10]. (a) Original code. (b) TMR code.

```
void __xMR matrix_multiply(int mat1[][N],
    int mat2[][N], int mat_out[][N]) {
  int i, j, k;

  for ( i = 0 ; i < N ; i++ ) {
    for ( j = 0 ; j < N ; j++ ) {
      unsigned long sum = 0;
      for ( k = 0 ; k < N ; k++ )
        sum += sum +
          mat1[i][k]*mat2[k][j];
      mat_out[i][j] = sum;
    }
  }
}
```

Fig. 2. MxM kernel code showing in-code directive for triplication.

C. COAST

Our code protection tool, COAST [10], automatically adds data flow protection to user-provided programs. In the default configuration of the tool, we apply the protection scheme *VAR3*, as detailed in [6], which consists of replicating all compute operations and memory loads/stores while leaving a single set of control flow operations. The COAST tool offers both DWC and TMR protection modes; in the TMR configuration, all compute instructions are triplicated, and synchronization points contain a voter which will determine the correct data based on the three copies. The replication of instructions and the insertion of synchronization points is fully automated as part of the compilation process.

An example of the user's program before and after triplication is provided in Fig. 1. Unmitigated code is shown on the left and our mitigated code on the right. The bold text indicates changes made by our pass.

1) User Configuration: Central to our automated mitigation tool is the fact that the user has high control over the protection passes through the use of both command-line options and source code directives. For example, Fig. 2 shows the core function of our matrix multiplication (MxM) benchmark.

TABLE I
NEUTRON BEAM TEST RESULTS

Configuration (Board/Bench., Options)	Fluence (n/cm^2)	Faults (TMR Fixed)	Errors (SDC)	Hangs	Invalid Status	Code Size (KB)	Run-time (ms)	Cross Section (cm^2)	MWTF
RISC-V									
MxM, Unmitigated	3.9×10^{10}	N/A	80	58	101	104	5.97	2.0×10^{-9}	-
MxM, TMR	1.6×10^{11}	700	3	189	375	110	17.50	1.9×10^{-11}	-
Change						**↑ 1.06x**	**↑ 2.9x**	**↓ 105.3x**	**↑ 35.9x**
SHA256, Unmitigated	6.6×10^{10}	N/A	132	91	146	105	1.34	2.0×10^{-9}	-
SHA256, TMR	8.7×10^{10}	570	23	106	267	122	4.99	2.6×10^{-10}	-
Change						**↑ 1.16x**	**↑ 3.7x**	**↓ 7.7x**	**↑ 2.0x**
ARM A9 (PYNQ-Z1)									
MxM, Unmitigated	3.7×10^{10}	N/A	28	3	9	206	0.211	7.6×10^{-10}	-
MxM, TMR	8.3×10^{10}	163	2	165	11	239	0.666	2.4×10^{-11}	-
Change						**↑ 1.16x**	**↑ 3.2x**	**↓ 31.7x**	**↑ 10.0x**
MxM, Unmit., NoCache	8.3×10^{10}	N/A	13	14	37	206	5.33	1.6×10^{-10}	-
MxM, TMR, NoCache	8.1×10^{10}	26	3	19	12	239	18.80	3.7×10^{-11}	-
Change						**↑ 1.16x**	**↑ 3.5x**	**↓ 4.3x**	**↑ 1.2x**
ARM A53 (Ultra96)									
MxM, Unmitigated	1.8×10^{10}	N/A	0	11	4	240	1.02	$^{**}5.7 \times 10^{-11}$	-
MxM, TMR	2.9×10^{10}	0	0	16	11	256	3.05	$^{**}3.4 \times 10^{-11}$	-
Change						**↑ 1.07x**	**↑ 4.0x**	-	-
MxM, Unmit., NoCache	3.9×10^{10}	N/A	1	19	17	240	54.6	2.6×10^{-11}	-
MxM, TMR, NoCache	4.6×10^{10}	1	1	24	13	256	209.0	2.2×10^{-11}	-
Change						**↑ 1.07x**	**↑ 3.8x**	-	-

**No errors observed, so this is calculated given one error (assuming the worst-case, where an error could be observed in the very next instant of the experiment).

Note the `__xMR` attribute applied to the function, which indicates that TMR protection should be applied to this function.

