

# 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, COAST (COmpiler Assisted Software fault Tolerance), provides automated compiler modification of software programs to insert dual- or triple-modular redundancy.

In this work we extend COAST to support several new processing platforms, including RISC-V and Xilinx 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 4x–106x. In addition, a hardware-mitigation technique is tested using dual lock-step cores on the Texas Instruments Hercules platform, which is compared to the software-only mitigation approach.

**Index Terms**—Software fault tolerance, single event upset (SEU), silent data corruption (SDC), soft errors.

## 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 performant technologies. However, using such systems typically requires 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, increased memory usage), the approach can be successful at reducing error rate, and netting an increased mean work to failure (MWTF) [1]–[4].

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 protection of key data elements, or may be simply too much

B. James (b\_james@byu.edu), M. Wirthlin (wirthlin@byu.edu), and J. Goeders (jgoeders@byu.edu) are with the Dept. of Electrical and Computer Eng., Brigham Young University, 450 EB BYU, Provo, UT 84602 USA, and the NSF Center for Space, High-performance, and Resilient Computing (SHREC).

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

This work was supported by the IUCRC Program of the National Science Foundation under grant 1738550.

This work was also supported by the Los Alamos Neutron Science Center (LANSCE) which provided beam time under proposal NS-2018-7895-A.

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 COAST (COmpiler Assisted Software fault Tolerance), 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 micro-controller (a Texas Instruments MSP430) and demonstrated a 4–7x increase in mean work to failure (MWTF) [10].

In this work 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 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 work are:

- 1) Extending the COAST tool to support the Freedom SDK RISC-V toolchain, as well as 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). 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 4x–106x.

This paper is organized as follows. Section II provides background information, describing previous software protection techniques, as well as 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 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.

EDDI (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 re-used 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], the work 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, super-scalar 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 papers [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 the next subsection, is designed to be open-source, publicly accessible, and targetable to a wide range of architectures, from simple microcontrollers to more advanced multi-core 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 paper.

### 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 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 Figure 1. Unmitigated code is shown on the left and our mitigated code on the right. The bold text indicates changes made by our pass.

<pre>do:   ld r0 = i    r1 = sub r0 , 1    r2 = cmp r1 , 0    br neq r1 do</pre>	<pre>do:   ld r0 = i   <b>ld r10 = i_copy</b>   <b>ld r20 = i_copy2</b>   r1 = sub r0 , 1   <b>r11 = sub r10, 1</b>   <b>r21 = sub r20, 1</b>   r2 = cmp r1 , 0   <b>r12 = cmp r11, 0</b>   <b>r22 = cmp r21, 0</b>   <b>r3 = cmp r2, r12</b>   <b>r4 = select r3, r2, r22</b>   br neq r4 do</pre>
--	---

