Software Resilience and the Effectiveness of Software Mitigation in Microcontrollers

Quinn, Heather Marie
Fairbanks, Thomas D.
Tripp, Justin Leonard
Duran Il, Melvin G.

To be published in: IEEE Transactions on Nuclear Science

DOI to publisher's version: 10.1109/TNS.2015.2496342

Permalink to record: http://permalink.lanl.gov/object/view?what=info:lanl-repo/lareport/LA-UR-15-25035

Software Resilience and the Effectiveness of Software Mitigation in Microcontrollers

Heather Quinn, Zachary Baker, Tom Fairbanks, Justin L. Tripp, George Duran

***Abstract*—Commercially available microprocessors could be useful to the space community for non-critical computations. There are many possible components that are smaller, lower-power and less expensive than traditional radiation-hardened microprocessors. Many commercial microprocessors have issues with single-event effects (SEEs), such as single-event upsets (SEUs) and single-event transients (SETs), that can cause the microprocessor to calculate an incorrect result or crash. In this paper we present the Trikaya technique for masking SEUs and SETs through software mitigation techniques. Test results show that this technique can be very effective at masking errors, making it possible to fly these microprocessors for a variety of missions.**

***Index Terms*—soft errors, software, software fault diagnosis, software fault tolerance**

I. Introduction

While most spacecraft rely on radiation-hardened microprocessors for on-board processing, there are many reasons why using commercially available microprocessors might be beneficial. Many commercial microprocessors are cheaper, smaller, faster and take less power than traditional radiation-hardened microprocessors. Not all of the spacecraft's computation requires a radiation-hardened microprocessor, such as configuration, monitoring and background tasks. If the non-mission critical tasks are completed on a secondary microprocessor, the space-grade microprocessor can be devoted to mission critical processing so that more mission data is processed in-situ.

There are a number of commercial microprocessors that could be useful to the space community. In particular, a number of microcontrollers and ARMs have non-volatile memory that the microprocessor uses to *self host*, which means the microprocessor initializes and starts its program once powered. Many of these microprocessors have useful peripherals, such as analog-to-digital converters, comparators and digital-to-analog converters. In previous research, we have presented information on a number of microprocessors that do not have sensitivities to single-event latchup (SEL) and can tolerate a multiple-year mission [1]–[3]. In this paper, we focus on the Texas Instruments (TI) MSP430 microcontrollers and two ARMs.

SEEs can have a complex effect on a microprocessor, especially commercially available microprocessors that are not designed for high radiation environments. During our previous work on digital signal processors (DSPs), we created a taxonomy of effects from SEUs that is useful for microprocessors. This taxonomy, shown in Fig. 1, breaks down how SEUs can corrupt all forms of on-chip memory: caches, SRAM structures, and registers. SEUs can also corrupt memory in control logic and peripherals. SEUs in microcontrollers often have greater consequence in these components, as most memory structures lack error-correcting codes. Likewise, latched SETs could cause corruption in register values, control logic and peripherals. These SEUs and latched SETs can cause data and instruction corruption, which can lead to silent data corruption (SDC), wrong instruction execution, incorrect branches, incorrect jumps, program exceptions, and program crashes. Program crashes are commonly categorized by SEFIs, although the exact root cause of crashes is not well understood.

For the past few years we have been studying whether the effects of SEFIs, SEUs and SETs could be masked through software mitigation [4]. The idea is to alter the software so that corruption of individual data variables and instructions could happen without the calculation being corrupted. Techniques, such as triple modular redundancy (TMR), are generally applicable to most systems, including software. Furthermore, the mitigation process can be automated so that modifications could be made without expert knowledge of the algorithms. The results from these tests using hand-mitigated software have been successful [4]. We are in the process of refining our technique and designing an automated tool, called Trikaya.

This paper is organized as follows. Section II presents other methods for suppressing or masking SEUs or SETs in microprocessors. The Trikaya method is presented in Section III. The experimental setup from testing the Trikaya method are presented in Section IV. Performance results are presented in Section V. Radiation test results are presented in Section VI.

II. Related Work

There are a number of software-based techniques for increasing the robustness of microprocessor-based systems. Many of these are solutions that only work for certain types of algorithms, such as dynamic programming, approximate, matrix, or self-adapting [5]–[18]. There are also methods that work specifically with control flow or datapaths. We will discuss these methods in this section.