One important option that COAST provides is the ability to count the number of faults that are corrected by the voting code. This requires a more complex voting code than shown in Fig. 1, and requires updating a global variable each time the voter detected and corrects a discrepancy. While this option negatively affects runtime, and would not be enabled in a deployed system, it is useful for experimental testing, and allows us to collect results on the number of faults corrected, as shown later in Table I.

2) Verifying Correctness: Since COAST works by modifying a user's program, it is critically important that it does not affect the functionality of the program. To ensure this, daily automated testing is performed, which runs COAST on a suite of self-verifying C code benchmarks. This benchmark suite consists of matrix multiply, quicksort, cyclic redundancy check (CRC), advanced encryption standard (AES), fast Fourier transform (FFT) (four variants), LLVM-stress, MiBench (six programs), CHStone (12 programs), and CoreMark (two configurations). We also have unit tests designed to exercise very particular use cases of the protection algorithms. Together these give us over 30 benchmarks to test against, providing a good spread of algorithm types and code sizes.

III. EXTENDING COAST TO NEW PLATFORMS

In order to test COAST's performance on different platforms, it was necessary to extend COAST to support additional toolchains. In this article, we extended the tool to support the Freedom SDK for RISC-V, and the Xilinx SDK for Xilinx's ARM SoC parts. The approach taken for both toolchains was very similar, and thus, we believe that users could easily extend COAST to support additional architectures and boards. While the details of these modifications are beyond the scope of this article, a brief description of the process is included.

Fig. 3 shows the build flow for both the freedom RISC-V and Xilinx ARM platforms. Both of these toolchains utilize the GCC compiler out-of-the-box. However, COAST relies upon the LLVM compiler framework [22], an alternative to GCC. Rather than migrating the build of all platform files to LLVM, which would require significant user effort, our approach is to only compile the user's core program using the LLVM-based COAST compiler. As shown in the diagram, the platform support files, which include the board support package (BSP), continue to be compiled using the existing toolchain compiler. The binaries from both flows are then linked, again using the existing toolchain linker. For the Freedom RISC-V and Xilinx ARM flows, this means that the platform files can continue to be compiled using the GCC toolchain, and the final binary will still be produced by GCC.

This approach minimizes the user effort required to utilize the COAST tool on new platforms. The main responsibility placed on the user is to determine the flags that must be passed to COAST (both the Clang front-end, and the LLVM optimizer) to compile the core

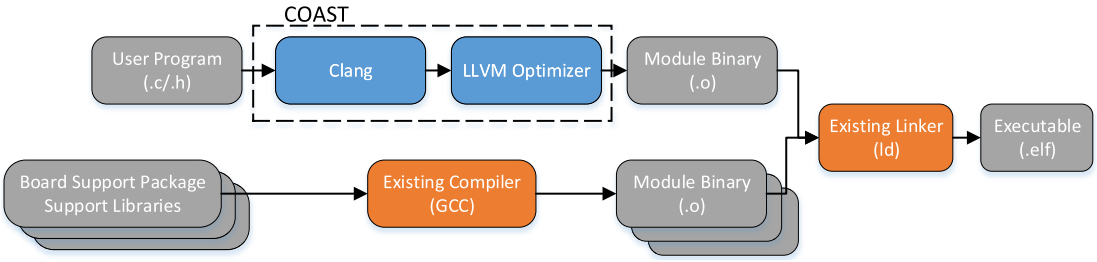

Fig. 3. COAST compiler flow for supporting new toolchains.

user program. For example, for the RISC-V platform these consist of passing "`-m32 -target=riscv32`" to Clang and "`-march=rv32imac -mabi=ilp32 -mcmodel=medany`" to LLVM. Thus, any CPU architecture supported by LLVM should be targetable by COAST, with minimal user effort.

As part of this process, we developed extensible Makefiles for both of these toolchains in order to implement the build process shown in Fig. 3. Users of COAST can support new platforms and boards by making minimal modifications to these Makefiles to connect into their existing toolchains.

IV. Experimental Setup

We tested our COAST software protection tool at LANSCE. The tests were designed to both demonstrate the applicability of our tool to several different platforms, as well as gain radiation sensitivity results for a few popular and emerging platforms.