(a) Original code

(b) TMR code

Fig. 1: Code before and after TMR mitigation, from [10]

```
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: Matrix Multiplication kernel code showing in-code directive for triplication

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, Figure 2 shows the core function of our matrix multiplication benchmark. Note the `__xMR` attribute applied to the function, which indicates that TMR protection should be applied to this function.

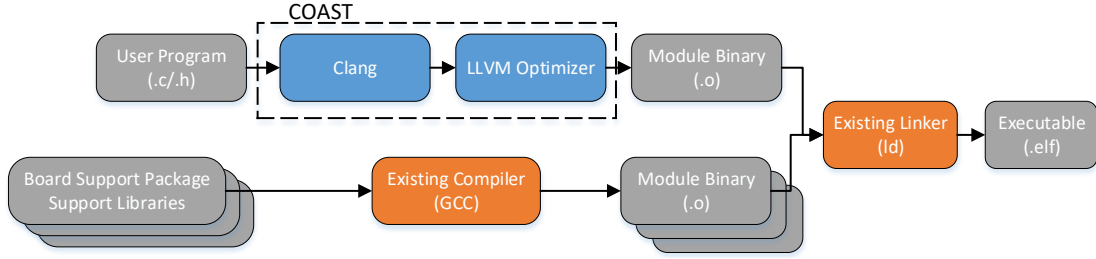


Fig. 3: COAST compiler flow for supporting new toolchains

One important option that COAST provides is the ability to count the number of faults that are corrected by the voting code. This requires more complex voting code than shown in Figure 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, CRC, AES, FFT (4 variants), llvm-stress, MiBench (6 programs), CHStone (12 programs), and CoreMark (2 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 work we extended the tool to support the Freedom SDK for RISC-V, as well as 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 paper, a brief description of the process is included.

Figure 3 illustrates 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 determining the flags that must be passed to COAST (both the Clang front-end, and the LLVM optimizer) to compile the core user program. For example, for the RISC-V platform these consist of passing “-m32 --target=riscv32” to Clang and “-march=rv32imac -mabi=ilp32 -mcmmodel=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 Figure 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 the Los Alamos Neutron Science Center (LANSCC). 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:

**SiFive HiFive Board (RISC-V)** This board contains a SiFive Freedom E310, a 32-bit 320MHz 130nm RISC-V processor, with a 16KB L1 instruction cache and a 16KB SRAM scratchpad (non-ECC). In our benchmarks, all data was contained within this scratchpad.

**PYNQ-Z1 board (A9)** This platform contains a Xilinx ZYNQ XC7Z020 FPGA, which contains an embedded 2-core 32-bit 667MHz 28nm ARM A9 processor. There is a 32 KB instruction and 32 KB data cache per core (non-ECC). The FPGA fabric was not utilized or tested.

**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 1.5GHz ARM A53 processor. There is a 32 KB instruction and 32 KB data cache per core (ECC), with a 1MB L2 cache. The FPGA fabric was not utilized or tested.

All platforms were configured as a bare-metal system, with only essential board support package (BSP) software. Two

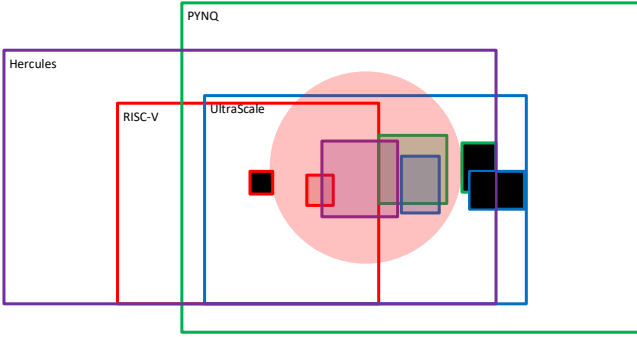


Fig. 4: Cross-section view of board placement in neutron beam. 2” diameter beam cross-section is shown as a red circle, boards are shown as the large rectangles, with DUTs shown as the colored shaded small rectangles within the beam. The black-filled rectangles show placement of chips that we tried to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

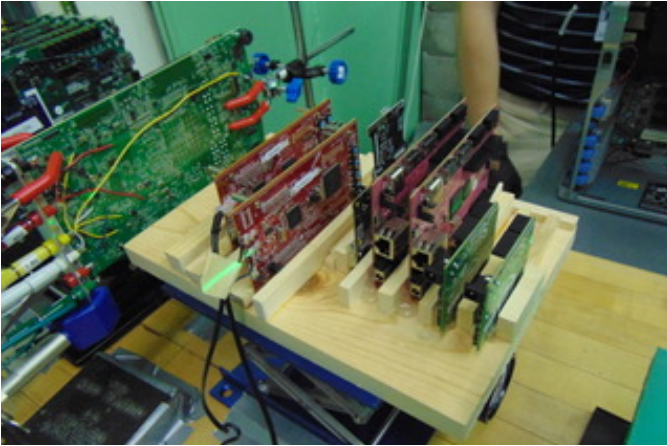


Fig. 5: Picture of LANSCE neutron test. The RISC-V boards are at the far right, the PYNQ-Z1 boards to their left, the Ultra96 next to the left, then a Texas Instruments Hercules 5F board, followed by boards from other experiments.

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 Figure 5. The boards were spread from a distance of 81cm (Hercules 5F board) to 99cm (2nd RISC-V board) 83cm 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 DRAM chips. We did our best to ensure there were located outside the 2” diameter neutron beam, as shown in Figure 4; however, the close proximity to the DUT meant that placement was barely outside of the beam radius. However, since DRAM is not as susceptible to neutron-based upsets, we expect that it had minor impact on our results.