One of the most common forms of mitigation is algorithm-based fault tolerance (ABFT). It is common to use ABFT on matrix operations. One of the earliest and most common techniques is to place checksums on the rows and columns on the result matrix [8]. This method adds one column and one row to

H. Quinn, Z. Baker, T. Fairbanks, J. Tripp and G. Duran are with Los Alamos National Laboratory, Los Alamos, NM, 87545 USA, e-mail: hquinn@lanl.gov

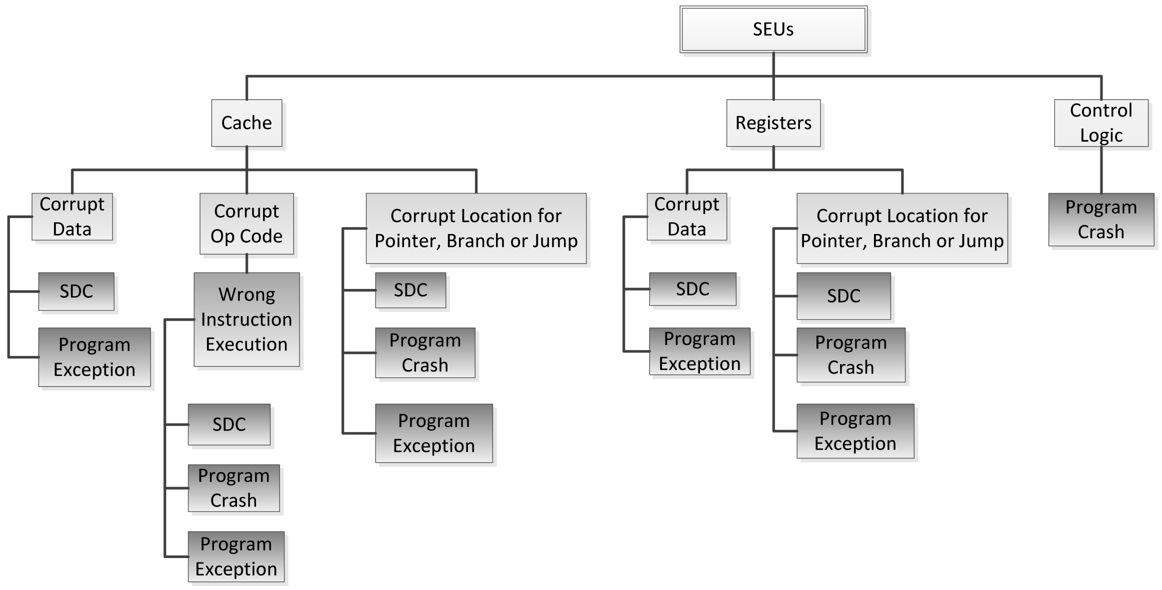

Fig. 1. Taxonomy of how SEUs affect microprocessors and DSPs [4]

result matrix, as well as the necessary computation needed to calculate the checksum for each column and row. Others have created ABFT methods for sparse [9] or dense [10] matrices. ABFT can be applied to many types of algorithms. In [5] the authors provide a technique to increase the robustness of dynamic programming algorithms using outlier detection. A novel system by [11] takes the standard self-adapting software system framework and adapts it for resilience so the system monitors itself for changes. When the system finds non-operational or faulty modules, a plan for working around the faulty portions can be developed and executed by the system. A similar technique is used in [12].

In [6], the authors explore the intrinsic resilience of approximate algorithms, which use heuristic solvers. [6] discusses the possibility of using more resilient approximate solutions to increase system robustness, decrease power consumption and increase execution speed. [7] combines approximate computing with light-weight checking under the assumption that checking for a correct answer is less computationally complex than the actual computation. With this method, the checker determines whether the calculation meets the necessary quality of solution and recalculates as necessary.

There are also a number of methods for mitigating data integrity or control flow errors. Many of these systems are rule-based transformations of the software at very low-levels, such as block statements. For example, in [13], the authors transform the software to have test assertions to make certain the transitions between blocks is valid and setting signatures in the block. These types of transformations are common to prevent control flow errors [15]–[17]. Other transformations can insert low-level duplication with compare (DWC) instructions to protect data variables from SEUs [14]. In [18], the authors warn that there are limitations to these types of software transformation. These authors determined that transformations based on DWC and inverted branch techniques are very effective, but that software signatures are not. Trikaya implements data protections similar to DWC, as the primary problem in these microcontrollers is SEUs in the program's data. Currently, control flow errors are not mitigated as it is a secondary problem.

While the possibility of using self-healing and approximate systems might be useful for spacecrafts, these approaches currently do not meet mission requirements for many systems. Furthermore, many of these techniques have not been tested in radiation environments and have unknown effectiveness. In creating the Trikaya technique, we wanted to create a technique that would work on variety of algorithms without expert knowledge of the algorithm or the data. In this sense, we are interested in determining what the most vulnerable data variables and functions are so that these objects can be protected through software mitigation. We are also designing the tool based on our experience in testing microprocessors with a variety of software codes, including several tests with hand-mitigated codes. To this end, we feel the Trikaya technique can be widely applied to a number of different systems, different algorithms and different environments.

