Comparison of Neutron Radiation Testing Approaches for a Complex SoC

Wesley Stirk, *Student Member, IEEE*, Evan Poff, *Student Member, IEEE*, Jackson Smith, *Student Member, IEEE*, Jeff Goeders, *Senior Member, IEEE*, Mike Wirthlin, *Senior Member, IEEE*

***Abstract*—Commercial Systems-on-a-Chip (SoC) have grown more abundant in recent years, including in space applications. This has led to the need to test SoCs in radiation environments, which is difficult due to their inherent complexity. In this work we present two complementary approaches to testing digital SoC devices—a bare metal approach and an operating system based approach—and discuss their advantages and disadvantages. Experimental data collected using these two methods in September 2021 from Los Alamos Neutron Science Center (LANSCE) on the Xilinx UltraScale+ MPSoC is presented and discussed.**

***Index Terms*—COTS, MPSoC, neutron testing, seu, single event upset, SoC, system on a chip, Xilinx**

I. Introduction

COMMERCIAL Systems-on-a-Chip (SoCs) devices have grown rapidly in computational capability, logic density, and I/O bandwidth. These devices contain multiple processing cores, multi-level caches, support for multiple I/O standards, and even integrated programmable logic for custom logic functions. These complex SoCs are used for a wide variety of applications that require high computational performance and low power requirements. The computational performance of these modern commercial off-the-shelf (COTS) SoC devices are attractive in many space applications that must perform increasingly complex computations with limited power availability. A number of projects have demonstrated the benefits of using these complex SoC devices in spacecraft applications [1]–[4].

Although these heterogeneous commercial SoC systems provide significant computing power for space-based applications, they are subject to the ionizing radiation found in space environments. This radiation may cause a variety of negative effects on the device that may limit its use in such environments. Of particular concern are single-event effects (SEE) such as single-event upsets (SEU) and single-event transients (SET) that cause non-destructive changes to the state of the SoC system. Because these systems contain multiple processors, multiple cache and scratch pad memories, and other components that contain a large amount of state, they are highly susceptible to single-event induced failures. Before using such devices in a radiation environment, the various SEE failure modes and failure rates need to be understood and measured.

Identifying the failure modes and rates of these complex SoCs with radiation testing is challenging. Creating the software system for an application to operate on a complex SoC for a radiation test is more complex than a single program running on a single processor. Further, failures observed at a radiation test on the SoC may be highly dependent on the way the SoC is configured and the application running in the processors. System failures observed in a radiation test with one application may not occur at the same rate or even at all for another SoC application. It can be difficult to generalize the radiation test results from one application example for use in estimating the failure rate of a different application. When radiation-induced failures do occur in these systems, it is often difficult to isolate the location of the failure and thus understand how other applications running on the SoC may fail in the future.

A long-term goal of our research is to investigate radiation testing methodologies that would provide greater observability into SoC failure mechanisms, with the ability to better locate the source of failures within complex SoCs that may have many different sub-components. We believe that as these techniques mature, the radiation test data will be better suited to be generalized and used to estimate the failure rate of other applications. Ideally, system designers will be able to more accurately predict how their individual application may perform, given the types of SoC resources it relies upon.

This work takes a step toward that overall goal by focusing on two techniques that provide better understanding of SoC behavior in high radiation environments. First, we seek to identify the failure rate (cross section) of several individual components of the SoC in isolation of the system. Second, we seek to organize our radiation testing in such a way that allows for measurements on multiple components at the same time to reduce testing time.

To explore and evaluate these approaches, we have developed two experimental systems, with differing approaches to gaining observability into the per-component failure rates. The first system utilizes a low-level, component-oriented approach implemented on a bare metal system (i.e., no operating system), while the second system is a high-level, operating system-based approach implemented with the Linux operating system. Both SoC radiation test approaches were applied to the Xilinx MPSoC device at Los Alamos Neutron Science Center (LANSCE). The radiation test results show that both approaches measure the neutron cross sections of several

The project or effort depicted was or is sponsored by the Department of the Defense, Defense Threat Reduction Agency under award HDTRA1-20-2-0002. The content of the information does not necessarily reflect the position or the policy of the federal government, and no official endorsement should be inferred.

All authors are with Electrical and Computer Engineering Department, Brigham Young University, Provo, UT 84602 USA

components within the SoC with similar results but using very different methods. The data is analyzed in Section VI and the similarity of the cross sections is emphasized in Sections VI-D and VI-E. The relative strengths and weaknesses of each approach will be described along with the individual testing results.

II. Background

In devices as complicated as an SoC, there are a variety of different components that could fail in a radiation environment. Each component of the system has a unique blend of susceptibility to radiation and criticality for the system as a whole. To fully understand how an SoC will behave in a radiation environment, the behavior of each of the components must also be understood, as well as the interaction between them. However, the interactions between components can make it difficult to ascertain the individual characteristics of each component. For example, if a piece of software performs a memory read, and the resulting data is incorrect, was the problem in the memory storage, the memory controller, the cache, or the CPU? A fault in any one of those components may appear similarly. Being able to isolate these differences is a major goal of the various methodologies that we present. This is different than merely creating a cross section for the entire chip, as is often done for processors (see Section II-B). Instead, our methodologies attempt to characterize the various components and aspects of the SoC and develop a cross section for each one.

Most research that has been done on radiation response of SoCs has focused on a single aspect of the SoC, such as a processor, a memory, an embedded FPGA, etc. [5]–[7]. On occasion, multiple aspects of the SoC are tested, but this is typically best viewed as concurrent independent tests rather than a single comprehensive test [8]. Such a test—a single test infrastructure that could test different aspects of an SoC—offers a number of potential benefits, such as more systematic testing of the entire SoC and reduced beam time (since all components are tested together, rather than in different tests).

A. *Cache Testing*

Caches are common in SoC devices and are used to hide the latency of memory accesses by storing strategic, quick-to-access copies of values that are likely to be used soon. These caches are valuable components to test because they comprise a sizeable amount of the physical area that might experience an SEU.