## B. Benchmarks

Most configurations were tested using a matrix multiplication (MxM) benchmark, with the core computation shown in Figure 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 compare against a hard-coded, predetermined golden value. The size of the matrices were chosen such that they filled the memory in the TMR configuration (scratchpad for the RISC-V platform, 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, 4000 bytes, 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 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 run-time 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 is presented in the results below.

## 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 result 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

TABLE I: Neutron beam test results

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

\*\*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).

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 10x 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 status occurred, the board would be power cycled and reprogrammed.

The next two columns, *Code Size* and *Run-time* report the executable size and executable run-time, 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 run-time overhead that would not be necessary in a production environment, but does not affect the fault mitigation performance of our tool. By disabling this option, runtime can be improved by up to 10%, depending on the benchmark and configuration, but still remains about 3–4x 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 Equation 1.

$$MWTF = \frac{\text{amount of work completed}}{\text{number of errors encountered}} \quad (1)$$

$$= (\text{raw error rate} \cdot AVF \cdot \text{execution time})^{-1}$$

Figure 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 3x 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 Figure 6, where

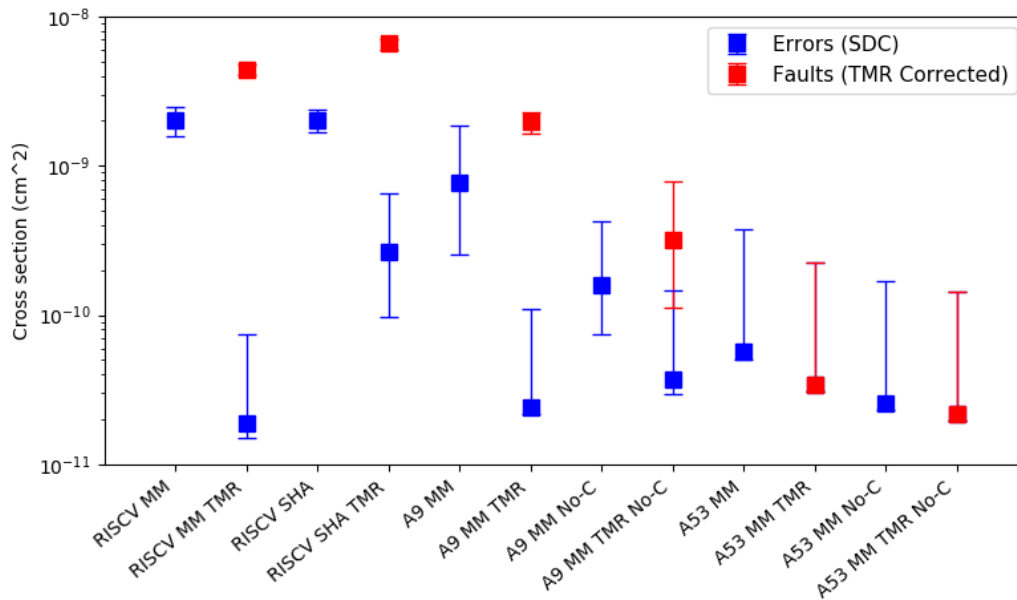


Fig. 6: Cross sections with 95% confidence interval.

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 the figure, the increase is calculated to be 1.6x–3.0x, 2.5x–4.2x, 0.9x–8.9x, and 0.3x–10.7x, for the first four configurations in the figure, respectively.

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

The MWTF metric accounts for this increased run-time, and across the configurations, the MWTF increase ranges from 1.2x to 35.9x.

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], [11]–[13], and will synchronize data values on control-flow branches as well as memory stores.

Depending on the benchmark, there may be a significant variation in the number of synchronization points. For exam-

ple, 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 matrix multiplication, each element in the input matrices are accessed multiple times throughout the multiplication process. This may make MxM more sensitive to upsets in the input data memory, compared to say, upsets in the processor pipeline. The results, as discussed in the next subsections, 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 vs 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 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 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, etc.).

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 (4.3x vs 31.7x) is not nearly as large as when upsets are frequently occurring in the memory. When accounting for run-time, the MWTF is only 1.2x.

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, that contains most of the single points of failure, often access recent values that would be located in the register file. Secondly, 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 work 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 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," in *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, Dec. 2015, pp. 2532–2538.