III. Trikaya Technique

The Trikaya technique is based on spatial and temporal TMR. The calculation is made spatially redundant by triplicating the program's variables and is made temporally redundant by triplicating the execution of the mitigated sub-routine. By executing the sub-routine multiple times with separate variables, it is possible to correct SEUs and SETs that cause corrupted calculations. The correction process is handled with majority voting. An overview of the changes to the program structure is shown in Fig. 2. While not shown in Fig. 2, we have also added peripheral scrubbing as part of the process to reduce issues with SEUs in the peripherals. In this section, information about the modification of the software, the automation process and the limitations to the technique are presented.

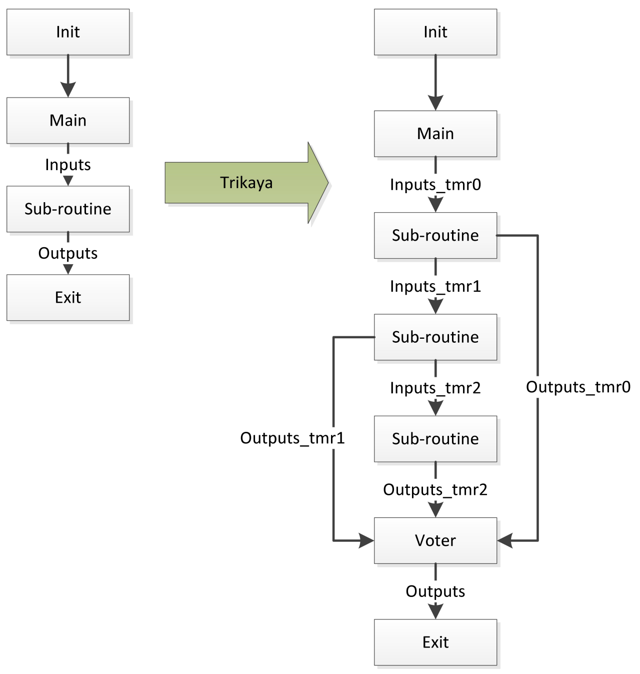

Fig. 2. Trikaya technique for applying full mitigation to a single algorithm in a software program

In these early stages, we are focusing on a coarse-grained and full mitigation of algorithms and proving out these techniques with radiation testing. Coarse-grained modules mitigate entire sub-routines, instead of individual instructions. Full mitigation also focuses on correcting all errors that occur, instead of focusing on the most vulnerable instructions and data variables.

In the future, we plan to move to finer-grained and partial mitigation techniques to reduce the amount of data duplication and execution overhead needed. DWC would be more time and memory efficient than TMR. By providing a wide range of mitigation approaches, the user will be able to determine how much mitigation is needed to meet their mission requirements.

A. Mitigation Process

The automated tool that inserts the data and instruction replication, the software voter and the peripheral scrubber as part of the compilation process. Trikaya uses an open-source compiler, LLVM[1], as a foundation [19]. After the source code is optimized and before writing out machine code, LLVM provides an intermediate representation (IR) of the optimized source code [20]. The IR has several usages, including being a "human readable assembly language representation" that is capable of representing most languages [20]. Because it is a language of its own, it possible to transform the IR by inserting, deleting and altering instructions. We are particularly interested in the ability to transform the IR between the optimization and machine code stage of the compiler so that mitigation can be inserted.

Because we are currently focused on coarse-grained and full mitigation of sub-routines, the changes to the program are minor:

[1]LLVM is not an acronym.

```
int voter (int &val0, int &val1, int &val2) {
   // detect error condition
   if ((val0 != val1) || (val0 != val2) || (val1 != val2)) {
      // find and correct error
      if (val0 == val1) { // val2 is erroneous
         val2 = val0;
         return 0;}
      else if (val0 == val2) { // val1 is erroneous
         val1 = val0;
         return 0;}
      else if (val1 == val2) { // val0 is erroneous
         val0 = val1;
         return 0;}
      else { //3 errors; return error state
         return 1;} } }
```

Fig. 3. Software voter in C

- Insertion of the replicated input and output variables;
- Insertion of the majority voter code;
- Insertion of the peripheral scrubbing code; and
- Insertion of the code to trigger multiple executions of the sub-routine, majority voting and peripheral scrubbing.

Issues with program structure are not addressed at a coarse grain, so it is not necessary to address issues with incorrect data causing problems with looping, branching or jumping.

One of the key program changes is the insertion of the voter. Voting is an important part of the mitigation process, as it protects the calculation. In Trikaya, voting is also handled in software. A C-language version of the software voter is shown in Fig. 3. Software voters are hindered in part by the lack of a three-way comparison. Detecting an error can take five to eight instructions, depending on whether the values are already in registers. Correction is also slow, as the majority value needs to be determined and the minority variable identified. The return codes can be monitored to determine if the voter fails, which then triggers a reset of the system.