Several cache-centric radiation tests have been done on various systems. In [9], the authors induce cache conflicts through intentionally large inputs that will not fit inside the device's caches, noting the rates of errors such as Linux kernel panic events. Another prior work, [10], uses the RAMINDEX system register on various ARM processors to perform live cache inspection. They use these observations to identify and visualize the system caches' general behavior and access patterns.

Work similar to ours is presented in [11], [12], where they measure cache upsets through indirect, memory access patterns. We present a novel technique that uses a direct cache access. More detail on the differences is discussed in Section III-1.

B. *Processor Testing*

As expected due to their prevalence, processors of many varieties have been well studied in radiation environments [13]–[20]. These tests primarily comprise coming up with some kind of benchmark or computation to be run repeatedly while in the presence of radiation. Any failure in the benchmarks is attributed to radiation and is included in the total failure count for either the processor or the application.

While this approach is beneficial for understanding how the system may behave under a real computational load, they do not supply much information regarding how individual pieces of underlying hardware respond to radiation, and often workloads are CPU-centric, without exercising many specialized hardware modules. While our work still does incorporate a traditional approach of running a benchmark to stress the CPUs of our DUT, it is only one piece of a larger test framework that is designed to exercise a larger variety of SoC components.

C. *Xilinx MPSoC Device*

The Zynq Ultrascale+ MPSoC chip from Xilinx [21] was chosen to be our device under test (DUT) since it is a modern SoC with considerable complexity. It is a 16nm FinFet chip with four ARM Cortex-A53 application processing units (APU), two Cortex-R5F real-time processing units (RPU), an ARM Mali-400 MP2 GPU, L1 and L2 caches, several RAMs, FPGA fabric, and many specialized hardware modules, as shown in Figure 1.

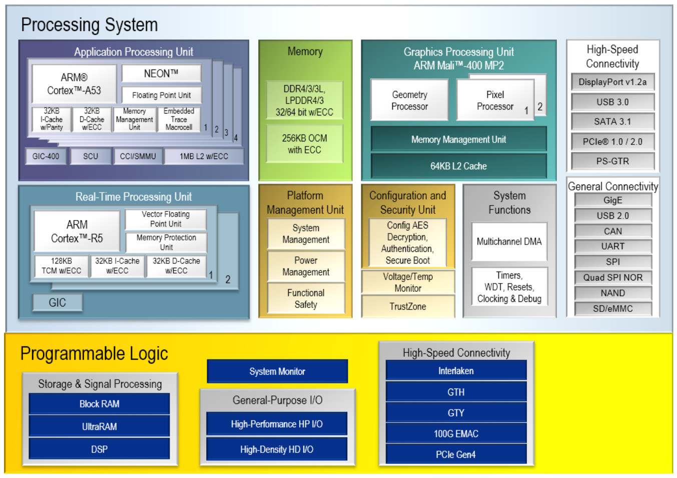

Fig. 1. Zynq MPSoC Block Diagram [22]

The caches are integrated with each of its four application processor cores. Each of these cores has its own L1 data and instruction caches that are each 32KiB in size, along with an associated tag RAM. These L1 caches are supported by a shared L2 cache, which is in turn supported by the main memory via a DDR controller. Each processor core also has a 512-entry translation lookaside buffer (TLB) [21]. Other components, such as the On-Chip Memory (OCM), DMA, FPGA, IO, etc. are independent and can be controlled by any of the processors.

While this chip has been studied in radiation environments before [5]–[8], [23], the research on this chip has generally focused on a specific aspect of the chip, rather than the "SoC" nature of the chip. The response of the Ultrascale+ FPGA fabric of the device has been reported in [6], [7], [23]. Other works have also focused on the APU processing cores and using the DUT to explore software reliability techniques [17], [18]. Xilinx has shown results on the MPSoC from a proprietary tool called the System Validation Tool (SVT), which runs randomized test vectors through the system [5], [24]; however, the tool's goal is to report a single cross section for the device, rather than studying the failure mechanisms of individual components.

III. Component Based Testing Methodology

The first method we explored is a component-based, bare metal approach. Components of an SoC are typically all connected to and controllable from the software processor, usually via the memory bus and memory mapped registers. Our technique in this approach is to create test code that runs on the processor and exercises the functionality of several SoC components, as individually as possible. By using bare metal code, we have direct access to device registers, giving us a high-level of control over each component, increasing the ability of the tests to isolate the radiation characteristics of a single component.

While tests for each component need to be relatively independent, that does not negate the possibility of running multiple component tests together. Non-interfering component tests can be collected and run one after another in a "round-robin" fashion. In other words, each test has the opportunity to run through its test. Once all tests have finished a single iteration, the process is repeated, as can be seen in Figure 2. (i.e., repeatedly going through the list of tests). Running these tests together increases the amount of information that can be collected on the SoC, while still maintaining the isolation of the individual components.

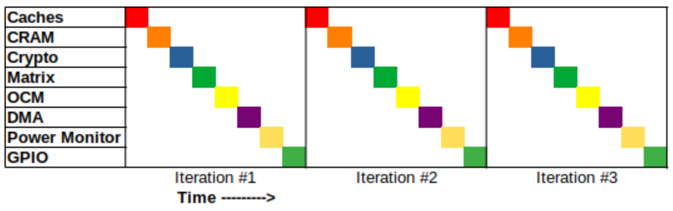

Fig. 2. Round-Robin Component Scheduling

For our experiment, we developed independent tests for the L1 caches, configuration RAM (CRAM), Advanced Encryption Standard (AES) and Secure Hash Algorithm (SHA) acceleration hardware (labeled as Crypto), the CPU (running matrix multiplication, a compute-heavy application, labeled as 'Matrix'), OCM, DMA, GPIO, as well as power monitoring code. The cache test was separated into its own executable on a different board, but all the other tests were combined into a single executable that cycled through the tests in a round-robin fashion, as shown in Figure 2. This process was continued indefinitely, until an error was detected. Once an error was detected, the test was allowed to continue for a short amount of time, to see if more upsets occurred. Then the board was power-cycled and the test was started over. An exception to this was for upsets in the CRAM, because they occurred frequently and had no effect on any of the other components.

