Precise Cache Profiling for Studying Radiation Effects

JAMES MARSHALL, George Washington University, USA
ROBERT GIFFORD, University of Pennsylvania, USA
GEDARE BLOOM, University of Colorado Colorado Springs, USA
GABRIEL PARMER and RAHUL SIMHA, George Washington University, USA

Increased access to space has led to an increase in the usage of commodity processors in radiation environments. These processors are vulnerable to transient faults such as single event upsets that may cause bit-flips in processor components. Caches in particular are vulnerable due to their relatively large area, yet are often omitted from fault injection testing because many processors do not provide direct access to cache contents and they are often not fully modeled by simulators. The performance benefits of caches make disabling them undesirable, and the presence of error correcting codes is insufficient to correct for increasingly common multiple bit upsets.

This work explores building a program's cache profile by collecting cache usage information at an instruction granularity via commonly available on-chip debugging interfaces. The profile provides a tighter bound than cache utilization for cache vulnerability estimates (50% for several benchmarks). This can be applied to reduce the number of fault injections required to characterize behavior by at least two-thirds for the benchmarks we examine. The profile enables future work in hardware fault injection for caches that avoids the biases of existing techniques.

CCS Concepts: • **Hardware** → **Transient errors and upsets**; • **Computer systems organization** → *Embedded software*; *Reliability*; • **Software and its engineering** → *Software fault tolerance*;

Additional Key Words and Phrases: Cache faults, cache profiling, single event upset

ACM Reference format:
James Marshall, Robert Gifford, Gedare Bloom, Gabriel Parmer, and Rahul Simha. 2021. Precise Cache Profiling for Studying Radiation Effects. *ACM Trans. Embed. Comput. Syst.* 20, 3, Article 25 (March 2021), 24 pages.
https://doi.org/10.1145/3442339

1 INTRODUCTION

The use of commodity processors in radiation environments raises issues with existing radiation mitigation solutions and testing methodologies. Caches, in particular, play a large role in radiation behavior, yet are mostly neglected in current literature due in part to a lack of hardware interfaces that allow direct manipulation of cache data. This work helps to fill this gap by allowing cache

Authors' addresses: J. Marshall, G. Parmer, and R. Simha, George Washington University, Department of Computer Science, 800 22nd St NW, Washington, DC 20052; emails: {jcmarsh, gparmer, simha}@gwu.edu; R. Gifford, University of Pennsylvania, Department of Computer Science, 3330 Walnut St, Philadelphia, PA 19104; email: rgif@seas.upenn.edu; G. Bloom, University of Colorado Colorado Springs, Department of Computer Science, 1420 Austin Bluffs Pkwy, Colorado Springs, CO 80918; email: gbloom@uccs.edu.

1539-9087/2021/03-ART25 $15.00
https://doi.org/10.1145/3442339

behavior to be traced at an instruction granularity, which allows cache vulnerability to be measured and enables new avenues of research.

Microprocessors are sensitive to electrical noise [1] and ionizing radiation [2] that may result in transient faults. There are several ways that these faults may be expressed, with one possibility being a bit-flip in which the electrical state of a component is altered enough to change the interpretation of a bit from a 0 to a 1 or vice versa. Traditional solutions such as shielding, duplication, or radiation hardening of the hardware have proven successful in high-radiation environments [3, 4].

Robust hardware mitigation solutions may not be available for smaller systems with stringent limits on budget, power, and volume, such as CubeSats [5], which are growing in popularity [6]. In place of traditional solutions, CubeSats often rely on hardware watchdog timers to detect a radiation-induced lock-up of their commodity processor [7]. Software-based radiation mitigation techniques such as process-level redundancy [8] and code-level changes such as those used by *d*OSEK [9] may be able to provide increased fault tolerance.

Testing and evaluating these software measures is difficult: effectiveness is dependent upon the hardware's radiation characteristics, hardware configuration, and the software workload [10]. In-situ testing is often impractical when the intended environment is space, and radiation testing is time-consuming, difficult, and expensive [11]. An alternative is hardware-based fault injection, which either requires a significant amount of instrumentation or is limited to targeting interfaces made available by the hardware [12].

Fault injection may be performed on a simulated target. There is a large range of simulation options that vary in how detailed the level of abstraction provided is, from flip-flop to source-level models. In general, the lower the level the abstraction is, the more accurate the fault injection will be [13]. The simulators are designed to be accurate to the level of abstraction that they target, such as an instruction set architecture (ISA), but the implementation may differ from the target hardware. In the case of caches, simulators may provide limited options such as the number of levels, associativity, and size, whereas real caches with these same attributes can differ due to features such as merge buffers, prefetching, and line-fill policies. Simulators may also have fidelity issues with respect to cache reliability, as we explore in Section 2.1.

Much of the research into fault injection assumes a fault model that excludes the memory hierarchy [13–17]. These works cite the availability of protection measures such as Error Correcting Codes (ECCs) as justification. In Section 3, we present the case that the large contribution caches make to a processor's *radiation cross section* (a measure of the probability that radiation will cause a fault) and the occurrence of multiple bit single word upsets casts doubt on this assumption in the context of commodity hardware.

The cache profile tool we explore here collects information using the commonly available JTAG interface, named after the Joint Test Action Group [18] that developed it. JTAG is widely available and removes the need for special hardware or modifications. The software under test is run on the target hardware, unmodified with few exceptions such as the addition of assembly labels to mark the section that will be profiled.

This article presents a cache profiling tool that enables users to gather utilization and vulnerability data for programs. These metrics can then be used to predict how the program will perform under radiation-induced faults and can reduce the number of fault injections required for fault injection campaigns. The contributions of this article are to

- —introduce On-Chip Debugging (OCD) Cache Tracer, a tool for collecting cache usage information at an instruction-level granularity;
- —examine benchmark cache profiles to show that cache data critical time [19] is a tighter bound than cache utilization and could better predict cache vulnerability;

— show that previous simulation results [19] that cache flushes can reduce cache vulnerability hold for different benchmarks running on hardware; and
— demonstrate how the cache profile can reduce the required number of fault injections in a theoretical campaign to characterize the behavior of the software under radiation-induced cache faults.

These contributions represent the currently realized benefits of collecting cache information with the ultimate goal of reducing the vulnerability of software running on commodity processors in radiation environments. This tool helps reach that goal by allowing precise cache usage information to be gathered directly on the target hardware, which informs decisions about the use of cache settings and software workload. The information may be applied in future work implementing fault injection tools that can inject single and multi-bit faults while overcoming the bias issues of current debugger-based fault injection techniques when targeting cache faults [20–22], described in Section 3.1.