We have also found that peripheral scrubbing can be helpful for many microcontrollers and add scrubbers to the mitigated code. For many of the microcontrollers the peripherals are configured through the use of registers. Hundreds of 32-bit wide registers are used to determine the functionality of all the peripherals. All of these registers are SEU sensitive, which can cause the peripherals to perform erratically. As these peripherals control the timers, watchdogs, data input and data output, failures in these peripherals can be disruptive. Furthermore, these failures, like many other types of SEFIs, cannot be self detected. Therefore, it can take external monitoring to determine that the peripheral is malfunctioning. We have found that blind scrubbing of the peripherals by re-initializing the peripherals at regular intervals can minimize how long the disruptions take and automatically reset the peripheral.

B. Limitations of Trikaya

Even with mitigation, some failures could still occur. It is also possible that multiple-independent upsets could accumulate on the component and overwhelm the mitigation technique. Constants can be an issue with the coarse-granularity approach, as these variables are not mitigated. We discuss both of these issues below.

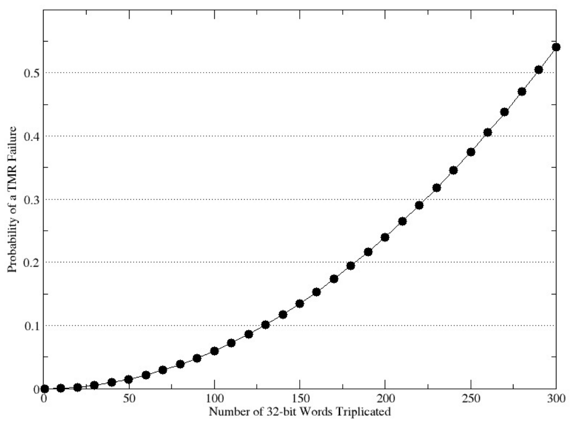

Fig. 4. Probability that two variables have errors caused by two SEUs in the system as a function of the number of words in the variable

1) Multiple-Independent Upsets: Generally speaking, most forms of TMR are only guaranteed to mask a single error in the system. Therefore, an accumulation of SEUs could cause the mitigated software to fail. For Trikaya that means there cannot be more than one SEU in a set of variables. If the same variable has errors in two redundant copies, the voter will fail to correct the errors.

The probability that two upsets can affect multiple copies of the same variable is based on the probability that two redundant variables can be corrupted. If each redundant variable has n words in a memory of m words, the probability of a TMR failure is defined as:

$$P(TMR\,Failure) = \left(\frac{3n}{m}\right)\left(\frac{2n}{m}\right) \tag{1}$$

Fig. 4 shows the probability of TMR failures based on the number of words being triplicated in a memory of 1,000 words. In cases where only a few words are triplicated, the probability of the two upsets being in two of three of the replicas is unlikely. As the number of words increases, the probability of the upsets affecting two replicas increases. As a general rule of thumb, the execution speed of the triplicated algorithm must be faster than two times the upset rate so that it is improbable that two sets of replicated data variables are not affected.

2) Constants: In coarse-grained mitigation, the input and output variables are triplicated, but the variables within the sub-routine are not. The primary advantage of this approach is to minimize the increase in the size of the code's variables. It is possible that SEUs in the algorithm's internal variables could affect the computation of the algorithm. The volatile variables will be reinitialized during the next execution of the algorithm, so the effect is limited in scope and corrected through the majority voting process. SEUs in constants, though, could affect multiple executions, because the constants are not reinitialized.

The probability of the constants having an upset scales by the percentage of the memory dedicated to constants. Therefore, if constants comprise 10% of the memory, then it is possible that 10% of the SEUs will affect constants. One way to avoid these types of failures would be to include these constants as input values to the algorithm so the constants are triplicated. Once triplicated, the constants will follow the same conditional probability for TMR failures. In the case where constants are 10% of the memory, the probability that a constant fails decreases to 6% if replicated.

IV. Experimental Setup

As the tool is currently not ready for testing, we are focused on radiation and performance testing of the hand-mitigated version of the full-mitigation technique. In this section we will provide information about the test setup and tests.

In December 2014 and January 2015, we tested the Trikaya technique on four components: TI flash-based MSP430 (MSP430F2619), TI FeRAM-based MSP430 (MSP430FR5739), TI Tiva, and Xilinx Zynq ARM. All of the radiation results were collected at the Los Alamos Neutron Science Center (LANSCE) Irradiation of Chips and Electronics (ICE) House I and II flight paths. A picture of the December 2014 test setup at LANSCE is shown in Fig 5.