A. Devices Under Test

The following platforms were tested in our experiment:

1) *SiFive HiFive Board (RISC-V):* This board contains a SiFive Freedom E310, a 32-bit 320 MHz 130 nm RISCV processor, with a 16 kB L1 instruction cache and a 16 kB SRAM scratchpad (non-ECC). In our benchmarks, all data were contained within this scratchpad.
2) *PYNQ-Z1 Board (A9):* This platform contains a Xilinx ZYNQ XC7Z020 FPGA, which contains an embedded 2-core 32-bit 667 MHz 28 nm ARM A9 processor. There is a 32 kB instruction and 32 kB data cache per core (non-ECC). The FPGA fabric was not utilized nor tested.
3) *AVNET Ultra96 Board (A53):* This platform contains a Xilinx Zynq UltraScale + MPSoC ZU3EG FPGA, produced using the TSMC 16FinFET + technology, and contains an embedded 4-core 64-bit 1.5 GHz ARM A53 processor. There is a 32 kB instruction and 32 kB data cache per core (ECC), with a 1 MB L2 cache. The FPGA fabric was not utilized or tested.

All platforms were configured as a bare-metal system, with only essential BSP software. Two RISC-V boards and two PYNQ-Z1 boards were used in the beam test; however, only one Ultra96 board was used, as the JTAG design of the board prevents two boards from being configured from a single computer.

The experiment was conducted in the 30R flight path at LANSCE, as shown in Fig. 4. The boards were spread

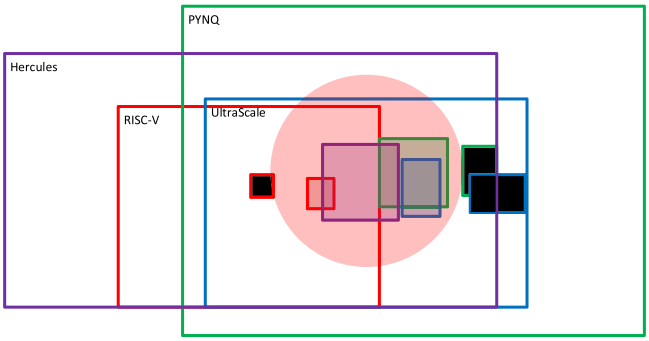

Fig. 4. Cross-section view of board placement in the neutron beam. The 2-in diameter beam cross section is shown as a red circle, boards are shown as the large rectangles, with DUTs shown as the colored shaded small rectangles within the beam. The black-filled rectangles show the placement of chips that we tried to place outside the beam area (power regulator for RISC-V board and DRAM chips for the Xilinx boards).

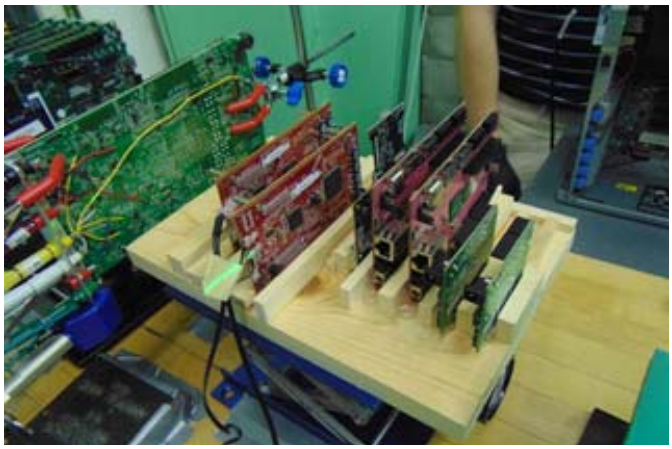

Fig. 5. Picture of LANSCE neutron test. The RISC-V boards are at the far right, the PYNQ-Z1 boards to their left, the Ultra96 next to the left, and then a Texas Instruments Hercules 5F board, followed by boards from other experiments.

from a distance of 81 cm (Hercules 5F board) to 99 cm (2nd RISC-V board) 83 cm from the detector. This results in a 10.7–12.9% attenuation in flux versus the measurements taken at the detector, which was taken into account in our results.

The A9 and A53 platforms contain external dynamic random access memory (DRAM) chips. We did our best to ensure these were located outside the 2-in diameter neutron beam, as shown in Fig. 5; however, the close proximity to the DUT meant that placement was barely outside the beam radius. However, since the DRAM is not as susceptible to neutron-based upsets, we expect that it had a minor impact on our results.