2 RELATED WORK

There are several methods of evaluating how a system will behave when radiation-induced faults such as a single event upset (SEU) occur, ranging from analytical to exposing the system to radiation sources.

Rehman et al. [23] estimate the reliability of an embedded system by calculating an "Instruction Vulnerability Index" for each instruction in an application. These values are then composed into an application vulnerability index when combined with information about the likelihood and frequency for each instruction to execute. The index depends on accurate measurements of instruction vulnerability and an accurate model of the hardware and software of the system. The evaluation of complex components such as caches would be a difficult addition as the cache state depends on all previous memory instructions.

Software implemented fault injection (SWIFI) [24] covers a range of techniques that do not require hardware modifications to inject faults. The host OS may be used to inject the fault, or the source of the target program may be modified. These techniques are limited to injecting faults at the interfaces made available at the software level targeted.

We divide the remaining methods into simulation and hardware-based techniques.

2.1 Simulation-Based Evaluation

Simulation-based approaches use a simulator in place of the target hardware to make it easier to study different aspects of the system, including reliability. The simulator may provide varying levels of abstraction, from a model based on a hardware description language to higher levels of abstraction such as the ISA. These approaches are naturally limited to the hardware simulated, and the accuracy of the results depends on how closely the hardware model matches the target hardware.

These approaches typically execute a golden run first: the software is executed to completion with output or the final system state recorded. Faults are injected in subsequent runs, and the output or final state is compared to the golden run to detect faults. FAIL* [25] uses a full system simulator such as Bochs or Gem5 and covers the entire fault space using pruning methods and experiment parallelization to make the task tractable. The Dynamic Robust Single Event Upset Simulator (DrSEUs) [22] performs injections in a probabilistic fashion. It is built on top of the Simics simulator and uses checkpointing to segment the run, inject faults, and compare system state across runs. The hardware model may need to be extended to add new components like a cache model, as the latest DrSEUs [26] work did to allow fault injection into the cache at specified

points in the execution. The cache model is simplified with features such as cache optimizations omitted. The fault injection results obtained using the cache model have not been compared to real radiation experiments or other injection approaches, so it is difficult to verify how accurate it is.

Using a simulation-based approach introduces two concerns: availability and fidelity. Concerning availability, a simulator with related models may not exist for the target hardware, be cost-prohibitive, or license restricted. This may be an issue for teams that are using commodity hardware because they are often resource-constrained.

Several issues may impact the fidelity of simulator-based approaches. The first is a compromise between accuracy and performance: in general, a lower level of abstraction results in a more accurate simulation at the expense of performance. The result is that simulators often approximate aspects of the target hardware and errors may persist even after validated and tuning, as shown by Gutierrez et al. [27] for performance modeling. In a comparison between simulation-based fault injection tools [28], the simulator used impacted fault injection outcomes *more* than the target architecture. The complexity of simulations and hardware models also introduces opportunities for programming and modeling errors to impact results, which have misled researchers investigating performance [29] in the past.

We have decided to pursue an approach that directly uses the target hardware instead of a simulator to partially address these concerns. JTAG is widely supported in commodity hardware and is accessible by a variety of hardware devices and open source software such as OpenOCD. Using the target hardware reduces the complexity of the system by removing the need for a simulator and hardware model, reducing the opportunity for modeling approximations and software errors to skew results. It eliminates the need to validate and tune the hardware model to match the target hardware.

Fault injection may be performed on virtualized architectures such as LLVM. LLFI [14] allows injections into LLVM's intermediate representation; a language that retains high level features such as function and variable names while incorporating assembly detail such as addresses and registers. Injections at this level have been shown to accurately re-create faults that cause silent data corruption at the cost of accuracy in representing crash causing faults [17]. Recent LLVM-based work has explored the effects of multiple bit upsets [16] and the propagation of data corruption [15]. Neither of these works considers the cache.

Asadi et al. [19] used SimpleScalar to simulate the cache behavior of SPEC benchmarks. This was applied to testing if using cache flushes with set periods (10k, 100k, and 1 million cycles) can decrease cache vulnerability using their metric for critical words. The first work looked at the L1 caches; later work looked at the L2 cache [30]. Our cache profiling tool is capable of providing the same critical word information, but on the target hardware without any of the concerns associated with simulation-based approaches. We have confirmed that Asadi's work holds for a subset of MiBench running on hardware (discussed in Section 6.3).

2.2 Hardware-Based Evaluation

Field Programmable Gate Arrays (FPGAs) may be used to synthesize a processor with saboteurs that can inject faults [31]. This approach allows for accurate fault injection but requires that the hardware description of the processor is available.

The use of OCD interfaces to inject faults was first explored using platform-specific techniques such as BDM [32]. Later work focused on the more widely supported JTAG standard, such as Portela-Garcia et al. [33], which injected faults into the main memory and CPU registers. Faults are injected by using JTAG to halt the processor, alter a register or memory value, and resume the execution. Heinig et al. [20] extended JTAG-based injection to be instruction-aware: code sections and instruction types are targeted using the program counter and decoded instruction. While fast

and effective at testing the system's response to specific faults, instruction-aware injection does not accurately model the likelihood of a fault being expressed, a limitation that the authors note.

Some previous work neglected the cache as a fault injection target because the JTAG interface on most processors does not allow direct manipulation of data stored in the cache. To approximate cache faults, faults may be injected into load statements. This leads to bias issues, which Wulf et al. [21] attempted to remedy by distributing fault injections uniformly over load instructions, which falls short of accurately simulating real cache behavior. Their goal is to ensure better coverage of the realm of possible faults, not evaluate cache vulnerability. No consideration is made for the actual cache state (a load that results in a cache miss should not have a fault injected), nor is a uniform distribution of faults across load instructions desirable for evaluating cache vulnerability.

The cache profiling tool presented here uses an on-chip debugger to track the cache state. This article focuses on the immediate utility of the cache profile; future work can use the profile when selecting fault injection targets to avoid the bias issues covered in Section 3.1.