It is worth noting that while these tests are run in sequence, we expect that some components tests will accumulate errors for the entire time frame. For example, while other tests are executing, memory upsets will continue to accumulate in the OCM and CRAM, and when the round-robin testing process returns to these tests, the upsets will be detected. This allows for potentially greater data collection than if these tests were strictly run independently. However, other tests such as the matrix multiplication (CPU) test, Crypto and DMA, primarily exercise their modules only while running their test. In such cases we anticipate if these modules were tested more often, more upsets would be detected. Furthermore, some of these modules, such as the Crypto and DMA, can be run asynchronously, meaning it would be possible to continue with testing other components while they executed, potentially increasing the amount of data that could be collected. Unfortunately we did not explore this approach in this work, and it will need to be investigated in the future.

This overlapping effect offered by the round-robin component-based approach is different than other approaches that may execute different programs in sequence, such as the work in [8], that executed different software executables in sequence to test different component failure rates. While our approach offers the benefit of collecting more upset data in less time, there is always a concern with how failures in one component may trigger failures in another. However, such cross-component failures may also be triggered when attempting to test one component in isolation, so future work will likely be required to further investigate this, and improve our methodologies.

It should also be noted that our previously described round-robin testing approach runs all of the code on one CPU core. This includes the CPU-specific matrix multiplication test that is described later in this section. While the DUT contains four CPU cores, we have not yet expanded our test framework to operate on multiple cores, and instead chose to focus on supporting a variety of components, rather than multiple CPU cores. In the future we hope to leverage additional cores to conduct more testing in parallel. Such an experiment may require additional exploration to understand how interactions between CPU cores may affect accumulation of error rates.

The following is a description of the various component tests that we chose to use, as well as a discussion of possible variations

1) Caches: Our cache test creates an on-DUT, *a posteriori* golden copy of initial cache contents through direct reads and then performs DUT-driven, mid-irradiation checks against current cache contents to detect upsets [25]. This was in contrast to prior works that approximate cache upset rates through simulation and fault injection or to those that measure cache upsets through indirect, *a priori* memory access patterns, such as [11], [12]. To do this, software running on each APU core directly read the caches and compared them against a golden copy. Detected differences were logged and sent sequentially from the individual processors to the experiment's host computer. (More details on the experimental setup can be found in Section V.) To allow the other methods to run independently and with their boards' caches enabled, this bare-metal cache test ran on a dedicated board.

This straightforward approach was made possible by the use of co-processing registers of the ARM Cortex A53 processors (CP15 registers) that allow access to the cache by *way*, *set*, and *offset* via platform-specific assembly instructions (see [26], *Chapter 6. Level 1 Memory System*) (previous work such as [11], [12] had to use indirect methods to obtain cache data). A golden copy of the contents of each APU processor's L1 data cache, data tag RAM, and TLB memories were stored in system memory at runtime by first disabling the caches and then reading the values at each cache location directly. Because our DUT uses Cortex-A53 processors, our implementation used AArch64 instructions to write encoded locations to dedicated system registers, which in turn placed current cache values into readable locations[26]. These direct reads continued throughout the test to compare current values against the golden copy's value for that location. When the current value and the golden copy's value disagreed, the difference was logged, and the board was reprogrammed to restart the test program [25].

2) RAM: As noted in Section II-C, the MPSoC has a variety of SRAM memories, including the on-chip memory (OCM), tightly-coupled memory (TCM), and CRAM [21].

While the OCM and TCM are general purpose memories, the CRAM is the memory used to store the configuration of the FPGA fabric of the chip and is not designed to be accessed regularly. In fact, it is not part of the memory space of the processors at all and is only accessible through the processor configuration-access port (PCAP) interface [21]. In order to access the CRAM, we used a modification of the PCAP driver described in [8], [27].

A variety of methods can be used to test a RAM:

1) **Initialization:** The contents of the RAM are homogeneously initialized to a known state. Any portion of the RAM contents can then be easily checked for changes.
2) **Golden Copy:** A copy of the RAM contents can be created, regardless of the value, and be placed in a different location, where it will not get corrupted. The RAM contents can then be compared against the golden copy at any point during the test.
3) **Error Detection Algorithms:** A variety of error detection algorithms exist (such as a cyclic rule check (CRC)) that allow for large chunks of memory to be tested for errors at a time. However, some algorithms may only report on the existence of errors and not their location in the memory.
4) **Built-in Error Detection:** Some RAMs have built-in error detection and correction. Typically, when the RAM detects an error, it will generate a system interrupt.

For our OCM test, we used a combination of *Initialization* and *Built-in Error Detection*. For our CRAM test, we used the *Golden Copy* method and stored a copy of the CRAM in DDR memory. While our test framework was set up to use these techniques, unfortunately our test code had a fatal bug that went unnoticed during our experiments: our interrupt handling code caught the upset event, but failed to *report* that it occurred. It did run code to *correct* the upsets, but unfortunately we had no observability that this was happening, so we were not able to collect data on these memory upsets.

3) CPUs / Computation: Testing of CPUs in a radiation environment is a well studied area with a variety of possible techniques (see Section II-B). A common option is to run a compute-heavy benchmark and monitor the correctness of the computed result. We followed this pattern and had the CPU perform a 125x125 matrix multiplication and an element-by-element division and exponentiation.

While this tests the memory controller, cache, etc. in addition to the CPU, it is a fair approximation of normal work loads, even compared to other component tests. For example, as described above in Section III-2, to test the CRAM the golden copy must be read from memory, using the memory system.

4) Cryptographic Acceleration: The MPSoC has hardware dedicated to accelerating the Advanced Encryption Standard (AES) and Secure Hash Algorithms (SHA), common algorithms in cryptography [21]. To test this functionality, we pre-generated a random array that was first passed through the AES accelerator and then the SHA accelerator. The correct hash of the encrypted data was pre-computed and was compared against the calculated answer.