B. Benchmarks

Most configurations were tested using an MxM benchmark, with the core computation, as shown in Fig. 2. The COAST tool was configured to enable TMR on the matrix computation, which resulted in triplication of all operations and all variables, including both input and result matrices. The input matrices contained hard-coded random values. Upon completion of the multiplication, the verification code would compute an XOR of the entire result matrix, and be compared against a hard-coded, predetermined golden value. The size of the matrices was chosen such that they filled the memory in the TMR configuration (scratchpad for the RISC-V platform and the L1 cache for the ARM platforms), and identical matrix sizes were then used for the unmitigated configurations. The size of the square matrix was 19 for the RISC-V and 30 for the A9 and A53.

The SHA256 benchmark (used only on the RISC-V platform due to experiment time constraints) is designed to compute the hash of a large input string. Once again, this was sized as large as possible to fit in the scratchpad memory for the TMR configuration, and the same length, 4000B, was used for the unmitigated version.

Both benchmarks were configured to repeatedly execute the operation as long as the result remained correct, with a periodic heartbeat output over universal asynchronous receiver/transmitter (UART) to an observing computer. If an error was detected, it was output over UART immediately, and the controlling system would automatically power cycle and reprogram the appropriate board. This approach was chosen to prevent any persistent faults. Although the A9 and A53 parts contain dual- and quad-core processors, respectively, the provided results are for the benchmark executing solely on one core.

Our COAST tool provides the option to track any data faults corrected by the TMR voting code. Although this requires extra runtime overhead and would not be utilized in a production system, it allows us to observe how often our TMR system is correcting problems. This option was enabled, and the fault correction data are presented in the results in the following.

C. Configurations

In our experiment, we attempted to demonstrate our COAST tool operating under a number of different conditions. While it would have been beneficial to collect data for an even larger number of configurations, it takes significant beam time to collect enough data to make the results statistically significant.

The RISC-V platform was tested with two benchmarks, MxM and SHA256.

The PYNQ A9 and A53 platforms were tested with only the MxM benchmark, but with the caches enabled and then disabled. Since the MxM benchmark was sized to fit in the L1 cache, the resulting differences should mainly reflect the cross-section difference between the processor core(s) and the L1 cache.

V. Experimental Results

The full results of the neutron beam test are provided in Table I. The second column lists the total fluence received by the board/configuration, and the next four columns provide the number of occurrences of different events. As explained previously, each benchmark would run continuously, outputting a periodic heartbeat via UART, until some irregular status occurred. The *Faults* column lists the number of faults that were detected and corrected by the TMR voters. The *Errors (SDC)* column lists the number of times the calculated result did not match the golden result. *Hangs* occurred when the system stopped responding after some time (roughly $10\times$ the expected heartbeat interval), and required resetting. *Invalid Status* occurred anytime the output message on the UART did not conform to the expected regular expression format. When any of these statuses occurred, the board would be power cycled and reprogrammed.

The next two columns, *Code Size* and *Runtime* report the executable size and executable runtime, respectively. These values capture the cost the user has to pay in order to protect their software program. It should be noted that in the experimental testing, we used the COAST option of counting occurrences of when TMR fixed faults in the data. This adds some additional runtime overhead that would not be necessary for a production environment but does not affect the fault mitigation performance of our tool. By disabling this option, the runtime can be improved by up to 10%, depending on the benchmark and configuration, but still remains about $3\times$ to $4\times$ slower than the original unmitigated code.

The final two columns report the benefit provided by the software protection. The *Cross Section* is calculated as $Errors(SDC)/Fluence$. The final column, *MWTF* reports the *mean work to failure* [21]. Since the software protection causes longer program runtimes, the protected programs have a greater chance of encountering a fault during execution. The MWTF metric captures the relationship between reliability and performance in (1)

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

Fig. 6 provides a plot of the cross-section data with 95% confidence error bars. In this plot, the *Error* data indicates when an incorrect result is computed (SDC error).

One may notice that the sum of the *Faults*, and *Errors* columns for the TMR'd code is much greater than the *Errors* of the unmitigated code. This is expected, as the COAST TMR process triplicates almost all operations and program data. This makes the program roughly $3\times$ more susceptible to single-event upsets. In addition, one should note that the TMR configuration typically was tested for longer durations in the beam and accumulated a greater fluence. This was done to increase the statistical significance of errors, which occur relatively infrequently in the TMR'd version. This relative increase of faults in the TMR'd configuration versus errors in the unmitigated configuration is evidence in Fig. 6,