Most cache profiling tools are concerned with performance and seek to be statistically representative of cache behavior. They are not concerned with accuracy at an instruction granularity. For example, Cachegrind [34] (based on the Valgrind framework) uses a simplified cache model with limited settings such as size and associativity. These profilers are intended for performance tuning and are insufficient for evaluating cache vulnerability because they are focused on cache utilization and hit-rate. Our approach tracks the *critical time* of data in the cache [19]: the period from when data enters the cache until it is last utilized, during which a cache fault will propagate beyond the cache. Data in the cache that is not read before being overwritten will increase the cache utilization, but not the vulnerability (as the data may experience a fault but will be overwritten before use).

More recent cache profiling tools such as Alleria [35] are capable of extracting memory access traces. While the traces produced by some of these tools include physical addresses and values, they do not track cache hits and misses, and would thus need to be used in conjunction with a cache simulator to produce the same information our tool provides.

3 BACKGROUND

A number of commodity processors with caches have been used in space in recent years [36], typically on low-cost missions in low Earth orbit. Among these processors, parity checks are very common, especially for L1 caches. ECCs, typically Single Error Correction Double Error Detection, are also available, primarily for L2 caches.

The use of caches is significant in radiation environments because they can greatly alter a processor's radiation cross section. For example, radiation experiments with Zynq-7000 processors show that when caches are enabled, there is an order of magnitude increase in the radiation cross section [37, 38].

Disabling caches may appear to be an effective way to mitigate this risk so long as the associated performance degradation is acceptable. However, in some cases, this may *increase* the chances of a fault occurring while a task is being performed: the reduction in radiation cross section is negated by the additional time needed to complete a task. Santini et al. [39] demonstrated that for their matrix multiplication workload, disabling all caches reduced the radiation cross section, but running with the L1 caches enabled resulted in a higher mean workload between failures.

With caches enabled, hardware protection measures such as ECC and memory interleaving can protect from some faults. ECC often implements Single Error Correction Double Error Detection, in which a single bit-flip in a memory word can be recovered from and two bit-flips will cause a hardware exception. Memory interleaving reduces the spatial locality of cells that make up a

logical word of memory, thus reducing the chances that a radiation strike will impact multiple bits in the same word.

Despite these protections, multiple bit upsets (MBUs) in which multiple bits in the same word are flipped have been observed in radiation experiments. Tambara et al. [38] report that 16% of observed events in the static random-access memory (SRAM) used to configure the Zynq-7000's FPGA were MBUs, and Wirthlin et al. [40] show that at the highest energy level tested, 8% of radiation events were MBUs with 3 or more bits impacted in the configuration memory of a Kintex-7 FPGA (which uses bit-interleaving). Hands et al. [41] further demonstrate that ECC can be overwhelmed despite bit-interleaving in SRAM chips. MBU rates will likely increase as the feature size of SRAM is reduced [42], and there is evidence that MBUs may be under-represented in radiation studies that do not consider different angles of incidence for particles [43]. The possible occurrence of 3-bit MBUs means that cache faults should be considered when estimating the absolute fault rate of a processor and workload in a radiation environment.

The intended environment for these processors is terrestrial, which has (thankfully) a lower fluence of high energy particles than low Earth orbit. Hardware protection measures have associated costs such as complexity, silicon space, and performance; manufacturers are likely to implement only what is necessary for reliable operation on Earth.

This article is primarily concerned with evaluating the contribution of the workload's cache usage to radiation behavior. We assume that backing memory has sufficient protection, as more robust protections may be used because latency and space requirements are more relaxed than for on-chip memory.

3.1 Bias in OCD-Based Fault Injection

Caches present some challenges to OCD-based techniques due to the limited access granted by interfaces such as JTAG. Without direct access to the cache, it is difficult to determine the cache state at the selected time of injection. This tool alleviates these issues by accurately tracking the cache state throughout a program's execution.

Previous OCD-based injection techniques have focused on injecting faults into processor registers. The injection itself is straightforward: JTAG allows you to halt the processor, read a register value, write an altered value back to the register, and resume execution. For example, a transient fault that impacts the arithmetic logic unit may be simulated by altering the result as stored in a CPU register. A transient fault that impacts a data bus may be simulated by altering the register storing the data before a send or after a receive.

Some transient faults may result in persistent errors. For example, a flipped bit in a configuration register may render a device unusable until reset. If these configuration registers are accessible by the JTAG interface, then these faults can be injected as well. Persistent errors that are caused by faults in components not accessible by JTAG, however, are more difficult to simulate.

Caches are typically not directly accessible to the JTAG interface outside of a handful of configuration options. This difficulty is dealt with by altering the data with the simulated fault after it is placed in a register by a `load` instruction. Injecting a single fault into a CPU register does not always accurately reflect the vulnerability of a cache. Cache faults may persist for an extended amount of time and impact multiple instructions. This is because caches by their nature store data for an uncertain amount of time and may be read many times before being updated.

The halt, inject, resume strategy used by prior JTAG-based injection work [20] has another potential flaw: it may bias the fault injection to target certain data depending on how targets are selected. Previous implementations would select a random time to inject the fault to simulate the random nature of radiation-induced faults. The fault is injected at that time, or the next occurrence

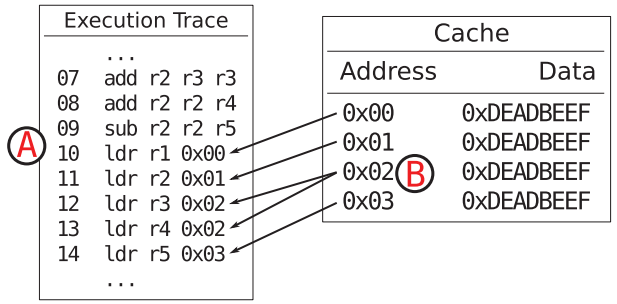

Fig. 1. This simplified program and cache diagram illustrates potential bias in injecting faults via JTAG. A: injection on the *next* `load` after the selected injection time will result in a disproportionate number of faults injected on line 10 compared to line 11. B: lines 12 and 13 both read from the same address, an effect that is ignored without considering the cache state.

of an instruction of interest (in the case of caches, `load` instructions). For a cache, this will bias injections to data that is loaded often or loaded first, whereas the vulnerability of data in the cache is dependent upon how long the data is resident.