5) DMA: In order to test the DMA, a section of memory was initialized which was then copied to a different location using the DMA core. Afterwards, the two memory regions were compared to look for differences. Any differences are assumed to be induced because of the DMA hardware.

Similar to the CPU test in Section III-3, this will also test the memory controller, and so is not 100% isolated. However, this is how the DMA actually gets used, and still produces valuable data.

6) Considerations and Issues: Our bare metal component-based approach does present a few potential problems. One of these is that the code and CPU used to test the components is in fact subject to radiation-induced faults itself. This has the potential to completely crash the CPU or cause the software to fail, depending on the location of the fault. However, when such a catastrophic event occurs, the test executable stops reporting information, which can be detected and made to trigger a system reset. Unfortunately in this case, the data on the fault that led to failure will be lost. Our experiments indicate that this happens relatively infrequently on the MPSoC compared to the amount of other observed upsets.

Even when the software does not crash, it does not guarantee that the component tests performed 100% correctly. Any result that indicates a fault in a component must be analyzed to ensure that it was truly caused by a fault in the component and not the controlling software/CPU. For example, a reported DMA failure may actually be the result of a CPU upset while running the DMA test code. In these cases, additional data collection may be necessary to distinguish between the two types of events. Alternatively, it may be possible to subtract the cross section of the CPU in determining per-component cross sections. This type of analysis was unfortunately not included in our experiments, and future work would be necessary to explore approaches to disambiguate these events. However, the failure rate of our DUT's CPU was very low, so while the possibility of a false interpretation of radiation induced faults is certainly possible, we believe it would be rare for our test platform.

It is also worth noting that the SoC architecture of our DUT contains many heterogeneous compute and memory components that are largely independent, with point-to-point connections to the processor. This makes the DUT well suited for our testing approach where multiple SoC components can be evaluated during the same test. Other SoC architectures may be more prohibitive, such as architectures that rely on mesh networks or networks-on-chip (NoCs) to pass data along across several different components [28]. In such architectures it may be much more difficult to isolate which individual component exhibited a failure.

While there are certainly issues to consider, our bare metal component-based method offers advantages as well. As mentioned earlier, the bare metal approach gives maximum control over each component, and the ability to isolate radiation characteristics. In addition, by using the SoC itself to test the various components, the experiment is self-contained. No external equipment is required beyond the standard equipment that is used to program and communicate with the SoC. This lends itself well to testing COTS (commercial-off-the-shelf) SoCs since development boards are readily available to facilitate programming and communication.

IV. Operating System-Based Method

The second SoC testing approach used in this work involves the use of a Linux operating system running on the MPSoC device and configuring the kernel in such a way as to report as many system and component errors as possible. SEU induced failures that occur within the system are detected by a variety of error reporting mechanisms within the kernel, device drivers, and application software. The two primary benefits of using an operating system like Linux for SoC testing is the ease of creating the test software environment and the ease of measuring the failure rate of multiple components. If the Linux operating system has been properly ported to the SoC of interest, then the operating system software contains most of the code necessary to exercise many of the components within the SoC and report on these component failures. Much of the low-level software needed to interact with these itnernal components already exists within the device driver kernel code ported for the device. A complex kernel with a memory management unit, support for multiple cores, and device drivers for the devices on the SoC can load the SoC much more effectively than in a bare metal system. Such loading on the SoC will expose failure modes that are not visible in a bare metal system.

A number of previous works have demonstrated the importance and impact of the operating system when performing radiation testing on a processor. [13] provides a set of guidelines on methods for testing microprocessor devices and emphasises the importance of the operating system on the overall results. This work suggests that if a complex operating system is used, it will heavily influence the results and interfere with attempts to characterize the basic response of the processor. This worked demonstrated the impact on processor hang rate using the Windows NT operating system. [29] provides a case study on microprocessor testing that suggests systems should be tested using the operating system used in flight to adequately understand full system failures. [30] measures the improvement in reliability provided by two fault-tolerant real-time operating systems operating on a 28-nm ARM processor core. [31] investigates the impact of an operating system on an embedded SoC. Results from this work indicate that the O/S had limited impact on silent data corruption but a significant effect on the functional interrupt rate.

There are a few disadvantages of using a complex operating system for SoC testing. First, the cross section of the processor system itself will be higher with Linux than in a bare metal system. This higher processor system cross section is a result of more system resources being utilized and thus more opportunities for failure. A higher processor system cross section will make it more difficult to tease out errors in other components as the processor will fail more often between other component failures. Second, it may not be possible to test some components for radiation-induced failure. If there are no device drivers for a component found within a particular SoC then it is not possible to access the component to check on its behavior at run-time. In addition, some components within a system cannot be accessed by the processor when the processor is configured for virtual memory. Third, there will be generally less control of the components under test within Linux and a pre-designed device driver. The user may have less control over configuring a component, monitoring a component, or reporting component failures in a Linux test than with bare metal.

1) Linux Test System: For our experimentation, we created a Linux system to test the MPSoC device and measure the neutron cross section of several components. Several kernel configuration changes were made to the standard "out-of-the-box" Linux kernel configuration for the device. These include enabling ECC on the caches and memory controller, increasing the kernel error reporting level, and enabling several kernel debugging features. Several application programs (described below) were configured to run on the system after kernel booting to provide both a system load and a test for SoC components. In addition, a custom PCAP driver was included in the kernel for CRAM scrubbing and error reporting. The system was designed to test the processor execution, cache upsets, and upsets within the configuration memory.

2) Linux Test Methodology: The primary goal of the Linux testing methodology is to measure the reliability of the Linux kernel executing on the application processors. To provide a computational load on the processors as well as the I/O system, the Dhrystone benchmark is executed on all four A53 processors. While the Dhrsytone benchmark is not a perfect test of the entire computing system [32], it keeps the CPUs busy computing, making OS related problems more likely to be observed. The Linux kernel contains significant self checking and error reporting code and provides significant detail on a variety of different exceptions, such as kernel panics.