The hardware setup is the same as the setup we used in previous microcontroller tests [1]. The test board is attached to a computer via Universal Serial Bus (USB) cables to a Joint Test Action Group (JTAG) programmer and a Future Technology Devices International (FTDI) device. The FTDI device connnects to the component's Universal Asynchronous Receiver/Transmitter (UART) and converts serial to USB so that status messages can be monitored in real time and stored on the computer for later analysis. The JTAG programmer is connected to the test board and can write directly to the component's SRAM or non-volatile memory. The JTAG programmer for the TI components writes to the non-volatile memory and the components self-boot when power cycled or reset. The JTAG programmer for the Zynq component writes to the SRAM memory and the component has to be programmed when power cycled or reset. In both cases, programming the codes to the memory is controlled by instrumentation software on the computer. The boards are independently biased at nominal voltages, are at nominal temperature and are at a normal incidence to the beam.

The software codes we use for testing the components is based on the benchmark that has been developed for radiation testing [21]. We implement these codes from the benchmark: AES-128 with NIST test vectors, Cache Test, Matrix Multiply (MxM) and Quicksort (Qsort). The size of these applications is purposefully kept as similar as possible. All of these codes are mitigated using the Trikaya technique. For testing purposes, the UART peripheral is scrubbed in both mitigated and unmitigated codes to decrease test interruptions from SEUs in the UART registers. Only a portion of the NIST test vector suite is able to fit into the TI FeRAM's non-volatile memory at a time.

The statistical design of the test has a Latin Squares construct [22]. This construct is useful for tests where one component will be tested with a variety of software programs. The Latin Square methodology transitions the test from one program to the next in an infinite loop so that each program

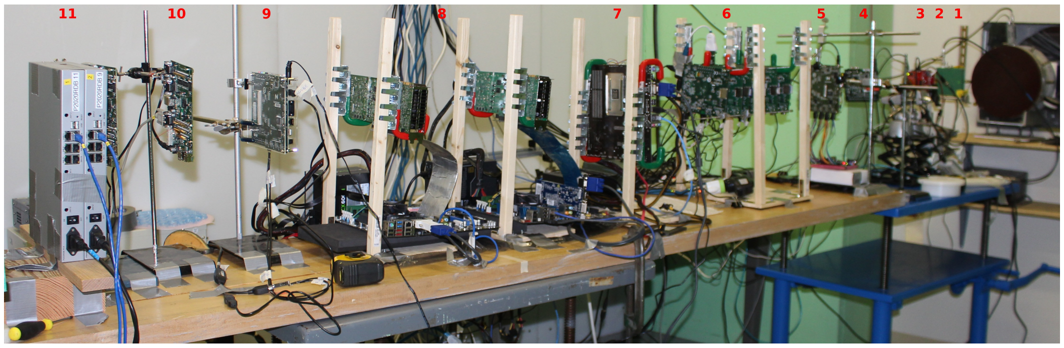

Fig. 5. Picture of LANSCE Test. The tests in this paper are numbered 1-3.

TABLE I
INCREASE IN THE SIZE IN THE VARIABLES, INSTRUCTIONS AND EXECUTION TIME FOR MITIGATED SOFTWARE

Processor	Program	Variables	Instructions	Time
$MSP430F2619$	AES	1.01	1.53	3.29
	Cache	2.74	1.26	8.04
	MxM	1.07	2.40	2.44
	Qsort	1.35	1.44	2.37
$MSP430FR5739$	AES	1.14	1.44	3.28
	Cache	2.40	1.26	79.11
	MxM	2.07	1.42	1.46
	Qsort	2.43	1.31	1.48
$Tiva$	AES	1.00	1.33	2.78
	Cache	2.99	1.24	6.57
	MxM	2.66	1.28	2.03
	Qsort	2.80	1.33	2.92
$Zynq$	AES	1.01	1.04	2.68
	Cache	1.00	1.02	983.22
	MxM	1.05	1.05	1.34
	Qsort	1.05	1.03	1.39

is executed for the same amount of time. The Latin Squares setup is implemented in a python script, which also transfers the codes to the components.

Performance testing was completed on the bench to determine the effect of the mitigation process on execution speed, power consumption and program size. All of the algorithms were measured for changes in overhead. The execution time was determined by the number of completed tests within two minutes. The current consumption was measured on a programmable power supply with the exception of the Zynq. Power consumption could not be measured on the Zynq test fixture. The changes to program sizes were determined by examining the compiled codes.

V. Performance Tests

Table I lists the overhead incurred by applying Trikaya, including changes to the amount of memory used for variables and instructions; and changes to the execution time and power consumption. It is expected that the memory for data could increase by three times and the execution speed could decrease by three times.

The impact of triplicating the input and output variables is less than triplicating all of the variables. In several cases, the triplicated variables are only a small portion of the total memory used for variables. For the AES code, only five 128-bit values are triplicated, so the increase in memory for the triplicated variables is very small for each AES implementation. For the other three codes, though, arrays and matrices are triplicated, which caused the memory used for variables to increase by as much as 2.99 times in the Texas Instruments components. The increase in the memory for triplicated variables in the Zynq is much smaller than the other microcontrollers. The Zynq has more global variables than the other components, and explicitly includes the heap as a global variable.