Figure 1 demonstrates two possible issues when injecting simulated cache faults. In example *A*, line 10 is a `load` instruction that is preceded by several non-load instructions and immediately followed by another `load` instruction. The first `load` instruction is more likely to be the first `load` executed after the processor is halted, and will thus have a disproportionately large number of faults injected. This may be incorrect behavior: if line 10 loaded data from backing memory while the data used in line 11 was resident in the cache, then the bias is the opposite of what is desired. Example *B* relates to lines 12 and 13, which both load data from the same address. If this data is resident in the cache and a simulated fault is injected prior to line 12 executing, then *both* `load` instructions need to be modified to correctly simulate the fault. Tracking the cache state is vital to accurately simulating faults: the corrupted data may be overwritten by `store` operations before being read, evicted due to collisions, or remain in the cache for extended periods with many reads.

Wulf et al. [21] partially addresses these issues; faults are distributed in a uniform fashion amongst `load` instructions, and faults will be injected on all following loads from the same address for a configurable number of cycles to simulate data persistence in the cache. This ensures that all instructions may have faults injected during a fault injection campaign; however, it potentially injects faults in situations that should not have a fault injected. For example, a `load` instruction that results in a cache miss should not have faults injected: the data in the cache was replaced by data from the backing memory. Corrupted data in the cache may be benign if it is evicted (without being written back to backing memory) or overwritten by a `store` instruction.

Our approach avoids these artificial biases and could be used to improve the accuracy of simulated cache faults. We track the cache accesses of a program so that we can determine the cache state at any point in the execution. This allows us to select the instructions to inject faults into by starting conceptually with the cache itself: selecting a cache set in which the fault is to be simulated and working forward to the eventual effects (or lack thereof) in execution. We believe this design best emulates real-world scenarios where radiation strikes a random location in the cache.

By working from the cache forward, we can avoid artificial biases as well as account for benign faults. A fault can be rendered benign at the cache level in several ways: the fault may be overwritten before it is read, it may have occurred in an invalid line, or the data may be evicted due to cache pressure. Accounting for benign faults is essential to accurately measure the effectiveness of software mitigation techniques because increasing the ratio of benign faults to expressed errors is a viable mitigation strategy.

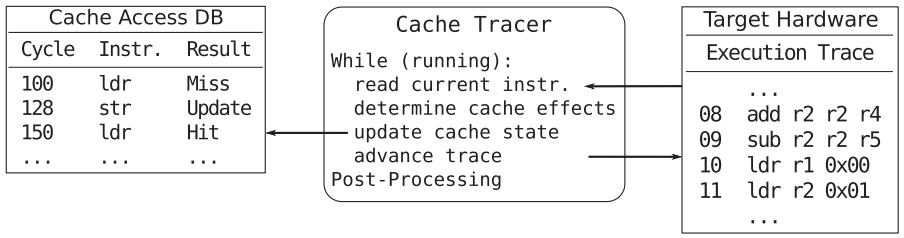

Fig. 2. The high-level design of OCD Cache Tracer has three primary components: a database to store cache accesses, the Cache Tracer program, which coordinates the data collection, and the execution trace of the program on the target hardware. The primary loop of Cache Tracer is to read the current instruction of the execution, determine how that line interacts with the cache, store the results in the cache access database, and advance the execution to the next line.

4 DESIGN

This section describes the design of OCD Cache Tracer and the factors that influenced that design. OCD Cache Tracer collects fine-grained cache usage information that allows us to understand how programs use the cache and how programs may react to cache faults.

Caches store data for an indeterminate amount of time. Data that is corrupted in the cache, such as by a radiation-induced upset, may not propagate beyond the cache depending on the state of the cache: the corrupted data may be invalid, or overwritten or evicted before use. For this reason, the cache usage information must include how and when data enters and leaves the cache. This allows us to determine the critical time of data in the cache.

Gathering the data needed to measure the critical time is subject to commercial constraints. Projects that use commodity hardware may have chosen that hardware in part due to the lower cost of it compared to radiation-hardened processors. Our solution should be as low-cost and widely available as the commodity hardware we are targeting. This makes using the target hardware preferable to simulation-based solutions. Using the target hardware directly has the added benefit of being as close as possible to the flight configuration, which is important because radiation behavior depends on both hardware and software [10]. It allows the users to abide by the best practice of "Fly as you test, test as you fly," [44] and alleviates the need to show that the tested system is functionally similar to the final product.

For these reasons, we have decided to use the target hardware itself through an OCD interface that is commonly available on commodity hardware. This interface allows direct access to some hardware components such as CPU registers and can also be used to control the execution of the processor (such as halting it to allow inspection). This is all done in a manner that is largely transparent to the software running on the processor. Using an OCD interface allows our tool to view the execution trace of a program on the target hardware with almost no modifications to the software under test.

Figure 2 shows a high-level view of our design: OCD Cache Tracer steps through the execution of a program on the target hardware using an OCD interface. The target program's execution is stepped through one instruction at a time. Cache effects of each instruction are observed and added to a database to allow for easy processing. For every `load` and `store` instruction, the following are recorded:

—The current cycle count of the program.
—For `load` instructions, whether it resulted in a hit or miss.
—The instruction address from the Program Counter.

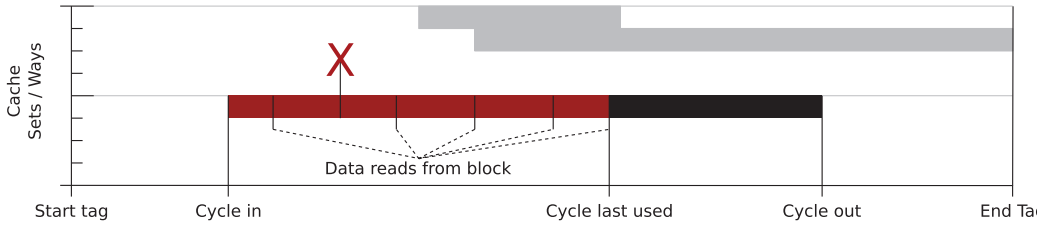

Fig. 3. The cache profile's view of critical data in the cache. The *cycle in* is recorded when data enters the cache (either via a store or load miss), and all subsequent load instructions are recorded. The last time the data is read is marked as *cycle last used*; the data is no longer critical after this point. The label *cycle out* marks when the data is evicted or overwritten by a store.

—The instruction, both raw and decoded.
—The target destination or source address for the data.

A database is preferable for tracking the cache accesses at an instruction granularity due to the large number of memory accesses made by even small programs. This database will contain every cache access the program makes but is not in a convenient form.