The experiment is organized to identify the following processor errors:

1) **Boot Failure:** The processor fails to boot and initialize the kernel.
2) **Process Failure:** One or more of the four Dhrystone processes fail.
3) **Kernel Failure:** The Linux kernel stops operating.

The details of kernel failures are logged and categorized to understand the kernel failure mechanism.

3) Linux Cache Testing: Upsets in the cache memories can be easily logged with the Linux test by enabling the ECC driver within the Linux kernel. When this driver is enabled, the ECC module on the cache memories is enabled and the ECC interrupts are caught by the kernel when ECC events are detected. Both single-bit and double-bit errors are reported, as well as the location within the cache. The logging system captures the details of each event for later analysis. Events are captured for ECC errors in the L1 and L2 caches and the TLB.

4) Linux CRAM Testing: The Linux test was also configured to capture upsets within the configuration memory of the programmable logic. Measuring CRAM upsets is usually done using external means such as JTAG. In this experiment, we use the PCAP component of the MPSoC to access the programmable logic from within the processing system. Counting CRAM upsets facilitates the calculation of the neutron cross section of the programmable logic.

Although a driver for accessing the PCAP is available for the kernel, this driver does not provide sufficient control of the PCAP or visibility of CRAM errors. Further, the PCAP is isolated from the kernel requiring changes in the kernel security controls. A custom driver for the PCAP was written to provide low-level access and control of the PCAP component. This driver is much more complicated and required significantly more effort than the low-level driver code used to access the PCAP in the bare metal component tests.

Other components could have been tested within the Linux system but additional device drivers would need to be configured and written to support such tests. However, some components may not be accessible from the kernel or may be configured within Linux in a manner that makes it unusable for a radiation test.

V. Experiment Setup

To test our proposed techniques, we performed an initial set of experiments in September 2021 at the Los Alamos Neutron Science Center (LANSCE). Our experiment was run in the experimental area known as ICE House I, using the 30L neutron flight path [33], [34]. This neutron beam provides a spectrum similar to what can be found in the atmosphere, but at a much higher flux [34].

In these experiments, we used our two approaches (bare metal and Linux) to find the single event upset (SEU) cross sections of components of the MPSoC chip. We placed five Ultra96 development boards in series to collect data for multiple experiments simultaneously over the five day period at LANSCE. The physical setup of these boards can be seen in Figure 3. Three boards were used to test the bare metal, component-level test. Another board was dedicated for the bare metal cache characterization technique. The final board was used for the system-level Linux testing approach.

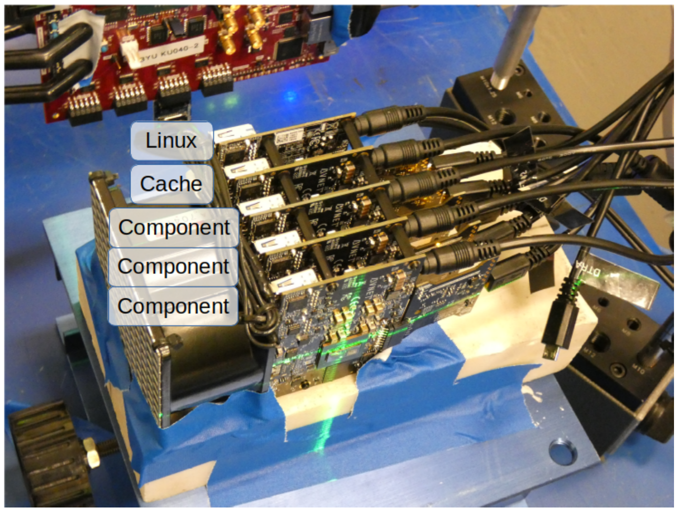

Fig. 3. Ultra96 Board Setup

The logical setup for a single board can be seen in Figure 4. In this setup, the host computer monitors the experiment from a safe location, but can communicate with and program the boards over UART and JTAG. The DUT reports the errors and state of health of its test over a UART communication link where it is logged and monitored by the host computer. When an upset is identified or the DUT fails to communicate over the UART, the host computer power cycles the DUT via a network-controlled power switch (the Netbooter in the diagram) and reprograms the DUT to begin the experiment once again. The dotted rectangle in Figure 4 represents the dividing wall between the experimental area and a safe area where radiation was not a concern.

VI. Results

The experiment was conducted for five days, during which our test area received a total neutron fluence of 2.72 $\times 10^{11}$ n/cm^2. The actual neutron fluence for each experiment was carefully measured by recording the net neutron fluence for each time frame that the board was executing the test code and summing across all executions. Most experiments have a lower total fluence as the boards were not running all of the time. Since the bare metal component experiments were running on multiple boards, these experiments have a net fluence greater than the total fluence.

The results for each experiment were tabulated by analyzing the extensive text logs with custom processing scripts. Timestamps are included in the logs to identify the start and stop

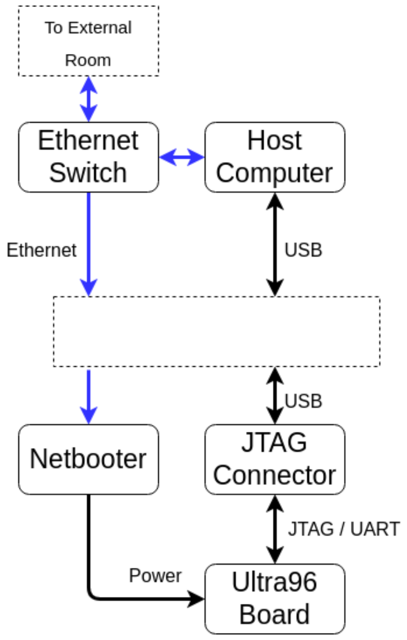

Fig. 4. Logical Experiment Setup

time of each experiment run to determine the total fluence. These scripts identified each of the reported failures during the experimental run.