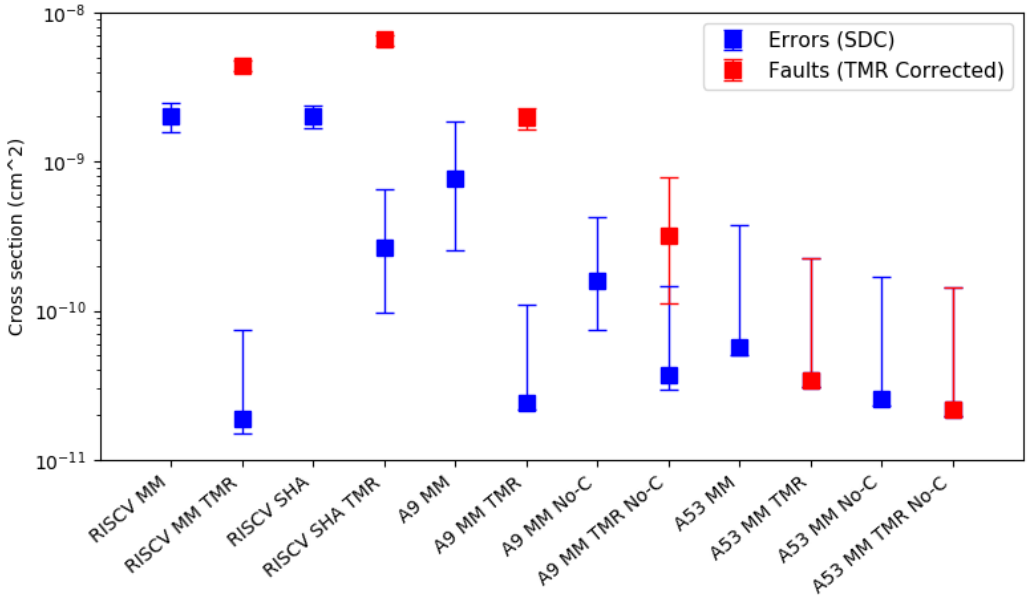

Fig. 6. Cross sections with 95% confidence interval.

where the cross section of faults in the TMR'd configurations (red squares) is noticeable larger than the cross section of errors in the unmitigated designs. Accounting for the 95% confidence intervals shown in Fig. 6, the increase is calculated to be 1.6× to 3.0×, 2.5× to 4.2×, 0.9× to 8.9×, and 0.3× to 10.7×, for the first four configurations in Fig. 6, respectively.

Based on the collected data, there are a number of significant trends. First, similar to our previous radiation testing [10], we have observed that COAST provides a significant improvement to cross section, with a reduction of 4.3× to 105.3×, depending on the platform and configuration. However, this benefit comes at a significant cost: runtime is increased by 2.9× to 4.0×. This increase is expected, as triplication of all instructions, plus voting instructions results in a runtime that is usually >3× (the 2.9× runtime increase is attributed to the compiler vectorizing some of the triplicated instructions, resulting in an increase of less than 3×).

The MWTF metric accounts for this increased runtime, and across the configurations, the MWTF increase ranges from 1.2× to 35.9×.

Since this range is quite dramatic, we provide some analysis of what causes the difference in benefit between the different platforms and configurations.

A. Variation Between Benchmarks

On the RISC-V platform, we tested two different benchmarks and saw significantly different results between the two. Further study after the radiation test revealed a significant difference in the way that COAST treats the two algorithms. By default, COAST uses the same synchronization rules as [3] and [11]–[13] and will synchronize data values on control-flow branches and memory stores.

Depending on the benchmark, there may be a significant variation in the number of synchronization points. For example, the MxM benchmark contains only three synchronization points, as the core compute algorithm is contained in a small portion of code. In contrast, the SHA benchmark, which does not benefit as much from COAST protection, contains 98 synchronization points. These synchronization points, by nature, contain single-points of failure, as triplicated values are aggregated and voted in. With more synchronization in the code, it is not surprising that the TMR protection does not provide as much benefit.

Some of our ongoing work is looking at the current rules COAST uses for insertion of synchronization points, and investigating whether they can be reduced to help alleviate the presence of single points of failure.

Another difference is the nature of the computation and data access patterns between the benchmarks. The SHA benchmark is a hashing algorithm, which sweeps through the input data in a streaming fashion. In contrast, in MxM, each element in the input matrices is accessed multiple times throughout the multiplication process. This may make MxM more sensitive to upsets in the input data memory, compared with say, upsets in the processor pipeline. The results, as discussed in the next sections, suggest that COAST is more effective at correcting upsets in the memory than other locations.