4.1 Cache Profile

The cache profile is an updated version of the cache access database that adds tables to track valid data throughout the program's execution. From the cache profile, we can determine the state of the cache at any point in the program's execution, which allows us to reason about possible impacts of cache faults.

Figure 3 is a graphical representation of the cache profile for a simplified example. The y-axis shows two cache sets broken into ways (four each); the x-axis shows the time in CPU cycles. Valid blocks in the cache are represented by rectangles, with critical time for the block denoted in red. Individual accesses to the data are marked with ticks. Data enters the cache (becomes valid) on a missed load or a store, which is marked as "cycle in." Subsequent accesses (load hits) are tracked, the last of which is marked as "cycle last used," after which the data is no longer critical. Finally, "cycle out" denotes when the data leaves the cache, from either cache eviction or the data being overwritten by a store.

The table of valid data blocks is added to the cache access database using Algorithm 1. The input is the list of instructions that affect cache state and related data that is collected by stepping the execution of the target program. Each entry tracks the cache line (set and way), cycle in, cycle last used, and cycle out. `CreateNewBlock` uses the round-robin scheme for selecting the cache way. A table of cache accesses is also created to link each instruction to the valid cache block that it reads from.

The cache profile tables may then be examined to calculate cache metrics for the program such as the proportion of critical data (as this work examines), or to enable accurate fault injections. To inject a fault in the cache block at a specific time, such as the "X" in Figure 3, all following instructions that access the corrupted data can be easily retrieved. In our example, there would be four instructions: one for each tick mark to the right of the "X," including the one corresponding to cycle last used.

Our approach allows the target bit and time for a fault to be selected randomly, which prevents the bias issues of previous approaches (Section 3.1). For example, injecting on the next load after the injection time ignores if that load accessed cache and biases injections based on the

preceding questions (example A in Figure 1). Our approach handles injecting a fault into multiple load instructions that access corrupted data (example B in Figure 1).

ALGORITHM 1: Build the cache profile from the table of instructions generated by the cache tracer.

```
Data: Table of instructions that access cache
Result: Table of valid data blocks
foreach instruction do
    /* Instructions are ordered chronologically by cycle                    */
    set, tag, cycle = RetrieveCacheAccess(instruction);
    /* Returns None or a block for the set/tag which is not closed          */
    currentBlock = FindExistingBlock(set, tag, cycle);
    if store instruction then
        if currentBlock then
            /* Close the existing block, data was overwritten by store      */
            CloseBlock(currentBlock, cycle);
        end
        /* Create a new block, store writes to cache                        */
        CreateNewBlock(set, tag, cycle);
    else
        /* load instruction                                                 */
        if currentBlock then
            /* Load hit: add access to block, data read from the cache      */
            UpdateBlock(currentBlock, set, tag, cycle);
        else
            /* Load miss: create a new block, data loaded from memory       */
            CreateNewBlock(set, tag, cycle);
        end
    end
end
```

4.2 Assumptions

This work requires several assumptions to be made. The program under test executes deterministically, is single-threaded, and interrupts are disabled. Deterministic execution is necessary for the generated cache profile to give accurate information about future executions, and is a common assumption in fault injection tools [25, 26]. While some OCD-based tools can likely handle non-deterministic execution, they suffer from the issues discussed in Section 3.1 when simulating cache faults.

The cache profile is generated by stepping the target processor; we assume that this execution is equivalent to normal operation with respect to cache behavior (examined in Section 6.1). Stepping the processor causes the processor to enter the debug state after each instruction is executed. We collect the instruction and counter information used for the cache profile while in the debug state and then resume execution for the next instruction. Entering the debug state causes the instruction pipeline to be flushed, and the completion of memory operations may be confirmed with the Debug Status and Control Register.

As a consequence of flushing the instruction pipeline, we are unable to observe the effects of pipeline stages individually. Fine-grained timing information is lost and we view the instruction as atomic. This abstraction is unlikely to impact the results when measuring the life of cache data

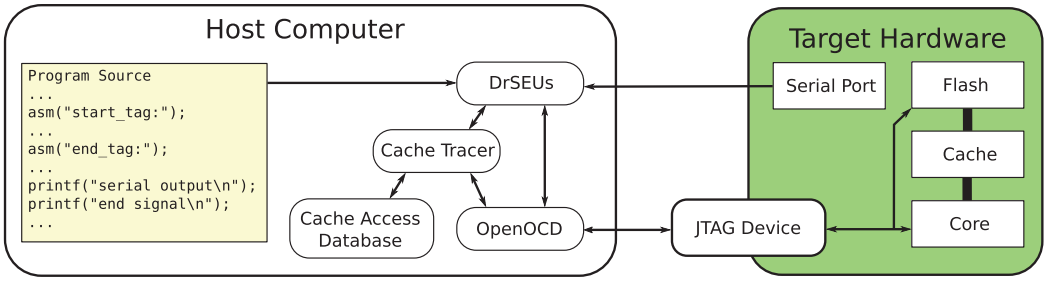

Fig. 4. An overview of the primary components needed for running OCD Cache Tracer. The host computer runs DrSEUs, which uses OpenOCD/JTAG to load the test program and controls execution. DrSEUs records and monitors the test program output through a serial port. OCD Cache Tracer is integrated with DrSEUs and creates a database of cache accesses by stepping the test program's execution and monitoring cache accesses through OpenOCD/JTAG.

over an entire program. Instruction execution is performed in-order, and we are unable to account for the effects of speculative execution.

The cache must also be configured to be write-through: a `store` writes to backing memory and cache at the same time. This prevents data from sitting in the cache before being written to backing memory and reduces the critical time for data as data corrupted during this time by an SEU may propagate into an error. Write-back caches may be supported in the future assuming that the write to backing memory occurs upon data eviction caused by an instruction we observe.

OCD Cache Tracer requires a mechanism to differentiate read-hits from read-misses at an instruction granularity (we choose to use performance counters; see Section 5). This allows us to observe the cache effects of each instruction to populate the cache profile. One could forgo this step by using a conceptual cache model for this information and using the cache tracer only to record cache-related instructions. For example, `load` instructions are recorded, but not whether the `load` resulted in a hit or miss. The validity of the results will be dependent upon the accuracy of the cache model. Our first experiment (Section 6.1) examines the differences between a conceptual model and the target hardware.