- [2] —, "Robust Duplication With Comparison Methods in Microcontrollers," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 338–345, Jan. 2017.
- [3] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, J. Tonfat, E. Macchione, F. Aguirre, N. Added, N. Medina, V. Aguiar, M. A. Silveira, L. Ost, R. Reis, S. Cuenca-Asensi, and F. L. Kastensmidt, "Reliability on ARM Processors Against Soft Errors Through SIHFT Techniques," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2208–2216, 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 Transactions on Reliability*, 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 *LATW 2012 - 13th IEEE Latin American Test Workshop*. IEEE, 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 *Proceedings of the 2005 International Symposium on Code Generation and Optimization, CGO 2005*, vol. 2005. IEEE, 2005, pp. 243–254.
- [8] M. Didehban and A. Shrivastava, "nZDC : A Compiler technique for near Zero Silent data Corruption," in *Proceedings of the 2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*. New York, New York, USA: ACM Press, 2016, pp. 1–6, DOI:10.1145/2897937.2898054.
- [9] J. Chang, G. Reis, and D. August, "Automatic Instruction-Level Software-Only Recovery," in *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 2006, pp. 83–92.
- [10] M. Bohman, B. James, M. J. Wirthlin, H. Quinn, and J. Goeders, "Micro-controller compiler-assisted software fault tolerance," *IEEE Transactions on Nuclear Science*, 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: Springer International Publishing, 2015, pp. 279–291.
- [12] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn, "S-SETA: Selective Software-Only Error-Detection Technique Using Assertions," in *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, Dec. 2015, pp. 3088–3095.
- [13] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, F. L. Kastensmidt, R. Reis, and S. Cuenca-Asensi, "Reliability on ARM processors against soft errors by a purely software approach," in *Proceedings of the European Conference on Radiation and its Effects on Components and Systems, RADECS*, vol. 2015-Decem. IEEE, Sep. 2015, pp. 443–447.
- [14] R. Vemu, S. Gurumurthy, and J. A. Abraham, "ACCE: Automatic correction of control-flow errors," in *Proceedings - International Test Conference*. IEEE, 2008, 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 Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, Mar. 2002.
- [16] A. Shrivastava, A. Rhisheekesan, R. Jayapaul, and C. J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2014, pp. 1–6, DOI:10.1145/2593069.2593195.
- [17] C. Fetzer, U. Schiffl, and M. Süßkraut, "An-encoding compiler: Building safety-critical systems with commodity hardware," in *Computer Safety, Reliability, and Security: 28th International Conference, SAFE-COMP 2009, Hamburg, Germany, September 15-18, 2009. Proceedings*, B. Buth, G. Rabe, and T. Seyfarth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 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 *International Symposium on Code Generation and Optimization, CGO 2007*. IEEE, Mar. 2007, pp. 244–256.
- [19] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 25–36, 2000.
- [20] N. Nakka, K. Pattabiraman, and R. Iyer, "Processor-level selective replication," in *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, Jun. 2007, pp. 544–553.
- [21] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *Proceedings - International Symposium on Computer Architecture*. IEEE, 2005, pp. 148–159.

- [22] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, p. 325, 2004.