A. Component Results

The results from the bare metal component test are summarized in Table I. These results include the estimated cross section of the CRAM using the PCAP component test, the Crypto component test (utilizing the SHA and AES accelerators), and the Matrix multiply benchmark test. These tests were all running as part of the same executable on the same board, demonstrating the effectiveness of running multiple experiments simultaneously.

TABLE I
BARE METAL TEST CROSS SECTIONS

	Total Upsets	Device Cross Section (cm^2)	Size (KiB)	Bit Cross Section (cm^2)
CRAM	3773	8.80×10^{-9}	5531	1.94×10^{-16}
Crypto	6	1.40×10^{-11}		
Matrix	5	1.17×10^{-11}		

Total Fluence: 4.288×10^{11} neutrons cm^{-2}

Not all of the component tests were successful. As shown by Figure 2, software for a number of additional components were included in the component test suite; however, several of these tests produced inconsistent results due to software bugs and testing anomalies. For example, the OCM had accidentally been configured in such a way where the ECC was enabled, but ECC interrupts were not. The end result is that single-bit upsets were *corrected* (which prevented the software from correctly checking), but not *reported*. These tests will be repeated in the future to address the problems observed at this test.

The results in Table I show very few upsets for the Crypto and Matrix tests (6 and 5 respectively). Previous works that have tested this part at LANSCE have likewise had difficulty observing significant compute errors [8], [18]. While we believe that the results show that the proposed technique is capable of observing compute errors, more beam time, or a higher flux facility would likely be required to better gain more accurate cross-sections and better evaluate the effectiveness of the proposed method.

B. Cache Results

The cache tests were performed on a dedicated board and the resulting data is shown in Table II. Unlike the other component tests, the cache test must be run on an isolated board so that the cache contents can be carefully controlled and monitored. The cache test relied on disabling the caches so that the contents were stable. The error counts within the cache are used to estimate the neutron cross section of the data bits, tag bits, and TLB of the L1 data caches. Although cache errors were observed, bugs in the cache test software only tested 1/8th of the cache memory, resulting in relatively few upset events and thus large error bars.

TABLE II
CACHE TEST CROSS SECTIONS WITH 95% CONFIDENCE INTERVALS

	Total Upsets	Device Cross Section (cm^2)	Lower 95%	Bit Cross Section (cm^2)	Upper 95%
Data	17	9.33×10^{-11}	4.14×10^{-16}	7.11×10^{-16}	1.14×10^{-15}
Data Tag	5	2.74×10^{-11}	6.70×10^{-17}	2.09×10^{-16}	4.90×10^{-16}
TLB	29	3.50×10^{-10}	3.57×10^{-15}	5.34×10^{-15}	7.66×10^{-15}

Total Fluence (Cache/Tag): 1.823×10^{11} neutrons cm^{-2}
Total Fluence (TLB): 8.288×10^{10} neutrons cm^{-2}
Note: The TLB has lower fluence as it was not tested for the entire duration of the testing.

C. Linux Results

Table III summarizes the data from our Linux system experiment. The Linux test includes the number of upsets and the estimated cross section values for CRAM and caches. In addition, it includes the total number of unexpected SoC failures that required a power cycle. These events are grouped by the cause of the restart:

Process Failure / Hangs: At some point during normal operation, the system stops responding and no more messages were received.

Kernel Failures: The Linux kernel reported a failure of some kind and then optionally reset the system.

APU Reset: The first stage bootloader (FSBL) and kernel boot up began without a prior warning or message about a problem. In other words, the processors were reset, but the reason went unreported.

As discussed below in Sections VI-D and VI-E, the cross-sections that are compared between Linux and the bare metal tests are comparable. This is somewhat surprising since Linux crashes more often, which we would expect to skew the results. As noted earlier, we were careful to only include the fluence

TABLE III
LINUX TEST CROSS SECTIONS

	Total Upsets	Device Cross Section (cm^2)	Size (KiB)	Bit Cross Section (cm^2)
CRAM	1453	1.32×10^{-8}	5531	2.91×10^{-16}
L1 Cache	148	1.34×10^{-9}	256	6.41×10^{-16}
L2 Cache	648	5.89×10^{-9}	1024	7.02×10^{-16}
TLB	23	2.09×10^{-10}	8	3.20×10^{-15}
Proc. Fail/Hang	13	1.18×10^{-10}		
Kernel Failures	17	1.55×10^{-10}		
APU Reset	38	3.45×10^{-10}		
SoC Total	68	6.18×10^{-10}		

Total Fluence: 1.100×10^{11} neutrons cm^{-2}

while any experiment was actively running. In other words, only the data from when Linux worked was tabulated in the final answer. However, any silent Linux failures would not be included.

D. Comparing Cache Results and Methods

As shown in Table IV, we find the bare metal and Linux methods produce comparable results for the cache memories. The data cache bit cross sections differ only by 4.53%. The data tag and TLB RAMs are not as close, with percent differences of 79.54% and 50.41% respectively, but the cross sections for the data, data tag and TLB RAMs are all within the same order of magnitude and have overlapping 95% confidence interval error bars. Both the Linux and bare metal methods produce detailed cache results, recording information about which cache was upset, address, and bit location. While the Linux method provides insight into more cache locations (including the L2 cache and snoop rams), the bare metal approach does not depend on the built in dedicated error correction and detection functionality. One downside of the Linux/ECC approach is that multiple cache upsets in quick succession might give insufficient information to identify the bit location, as the APU registers can only hold information for one upset at a time. Since the bare metal approach reads the contents of the cache directly, this problem is avoided. An upside of the bare metal approach is that we can identify the actual location of the error for the L1 instruction cache upsets whereas the Linux system cannot identify the location since only detection is supported with parity.

E. Comparing CRAM Results and Methods

In the case of the configuration memory, both the Linux and bare metal method implement the same functionality. As shown in Figure 5, these results produce similar data, suggesting that the two approaches would tend to yield similar bit cross sections results over repeated experimentation. To provide an external reference for these methods' results, we compare these confidence intervals with device reliability numbers provided by Xilinx. Xilinx's data, which was also obtained at LANSCE, list the bit cross section for UltraScale+ CRAM as being 2.67×10^{-16} cm^2 bit^{-1}[23].