There is also an impact to the amount of memory used for the code, as redundant executions, the voter and the peripheral scrubber are inserted in the code. In general, most of the text sections, where the instructions are defined, increased by 400 to 1500 words for the Texas Instruments components and by 1400 to 4800 words for the Zynq. When the increase is compared with the base codes, many of these increases are small. For the Texas Instruments components, the increase in the memory for the instructions is 1.26 to 2.40 times. The comparative increase for the Zynq is much smaller than the MSP430s and is between 2–5%. As the base code for the Zynq is 30-40 times larger than the MSP430 codes, the insertion of the extra instructions is less noticeable.

As the mitigated codes are executing the code three times and voting, the execution time of the sub-routine should increase by at least three times. The execution time for the AES code did increase 2.7 to 3.3 times. For MxM and Qsort the increase is between 1.35 to 2.44 times. Cache Test increased by 6-1000 times, depending on the microcontroller. In this case, though, the unmitigated code communicated only error conditions to the computer, whereas the mitigated code sends a constant stream of status messages to the computer. When we equalized the printing, the execution time increased by only 3.5 times. Furthermore, printing is generally consuming a lot of execution time, which masks some of the increased execution time. When most of the printing is removed from MxM on the Tiva, the codes ran 2.27 to 3.0 times faster. When the mitigated and unmitigated codes with limited printing are

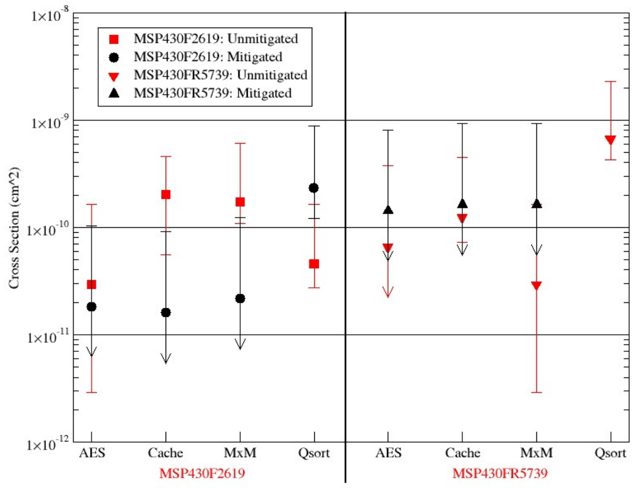

Fig. 6. Cross Sections with 95% confidence intervals for both MSP430s

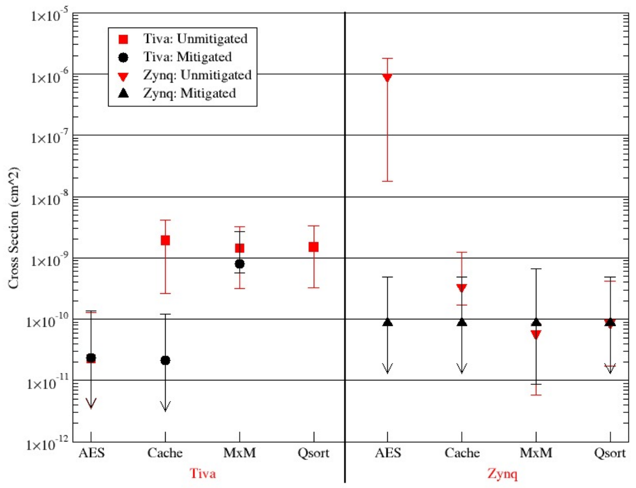

Fig. 7. Cross Sections with 95% confidence intervals for the Tiva and Zynq ARMs

compared, the mitigated code is 2.89 times slower than the unmitigated code. From this analysis, much of the execution time is spent printing output rather than computing.

Finally, we analyzed the power consumption of the Texas Instruments components, which are independently biased through programmable power supplies that allow for current monitoring. For the two MSP430s, there is no difference in the current for any of the codes. For the MSP430FR5739, all codes had a current consumption of 0.0003A. For the MSP430F2619, the current consumption is less than 0.0001A, making it unmeasurable. For the Tiva there are small differences between the codes. The AES code consumed the least current at 0.014A; MxM consumed the most at 0.0019A. The mitigated codes are generally within $\pm$ 0.0001A of the unmitigated codes. It is also clear that the current consumption drops during printing.

In summary, the size of the code, the size of the data and the execution time increased due to the mitigation process. While we expected the data size and the execution time to increase at least three times, many of the codes out-performed our expectations. In many cases, when the size of the base code and data are taken into account, the mitigation process is only affecting a small portion of the code and reduces the overall effect of increasing the code size. Finally, status messages, which are an important part of the testing process, need to be examined, as the printf code can impact the amount of time executing the test code.