5 IMPLEMENTATION

The goal of OCD Cache Tracer is to give us a clear picture of what is happening in the cache. Given the limited visibility into the cache state, we build this picture by observing cache behavior during a golden run of the target program. Conceptually this is straightforward: the program is executed one instruction at a time, and the instruction and hardware state is examined to see if it loads data from or stores data to memory. The instructions that access memory are stored in a database for later use. OCD Cache Tracer is open source.[1]

Unfortunately, collecting cache data directly from the target hardware is not trivial. The OCD interface we use, JTAG, is flexible and hardware vendors may choose what processor components to make accessible to it. Components such as configuration registers, hardware counters, and the core registers of the CPU are commonly accessible, but the cache is rarely accessible to debugging interfaces. Instead, the data must be gathered indirectly. This section details the implementation of OCD Cache Tracer and how these difficulties were overcome.

Figure 4 shows a high-level view of how OCD Cache Tracer operates. The three green rectangles represent the required pieces of hardware: a host machine, a JTAG device, and a device under test

[1] https://github.com/jcmarsh/OCD_Cache_Tracer.

(DUT). The host machine runs the software components of OCD Cache Tracer. The two primary components are DrSEUs, which coordinates the hardware and Cache Tracer, and Cache Tracer itself, which collects the cache usage data. Both components use OpenOCD to communicate with the DUT. Cache Tracer is described further in Section 5.1.

DrSEUs [22] is a fault injection framework that can coordinate and record fault injection campaigns via JTAG. It configures the target hardware, loads the test program, and records serial output from the test program. DrSEUs uses Cache Tracer during the setup phase for a fault injection campaign, and we plan to extend the fault injection capabilities to use the cache profile information.

DrSEUs was originally designed to work with simulated targets or physical targets that are running an operating system (Linux and VxWorks are supported) through a Secure Shell session. We modified DrSEUs to be able to work with targets that are running baremetal applications. Modifications involved adding functionality to DrSEUs to save the output from the DUT through a serial connection and extending the DUT reset functionality for our embedded device. DrSEUs typically searches for an output file on the DUT after detecting a terminal prompt denoting that the application under test had finished executing. Instead, we generate the output file on the local machine using an intermediary C program to read a stream of data from the serial port connected to the DUT. This prevents DrSEUs' python script from potentially dropping data if the serial buffer overflows. Resets and programming of the embedded device are done through the Xilinx System Debugger. DrSEUs is open source and our modifications are available.[2]

DrSEUs handles the loading of the test program onto the DUT and requires the user to make a few changes to the source code. Assembly labels must be added to mark the start and end of the code section that will be observed by Cache Tracer. A codeword must be printed to the serial port to signal to DrSEUs when the program ends, allowing DrSEUs to detect timeouts during fault injections.

OpenOCD [45] enables communication with the JTAG device. The host machine connects to OpenOCD via its telnet interface. OpenOCD supports a wide range of devices from single-board computers to dedicated devices such as the Digilent HS2. We chose the Digilent HS2 because it operates faster than single-board computers such as the Beaglebone Black and Raspberry Pi 3 that we used for initial development. The JTAG device interfaces with the DUT through a six pin connector to pause, resume, and query it for register information.

5.1 Cache Tracer

Cache Tracer builds a cache profile by recording every memory access and its impacts on the cache in a MySQL database. The profile can then be used to analyze cache usage or select instructions to target when simulating faults in the cache. To collect this information, Cache Tracer needs to be able to collect information about instructions that cause changes in the cache state as well as the current cycle count.

The instruction and counter information are collected during the golden run using the `step` command to halt execution after each line of the program is executed. This process is time-consuming, taking approximately 1 minute per 10,000 CPU cycles under normal execution, but only needs to be run once per target program and configuration. To speed up the golden run, we add start and end labels to the source code. The labels allow the user to target specific parts of the execution. For example, system initialization and a period of normal operation may be skipped to allow the caches to warm up.

[2] https://github.com/CHREC/drseus and https://github.com/jcmarsh/drseus.

Information about the current instruction is provided by the JTAG interface. The program counter register is directly accessible, and the instruction can be read by reading the memory at the address stored in the program counter.

The instruction is decoded into a human-readable format by OpenOCD. This step is not necessary, but greatly eases development and debugging work. Cache Tracer uses a simple interpreter to select whether the instruction can be discarded because it does not impact the cache (such as an `add` between two registers) or if further queries are needed to determine the target address of the instruction.

Determining the target address is complicated by the large number of addressing modes provided by ARM and multiple instructions that access memory. The address often uses a base addresses stored in a register along with an offset. For example, when the instruction `LDR r2, [r11, #-0x8]` is reached, the interpreter will have to identify that `r2` is the destination register, the base address is in register `r11`, and an offset of `-0x8` should be applied. Cache Tracer then needs to read register `r11` and calculate the source address correctly.

The ARM ISA has several other instructions and features that complicate Cache Tracer's implementation:

- `STM/LDM`: Store and load data to/from multiple registers.
- `STC/LDC`: Store and load data to/from the registers of a co-processor such as the floating-point unit.
- `PLD`: Pre-Load Data hints that may or may not cause cache accesses.
- `PUSH/POP`: pseudo instructions for `STMDB sp!` and `LDMIA sp!`.
- Conditional Execution: many instructions can be made conditional depending on flags set by the result of the last comparison performed.
- Multiple word lengths: many instructions have variations for multiple data sizes (8, 16, 32, or 64 bits).

Loads and stores to and from multiple registers are accounted for in the database with an entry for each register. The memory access for co-processor instructions is recorded and accounted for. Both of these features will require additional modifications when fault injection is implemented but are feasible. The hardware counters are used to reveal if the `PLD` instructions are executed. `PUSH` and `POP` instructions are directly translated. We deal with conditional execution by checking the flag register as necessary and acting accordingly. We did not observe instructions with word lengths other than 32 bits in the benchmarks we tested, so have not yet implemented support for these instruction variations. Loading data from backing memory did not require changes as the cache line as a whole is fetched.

Cache Tracer needs to be able to determine whether a piece of data is resident in the cache when a load operation is performed in order to determine the current state of the cache. This can later be used to validate that the cache model is true to the actual hardware behavior. It is also necessary to be able to start an injection campaign midway through a program's execution: we can determine if data was resident in the cache even if Cache Tracer did not observe the instruction that caused the data to be resident.