Our future testing will explore a greater variety of benchmarks, whether properties of the benchmark can be studied to predict the effectiveness of the fault mitigation, and possibly what code styles or structures should be employed to construct new programs with the best protection properties.

B. Variation Between Platforms

There are also noticeable differences in the matrix multiply benchmark results running on the RISC-V versus PYNQ platforms. It is the same benchmark, though the matrix sizes differ. There are significant platform differences that might come into play when influencing fault coverage. The primary contributor is likely the fact that on the RISC-V platform, the main memory is located on-chip and is susceptible to radiation upset, whereas the PYNQ and Ultra96 platforms have an external DRAM located outside of the beam radius.

In addition, other factors likely contribute to this difference; for example, the RISC-V processor has more general-purpose registers than the A9 and may have a significantly simpler processor pipeline.

Finally, we were not able to upset the A53 processor beyond a few single events, despite a week of testing. This validates the previous testing that we have performed that suggests this platform is very resistant to single-event effects. This is likely because the L1 data cache has ECC protection, as does the L2 cache. The TLB even has SED protection via parity bits.

C. Variation Between Cache Configuration

On the PYNQ platform, we tested configurations with the caches enabled and disabled. Since the main memory for the PYNQ is located in the off-chip DRAM, we expect that when the caches are disabled, most upsets will be located within the processor pipeline itself (register file, functional units, and so on).

The results show that when caches are disabled, COAST still provides significant cross-section reduction, which demonstrates that COAST can help protect against upsets in the processor core. However, the benefit (4.3× versus 31.7×) is not nearly as large as when upsets are frequently occurring in the memory. When accounting for runtime, the MWTF is only 1.2×.

The results suggest that COAST is not as effective at correcting faults that occur in the processor pipeline. This is likely for two reasons: first, the synchronization voter code, which contains most of the single points of failure, often access recent values that would be located in the register file. Second, when processor core elements are affected, the result can go far beyond simple data corruption: the program counter, stack pointer, TLB, or other special-purpose elements can be affected that cannot be corrected by the simple data-replication provided by COAST.

VI. Conclusion

This article has demonstrated how our COAST tool, which provides automated fault-tolerant protection to user programs, can be effectively deployed on a wide range of processing platforms. The produced software executables are more tolerant of single-event upsets, making COTS platforms more attractive for processing in high radiation environments. The results from the neutron beam test show that COAST provides a significant increase to MWTF across a wide range of platforms and configurations.

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, S. Mahlke, 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, 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, doi: 10.1109/LATW.2012.6261259.

[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. (CGO)*, Mar. 2005, pp. 243–254.

[8] M. Didehban and A. Shrivastava, "nZDC: A compiler technique for near zero silent data corruption," in *Proc. 53nd ACM/EDAC/IEEE Design Automat. Conf. (DAC)*. New York, NY, USA: ACM, Jun. 2016, pp. 1–6, doi: 10.1145/2897937.2898054.

[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] 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.

[11] 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 and Fault-Tolerant Design*. Cham, Switzerland: Springer, 2015, pp. 279–291.

[12] E. Chielle *et al.*, "S-SETA: Selective software-only error-detection technique using assertions," *IEEE Trans. Nucl. Sci.*, vol. 62, no. 6, pp. 3088–3095, Dec. 2015.

[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)*, Sep. 2015, pp. 443–447.

[14] 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, doi: 10.1109/TEST.2007.4437639.

[15] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.

[16] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2014, pp. 1–6, doi: 10.1145/2593069.2593195.

[17] C. Fetzer, U. Schiffel, and M. Süßkraut, "An-encoding compiler: Building safety-critical systems with commodity hardware," in *Proc. 28th Int. Conf. Comput. Saf., Rel., Secur. (SAFECOMP)*, Hamburg, Germany, B. Buth, G. Rabe, and T. Seyfarth, Eds. Berlin, Germany: Springer, Sep. 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 Gen. Optim. (CGO)*, Mar. 2007, pp. 244–256.

[19] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *ACM SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 25–36, 2000.

[20] 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.

[21] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August, "Design and evaluation of hybrid fault-detection systems," in *Proc. 32nd Int. Symp. Comput. Archit.*, Jun. 2005, pp. 148–159.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gen. Optim., Feedback-Directed Runtime Optim.*, 2004, pp. 75–86.