VI. Experimental Tests of the Full Mitigation Technique

In this section we discuss the efficacy of the mitigation technique, mitigating SEFIs and the root causes of software failures based on the algorithm. Results from these tests are shown in Figs 6 and 7. It should be noted that many of the unmitigated codes, except AES, were designed to fill the SRAM structures completely. In these cases, the size of the matrices and the arrays used in the unmitigated codes is several times larger than the mitigated codes. For comparison, we scaled the unmitigated codes to be the same size of the mitigated code to measure the change in the execution times and memory usage. We scaled the unmitigated cross sections by the difference in memory size, so that a direct comparison could be made.

A. Sensitivity to Output Errors

The mitigation technique is working on many fronts. The cross sections in Figs 6 and 7 are a measurement of the failures in both the unmitigated and mitigated software for the four codes tested. For the mitigated software, this measurement is of the failures that are not correctable. Many of the mitigated codes had no failures. In these cases, the data point is placed at $\frac{1}{fluence}$ and the lower error bar is zero, which is represented as a downward arrow. It can be difficult to compare these data sets due to the number of null data points. In some of these cases, such as the Cache Test and MxM programs on the MSP430F2619 (Fig. 6), the null data point for the mitigated software is an order of magnitude smaller than the measured cross section for the unmitigated software. In one case, AES on the Zynq (Fig. 7), the mitigated version of the software decreased the cross section by four orders of magnitude. For most of the parts and programs, there is an improvement when Trikaya is applied to the program. In the mitigated MxM code nearly two million errors in the result matrix are masked and only 41 failures were observed. The Cache Test has been difficult to mitigate in the past, because it spans all available SRAM. With the Trikaya technique the mitigated version for all of the components completed testing with no errors.

In some cases, such as the MSP430FR5739 in Fig. 6 and the Zynq in Fig. 7, the null data points for the mitigated software cross sections are plotted at larger values than the unmitigated software. While it is possible that the cross section is larger than the unmitigated software, it is also possible that more testing would have provided a null data point that is smaller than the unmitigated software.

B. Crashes

Previously, it seemed as if mitigating the software helped reduce the SEFI cross section on DSPs [4], although we could never explain these results. These results are not reproducible on microcontrollers. We did find that our test methodology did decrease crashing for the Zynq component, though. The Latin Square test methodology causes the systems to have the code refreshed and the component to be reset at regular intervals. We have found this process causes the Zynq to crash less frequently, possibly because SEUs cannot accumulate in critical code for too long. The Xilinx ARM is very sensitive to crashes, which might be because it uses SRAM to store the code and the data. For this component the SEFI states make it difficult to test and it frequently takes hands-on intervention to reset the component after a crash. The SEFI cross section for the Zynq ARM is approximately $2 \times 10^{-9} cm^2$, which means that crashes occur every five to ten minutes at LANSCE. Using the Latin Square test methodology, the Xilinx ARM is able to operate without intervention for 10-15 hours, which is a decrease of the SEFI cross section by two orders of magnitude.

While not a perfect solution, the process of resetting the system to a known good state decreases the chance of unanticipated crashes and allows the users to determine when to schedule the resets. Without more information about what is causing the crashes, there is currently no better solution for mitigating the problem.

C. Root Cause Analysis

Most of the calculation failures observed are from SEUs in the data variables. Three of the codes, Cache Test, Qsort and MxM have memory-bound calculations, where the instructions are dominated by loads and stores. Qsort is heavily dependent on memory movement, and the only logic instructions compare values. On the MSP430F2619 there is one failure that heavily skewed the Qsort results. In one test run 72 errors occurred at one time during a single execution of the code. Many of the errors were corrected by Trikaya, but 12 were not correctable. This problem did not repeat in the Qsort test on this component or any of the other components.

The AES code is the only logic-bound code that we tested. The amount of input and output data is only five 128-bit variables and the code is dominated on logic transformations of these variables. Because this code has a large ratio of logic instructions to memory instructions, it highlights SETs. Only two components had errors in AES. In the flash-based MSP430 there are 16 errors in one component and all of those errors occurred in the same computation. The Xilinx ARM had many failures in the AES code. All of these errors are related to the test vectors being stored in SRAM. Because there is no SEU-resistant memory on chip to store the test vectors, the test vectors have a large SEU cross section. It should be noted that mitigation of the test vectors with TMR is able to suppress all of these SEUs and no errors are observed on this component for the mitigated AES code.

VII. Conclusions

Commercially available microprocessors have many advantages for modern spacecraft, as the components are smaller and less expensive than their radiation-hardened counterpoints. SEEs can cause these microprocessors to fail in harsh radiation environments, including incorrect calculations and crashes. The Trikaya technique is designed to mask the effect of SEUs and SETs in microprocessor systems. Test results that show that Trikaya can be effective in decreasing corrupted computations by five to ten times.