Cache Tracer uses the Performance Monitor Unit (PMU) and hardware counters to determine data residency when a `load` instruction executes; i.e., does it result in a cache hit or a miss? The Zynq-7000's L2 Cache Controller (L2C-310) provides two counters for the L2 cache, and the PMU provides six counters that can trigger on events for the L1 and L2 caches. Of these counters, we found that the L2 Cache Controller counters for L2 access and L2 hit were sufficient to determine if an instruction resulted in a hit or a miss. However, these counters register each event multiple times when the processor is stepped. One of the PMU counter activates for memory accesses to

the L1 data cache or higher, which does not allow us to discriminate between L2 hits and misses, but has nearly a one-to-one relationship with the number of instructions that access the cache (it is complicated by instructions such as `LDM` that may cause multiple accesses).

The cycle count is retrieved from the PMU, which provides a counter with single-cycle granularity. Our original implementation used this counter set to determine how long an instruction takes to execute. From the execution time, we reasoned, one could determine if the instruction resulted in a cache hit or miss. However, stepping the processor one instruction at a time causes the pipeline to be flushed. This results in each instruction taking a minimum of 14 cycles, which is long enough to mask L1 hits. The cycle timing approach may work well for less heavily pipelined processors with more simplistic memory hierarchies. We decided to use L2 counters instead of the cycle counter to differentiate cache hits and misses so that we will be able to target L1 caches in the future.

Our implementation targets the 32-bit ARMv7-A ISA but should be applicable to other ISAs with modifications. Since we are tracking memory accesses, load/store architectures will be easier to adapt, as will simpler ISAs. A new instruction interpreter will be needed, as well as a way to discriminate between cache hits and misses at the cache-level being examined.

5.2 Limitations

Our work is limited by the black-box nature of many cache implementations and what is visible to the JTAG interface. Examining cache behavior is further complicated by the range of configuration options available.

The data collected during the golden run is limited in several respects:

- There is no knowledge of cache state that results from instructions outside of the start and end labels.
- Cache residency is determined using hardware counters while executing the program in a non-standard way (stepping).
- The counters do not inform us as to how data arrived at or departed from the cache.

We make several choices to match the cache settings of our target hardware (L2C-310 L2 cache controller as implemented by Zynq-7000 processors) and simplify cache operation. Altering these choices for different hardware configurations may require improvements to our implementation. We do not deal with cache flushes or locked cache lines within the labeled section of the test program. Only the L2 cache is enabled, and that cache is set to write-allocate (a `store` that results in a write-miss will cause a line-fill), writes to backing memory are on a per word basis (as opposed to per cache line), and performance enhancements such as prefetching are disabled. We recommend that the hardware configuration tested with this tool is used as the flight configuration ("fly as you test").

Updating our implementation to accommodate more complex behavior is straightforward when the effects of the behavior are observable during processor stepping and is not disrupted by entering debug mode. Behavior that we cannot observe may be accounted for by introducing a cache model. For example, we could model the prefetching of sequential data and update our cache access database accordingly.

5.3 Use with Multi-Core Processors

The limitations and assumptions (in Section 4.2) at first appear too restrictive for the techniques here to apply to many embedded processors available today, which often have multiple cores and several levels of cache available. For example, a processor may have four cores with per-core L1 data and instruction caches and a unified, shared L2 cache. Configuring commodity processors

with a single core and only the L2 cache enabled, as this work examines, may still achieve higher performance per watt than radiation-hardened processors due to the large disparities between the technologies. To realize further performance gains, this work can take advantage of multiple processor cores and cache levels.

Extending this work to incorporate L1 caches, individually or with the L2 cache, is possible so long as read-misses and read-hits can be detected (including which cache the data is resident in). This may increase the execution time of the golden run if more hardware counters are queried after each memory operation. However, in a two-level cache hierarchy, it should be possible to infer L1 read hits using only the currently used L2 Cache Controller counters: a memory access that results in an L1 read hit will not result in an L2 access. Modifications will also be required to the cache profile database and scripts used to analyze the cache metrics, but we anticipate that these will be straight forward.

The addition of multiple processor cores complicates the matter, in part because the L2 cache is often shared by all of the cores. Our cache tracer is only capable of tracing one single-threaded program at a time and is unable to account for the effects of data shared between threads on the cache state or the non-determinism introduced by multi-threaded programs (a common limitation in this domain).

The work, however, can still be applied to systems that are running multiple programs so long as they are not using shared data. The programs may be examined in isolation with the tools presented here, and the results will be valid due to the hardware provided memory isolation between programs. The primary cache effect of running the programs together will be increased cache pressure as more of the cache is utilized with more programs running. Work by Santini et al. [37] has shown that increasing cache evictions reduces the occurrence of faults, which is to be expected: data is vulnerable in the cache for less time. The reduced critical time means that the results obtained from isolated cache traces will be the worst-case scenario: the amount of actual critical data will likely be less than that measured by our tool. A worst-case measure is valuable because it can be used to inform decisions about the reliability of the system.

6 EXPERIMENTS

To exercise our system, we selected 10 benchmark programs. Eight are the telecom and automotive benchmarks from MiBench [46]: Basicmath, Bitcount, Quicksort, Susan, ADPCM, CRC, FFT, and GSM. We selected MiBench because it has been widely adopted in the embedded community, the code is ready to compile, and standard inputs are provided. The other two benchmarks, LZO and Matrix Multiplication, were selected because they were used by Santini et al. [37] to perform one of the more in-depth studies on the Zybo-7000. Unfortunately, Santini did not release the source code or inputs used, so we selected publicly available implementations: Lempel-Ziv-Oberhumer compression (LZO) from miniLZO [47], and Matrix Multiplication from Quinn et al. [48]. Quinn has proposed a set of uniform benchmarks for use in radiation testing, but no single set of benchmarks has been widely adopted in the radiation testing community.

We reduced the inputs for each benchmark to shorten the golden run execution time to approximately 500 minutes. The parameters are shown in Table 1. In addition to shortening the inputs used, we modified the benchmarks as follows (all modifications and inputs are available[3]):

[3]https://github.com/jcmarsh/mibench_zybo, https://github.com/jcmarsh/OCD_Cache_Tracer/tree/master/jtag_eval/apps/lzo, and https://github.com/jcmarsh/benchmark_codes/tree/zybo.