TABLE IV
COMPARISONS OF BIT CROSS SECTIONS

	Total Upsets	Device Cross Section (cm^2)	Lower 95%	Bit Cross Section (cm^2)	Upper 95%
			Bare-metal		
Data	17	9.33×10^{-11}	4.14×10^{-16}	7.11×10^{-16}	1.14×10^{-15}
Data Tag	5	2.74×10^{-11}	6.70×10^{-17}	2.09×10^{-16}	4.90×10^{-16}
TLB	29	3.50×10^{-10}	3.57×10^{-15}	5.34×10^{-15}	7.66×10^{-15}
CRAM	3773	8.80×10^{-9}	1.88×10^{-16}	1.94×10^{-16}	2.01×10^{-16}
			Linux		
Data	86	7.82×10^{-10}	5.85×10^{-16}	7.46×10^{-16}	9.06×10^{-16}
Data Tag	7	6.36×10^{-11}	1.94×10^{-16}	4.86×10^{-16}	9.99×10^{-16}
TLB	23	2.09×10^{-10}	2.03×10^{-15}	3.19×10^{-15}	4.77×10^{-15}
CRAM	1453	1.32×10^{-8}	2.76×10^{-16}	2.92×10^{-16}	3.07×10^{-16}

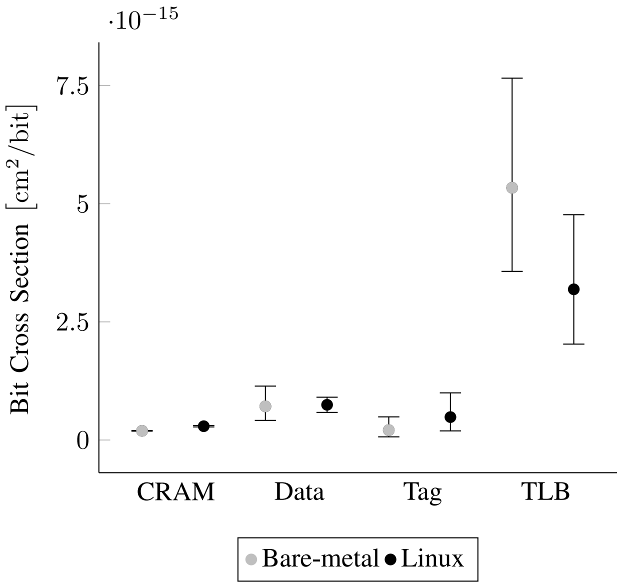

Fig. 5. A scatter plot of bare-metal and Linux method results

VII. FUTURE WORK

Our long-term research aims to develop testing and characterization methodologies that can be used to more effectively test complex SoC devices and understand how they behave and fail in radiation environments. We feel like this test was a step in the direction of rigorous and comprehensive SoC radiation testing, but that there is clearly more to accomplish.

In this experiment, the bare metal approach tested a relatively small subset of the available hardware components in the MPSoC. We plan on developing a larger suite of component tests that will be used in the future to better understand more aspects of the MPSoC. For example, as described in Section III-3, a matrix multiplication not only tests the CPU, but inadvertently tests the memory system as well. In subsequent tests, we hope to find a way to test the CPU alone.

Many modern SoCs, such as the MPSoC, have multiple processors on board. Some work has been done on understanding the sensitivity of many-core chips [16], we hope to better test the interactions of the multiple cores in an SoC and explore the advantages that multiple cores can offer to SoC testing.

VIII. Conclusions

In this work, we have shown two complementary approaches for testing SoC radiation response with the goal of understanding failure rates of individual components, rather than just the system as a whole. A bare metal approach was presented that executed self-test software of several system components, as well as a Linux system-level technique that relied on operating system logging with custom kernel modifications. Both systems were successful in providing insight into failure rates of individual SoC components; however, much future work remains to develop more robust test software and to test more aspects of the MPSoC and other SoCs.

References

[1] Z. Towfic, D. Ogbe, J. Sauvageau, *et al.*, "Benchmarking and Testing of Qualcomm Snapdragon System-on-Chip for JPL Space Applications and Missions," in *IEEE Aerospace Conference*, Big Sky, MT, USA, Aug. 2022, pp. 5008–5019.

[2] D. Rudolph, C. Wilson, J. Stewart, *et al.*, "CSP: A multifaceted hybrid architecture for space computing," in *Conference on Small Satellites*, Logan, UT, USA, Aug. 2014.

[3] C. Wilson and A. George, "CSP hybrid space computing," *Journal of Aerospace Information Systems*, vol. 15, no. 4, pp. 215–227, Apr. 2018.

[4] X. Iturbe, D. Keymeulen, P. Yiu, *et al.*, "On the Use of System-on-Chip Technology in Next-Generation Instruments Avionics for Space Exploration," in *VLSI-SoC: Design for Reliability, Security, and Low Power*, Y. Shin, C. Y. Tsui, J.-J. Kim, K. Choi, and R. Reis, Eds., Cham, 2016, pp. 1–22.

[5] P. Maillard, J. Arver, C. Smith, O. Ballan, M. J. Hart, and Y. P. Chen, "Test Methodology & Neutron Characterization of Xilinx 16nm Zynq UltraScale+ Multi-Processor System-on-Chip (MPSoC)," in *IEEE Radiation Effects Data Workshop (REDW)*, Waikoloa, HI, USA, Jul. 2018, pp. 206–209.

[6] M. Glorieux, A. Evans, T. Lange, *et al.*, "Single-Event Characterization of Xilinx UltraScale+® MPSoC under Standard and Ultra-High Energy Heavy-Ion Irradiation," in *IEEE Radiation Effects Data Workshop (REDW)*, Waikoloa, HI, USA, Dec. 2018, pp. 189–193.

[7] D. M. Hiemstra, V. Kirischian, and J. Brelski, "Single Event Upset Characterization of the Zynq UltraScale+ MPSoC Using Proton Irradiation," in *IEEE Radiation Effects Data Workshop (REDW)*, New Orleans, LA, USA, Jul. 2017, pp. 135–138.