References

[1] H. Quinn, T. Fairbanks, J. L. Tripp, G. Duran, and B. Lopez, "Single-event effects in low-cost, low-power microprocessors," in *Proceedings of the IEEE Radiation Effects Data Workshop*, Dec. 2014.

[2] H. Quinn, T. Fairbanks, J. L. Tripp, G. Duran, and B. Lopez, "Radiation effects in low-cost, low-power microprocessors," in *Proceedings of the Hardened Electronics and Radiation Technology Technical Interchange Meeting*, Apr. 2015.

[3] T. Fairbanks, H. Quinn, J. Tripp, J. Michel, A. Warniment, and N. Dallmann, "Compendium of TID, neutron, proton and heavy ion testing of satellite electronics for Los Alamos National Laboratory," in *2013 IEEE Radiation Effects Data Workshop (REDW)*, 2013, 10.1109/REDW.2013.6658191.

[4] H. Quinn, T. Fairbanks, J. L. Tripp, and A. Manuzzato, "The reliability of software algorithms and software-based mitigation techniques in digital signal processors," in *2013 IEEE Radiation Effects Data Workshop (REDW)*, 2013, 10.1109/REDW.2013.6658218.

[5] A. Suresh and J. Sartori, "Automated algorithmic error resilience based on outlier detection," *accepted to IEEE Micro*, 2015.

[6] S. Venkataramani, S. Chakradhar, K. Roy, and A. Raghunathan, "Computing approximately, and efficiently," in *the proceedings of Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, March 2015, pp. 748–751.

[7] B. Grigorian and G. Reinman, "Dynamically adaptive and reliable approximate computing using light-weight error analysis," in *the proceedings of the 2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, July 2014, pp. 248–255.

[8] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, Vol. C-33, No. 6, pp. 518–528, June 1984.

[9] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic approaches to low overhead fault detection for sparse linear algebra," in *the proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2012, pp. 1–12.

[10] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," *SIGPLAN Not.*, Vol. 47, No. 8, pp. 225–234, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2370036.2145845

[11] J. Camara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira, "Robustness-driven resilience evaluation of self-adaptive software systems," *accepted to IEEE Transactions on Dependable and Secure Computing*, 2015.

[12] C. L. McGhan, R. M. Murray, R. Serra, M. D. Ingham, M. Ono, T. Estlin, and B. C. Williams, "A risk-aware architecture for resilient spacecraft operations," in *the proceedings of the IEEE Aerospace Conference*, March 2015, pp. 1–15. [Online]. Available: 10.1109/AERO.2015.7119035

[13] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Nov 2003, pp. 581–588.

[14] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *IEEE Transactions on Nuclear Science*, Vol. 47, No. 6, pp. 2231–2236, Dec 2000.

[15] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, Vol. 51, No. 1, pp. 111–122, Mar 2002.

[16] H. Wang, H. Wang, and Z. Jin, "Bipartite graph-based control flow checking for cots-based small satellites," *Chinese Journal of Aeronautics*, Vol. 28, No. 3, pp. 883 – 893, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1000936115000734

[17] S. Asghari, H. Taheri, H. Pedram, and O. Kaynak, “Software-based control flow checking against transient faults in industrial environments,” *IEEE Transactions on Industrial Informatics*, Vol. 10, No. 1, pp. 481–490, Feb 2014.

[18] J. Azambuja, F. Sousa, L. Rosa, and F. Kastensmidt, “The limitations of software signature and basic block sizing in soft error fault coverage,” in *11th Latin American Test Workshop (LATW)*, March 2010, pp. 1–8. [Online]. Available: 10.1109/LATW.2010.5550346

[19] “The LLVM compiler infrastructure,” Last accessed 6/2015. [Online]. Available: http://llvm.org/

[20] “LLVM language reference manual,” Last accessed 9/2015. [Online]. Available: onwebathttp://llvm.org/docs/LangRef.html

[21] H. Quinn, W. H. Robinson, P. Rech, M. Aguirre, A. Barnard, M. Desogus, L. Entrena, M. Garcia-Valderas, S. M. Guertin, D. Kaeli, F. L. Kastensmidt, B. T. Kiddie, A. Sanchez-Clemente, M. S. Reorda, L. Sterpone, and M. Wirthlin, “The use of benchmarks for high-reliability systems,” *submitted to the IEEE Transactions on Nuclear Science*, 2015.

[22] B.-S. Wang, X.-J. Wang, and L.-K. Gong, “The construction of a Williams design and randomization in cross-over clinical trials using SAS,” *Journal of Statistical Software, Code Snippets*, Vol. 29, No. 1, pp. 1–10, 2 2009. [Online]. Available: http://www.jstatsoft.org/v29/c01