[8] J. D. Anderson, J. C. Leavitt, and M. J. Wirthlin, "Neutron radiation beam results for the Xilinx UltraScale+ MPSoC," in *IEEE Radiation Effects Data Workshop (REDW)*, Waikoloa, HI, USA, Jul. 2018, pp. 194–200.

[9] T. Santini, P. Rech, L. Carro, and F. R. Wagner, "Exploiting cache conflicts to reduce radiation sensitivity of operating systems on embedded systems," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Amsterdam, Netherlands, Oct. 2015, pp. 49–58.

[10] D. Tarapore, S. Roozkhosh, S. Brzozowski, and R. Mancuso, "Observing the Invisible: Live Cache Inspection for High-Performance Embedded Systems," *IEEE Transactions on Computers*, vol. 71, no. 3, pp. 559–572, Feb. 2021.

[11] E. Carlisle and A. D. George, "Cache fault injection with DrSEUs," in *IEEE Aerospace Conference*, Big Sky, MT, USA, Mar. 2018, pp. 3137–3147.

[12] F. Irom, F. Farmanesh, A. Johnston, G. Swift, and D. Millward, "Single-event upset in commercial silicon-on-insulator PowerPC microprocessors," *IEEE Transactions on Nuclear Science*, vol. 49, no. 6, pp. 3148–3155, Dec. 2002.

[13] F. Irom, "Guideline for ground radiation testing of microprocessors in the space radiation environment," Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, Tech. Rep. 08-13 4/08, 2008.

[14] D. Oliveira, F. F. dos Santos, G. Piscoya Dávila, *et al.*, "High-Energy Versus Thermal Neutron Contribution to Processor and Memory Error Rates," *IEEE Transactions on Nuclear Science*, vol. 67, no. 6, pp. 1161–1168, Jun. 2020.

[15] H. Quinn, T. Fairbanks, J. L. Tripp, G. Duran, and B. Lopez, "Single-Event Effects in Low-Cost, Low-Power Microprocessors," in *IEEE Radiation Effects Data Workshop (REDW)*, Paris, France, Jul. 2014, pp. 246–254.

[16] V. Vargas, P. Ramos, V. Ray, *et al.*, "Radiation Experiments on a 28 nm Single-Chip Many-Core Processor and SEU Error-Rate Prediction," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 483–490, Dec. 2016.

[17] M. Bohman, B. James, M. J. Wirthlin, H. Quinn, and J. Goeders, "Microcontroller Compiler-Assisted Software Fault Tolerance," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 223–232, Dec. 2018.

[18] B. James, H. Quinn, M. Wirthlin, and J. Goeders, "Applying compiler-automated software fault tolerance to multiple processor platforms," *IEEE Transactions on Nuclear Science*, vol. 67, no. 1, pp. 321–327, Dec. 2019.

[19] "Software resilience and the effectiveness of software mitigation in microcontrollers," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2532–2538, Dec. 2015.

[20] A. Lindoso, M. García-Valderas, L. Entrena, Y. Morilla, and P. Martín-Holgado, "Evaluation of the Suitability of NEON SIMD Microprocessor Extensions Under Proton Irradiation," *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1835–1842, Apr. 2018.

[21] Xilinx, "Zynq UltraScale+ Device Technical Reference Manual (UG1085 v2.2)," Tech. Rep., Dec. 2020.

[22] Xilinx, "Zynq UltraScale+ MPSoC product tables and product selection guide," Tech. Rep., 2016.

[23] Xilinx, "Device Reliability Report First Half 2021 (UG116)," Tech. Rep., Nov. 2021.

[24] P. Maillard, M. Hart, J. Barton, J. Arver, and C. Smith, "Neutron, 64 MeV proton & alpha single-event characterization of Xilinx 16nm FinFET Zynq UltraScale+ MPSoC," in *IEEE Radiation Effects Data Workshop (REDW)*, New Orleans, LA, USA, Jul. 2017, pp. 139–143.

[25] E. Poff, "A Direct-Read, A Posteriori Golden Copy Method for Measuring SoC Cache Upsets," M.S. thesis, Brigham Young University, Jun. 2022.

[26] ARM, "Arm Cortex-A53 MPCore Processor Technical Reference Manual (Revision: r0p4)," Tech. Rep., 2018.

[27] A. Stoddard, A. Gruwell, P. Zabriskie, and M. Wirthlin, "High-speed PCAP configuration scrubbing on Zynq-7000 All Programmable SoCs," in *International Conference on Field Programmable Logic and Applications (FPL)*, Lausanne, Switzerland, Sep. 2016, pp. 21–28.

[28] R. Schooler, "Tile processors: Many-core for embedded and cloud computing," in *Workshop on High Performance Embedded Computing*, vol. 35, Lexington, MA, USA, Sep. 2010.

[29] H. Quinn, "Challenges in testing complex systems," *IEEE Transactions on Nuclear Science*, vol. 61, no. 2, pp. 766–786, Apr. 2014.

[30] T. Santini, C. Borchert, C. Dietrich, *et al.*, "Effectiveness of software-based hardening for radiation-induced soft errors in real-time operating systems," in *International Conference on Architecture of Computing Systems*, Springer, Vienna, Austria, Mar. 2017, pp. 3–15.

[31] T. Santini, L. Carro, F. Rech Wagner, and P. Rech, "Reliability analysis of operating systems and software stack for embedded systems," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2225–2232, Mar. 2016.

[32] A. R. Weiss, "Dhrystone benchmark: History, analysis, scores and recommendations," 2002.

[33] S. Wender and L. Dominik, "Los Alamos High-Energy Neutron Testing Handbook," Tech. Rep., 2019.

[34] Los Alamos National Science Center. "Weapons Neutron Research Flight Paths." (May 19, 2022), [Online]. Available: https://lansce.lanl.gov/facilities/wnr/flight-paths/index.php.