Table 1. Benchmarks and Inputs Used for Tests

Benchmark	Code Source	Inputs
Matrix Mult	Quinn	16 12 × 12 multiplications
LZO	miniLZO	64 KB Kepler Data
Basicmath	MiBench auto	Small with 1,200 cubic eqs.
Bitcount	MiBench auto	3,000 iterations
QSort	MiBench auto	1,000 words
Susan	MiBench auto	Edge detection on 76 × 35 pgm
ADPCM	MiBench tele	64 KB pcm
CRC32	MiBench tele	128 KB pcm
FFT	MiBench tele	4,096 max size, four waves
GSM	MiBench tele	Encode eight au frames to gsm

—Add platform-specific code for running baremetal on a Zynq-7000.
—Mark the start and end of the benchmark's primary work with assembly tags to track cache accesses.
—Flush the cache before the primary work starts.
—Modify cache configuration and disable the L1 caches.
—Bundle input data with the executable.
—Remove all print statements from the tracked region to shorten the runtime.
—Print a keyword to indicate to DrSEUs that the benchmark is complete.

The DUT is a Digilent Zybo [49], which has a Zynq-7000 [50] system on a chip. We are running our tests on a single core of the dual-core ARM Cortex-A9 processor and do not use the FPGA. The ARM cores have separate L1 instruction and data caches with a size of 32 KB and parity support. The L1 caches do not support write-through. The L2 cache is shared, has 512 KB of data, supports parity, and supports write-through. For our experiments, we use the L2 cache with write-through enabled. We chose the Zynq-7000 as the target chip because it has undergone several radiation tests and is currently in use in low Earth orbit [51].

6.1 Experiment 1: Hardware vs. Cache Model

This experiment is designed to build confidence that stepping the processor is equivalent to normal execution and that hardware counters and cache operate as expected. From the cache documentation, we have built a model of how we expect the cache to operate based on size, replacement policy, and other cache features. The cache model is only used for comparison: our tool gathers all required information from the hardware. Ideally, the cache model would perfectly match the observed behavior of the hardware.

We examine each memory access from the golden run and attempt to guess using our model whether or not the data in question was resident in the cache. We do this by comparing the guess to the recorded cache counter values. Any difference between the two should reveal inaccuracies with our understanding of how the cache and hardware counters work.

Table 2 shows the cache behavior of the benchmarks compared to the expected behavior. "Hit" indicates that a hit was expected by our model and reported by the cache counters. "Miss" indicates that the model and counters were in agreement that the instruction resulted in a miss. "Anomalous Hit" indicates that the cache counters reported that there was a hit, but our cache model predicted a miss. The caches were flushed immediately prior to the assembly label that marks the start of the profiled section; when no flush was performed, we observe an increase in Anomalous Hits

Table 2. Memory Accesses of Each Benchmark Divided into Cache Hits, Misses, PreLoad Data (PLD) Instructions, Hits that Our Model Predicted Should be Misses (Anom. Hit), and Misses that Our Model Predicted Should be Hits (Anom. Misses)

Benchmark	Hit	Miss	PLD	Anom. Hit	Anom. Miss
Matrix Mult	349,167	2	0	0	0
LZO	193,357	61	0	1,989	0
Basicmath	67,617	26	179	29	0
Bitcount	51,013	12	0	2	0
QSort	220,346	4,001	0	0	0
Susan	267,912	105	0	5	0
ADPCM	112,639	2,054	0	0	0
CRC32	258,015	4,130	0	0	0
FFT	155,546	1,035	1021	13	0
GSM	315,677	77	367	1	0

because the instructions that loaded the relevant data occurred before the profiler began tracking. "Anomalous Miss" indicates that the cache counters reported that there was a miss, but our cache model predicted a hit. "PLD" indicates that there was a PreLoad Data instruction that did not load the data specified. The vast majority of the PLD instructions encountered resulted in no action being taken according to the cache counters.

Overall, we observed that our cache model matches the target hardware behavior very well, which indicates that the cache operates as expected when stepping the processor. For four benchmarks every memory access is properly accounted as either a Hit or Miss, and less than 0.1% of Basicmath, Bitcount, Susan, FFT, and GSM are incorrectly categorized. The only outlier is LZO with 1% of accesses incorrectly categorized. The cache metrics will be measured based on the actual cache behavior (from the hardware counters); these results tell us that these behave largely as expected. In this case, following the observed hardware behavior is the conservative choice: the data from the anomalous hits are assumed to have entered the cache before measurements began, resulting in a slightly higher amount of critical data.

The Anomalous Hits in the LZO benchmark appear to be the result of prefetching the next cache line when reading data from sequential addresses. The first read from a new cache line is expected to be a miss (as there are no previous recorded accesses) but is instead reported as a Hit by the cache counters. This behavior was unexpected: the anomalous hits occurred even though we configured the cache with optimizations, including prefetching, disabled. This discrepancy between our cache model and the hardware counters requires further investigation, and it demonstrates the difficulties with understanding hardware behavior even with simplified settings.

We did not observe any cache evictions. Given the large size of the L2 cache and the relatively low cache usage of the benchmarks, this was to be expected.

6.2 Experiment 2: Measuring Critical Data in Cache

We can measure different cache usage metrics from the cache profiles generated in the previous experiment. One metric is cache utilization: how much of the cache has valid data in it at a given point in time? Perhaps more interesting to us is the amount of data in the cache that is valid and will be used again. In other words, what is the cumulative *critical time* of the data in the cache? We refer to these metrics by the state of the data: *resident* and *critical*.

Table 3. Percent of Cache Data that are Resident, Critical (will be Read Again), or Dormant Critical (Read Again After 10K Cycles)

Benchmark	% Resident	% Critical	% Dormant Critical
Matrix Mult	1.114	0.190	0.118
LZO	23.525	9.804	6.118
Basicmath	4.171	0.252	0.170
Bitcount	0.056	0.046	0.016
QSort	19.790	9.852	9.814
Susan	2.952	1.573	1.505
ADPCM	7.852	0.033	0.005
CRC32	12.648	0.206	0.035
FFT	11.527	3.438	3.388
GSM	0.771	0.319	0.271

Cache utilization is a bound on cache vulnerability for the data section of the cache: only faults that occur in valid cache lines have a chance of propagating in the future. Critical data is a subset of utilization, as the data must be valid *and* used in the future, and serves as a tighter bound on the contribution caches make to a processor's radiation cross section. Having a tighter bound will allow for designers to better understand the radiation characteristics of their workload *before* performing a fault injection campaign (Section 7 explores how this information may be used to reduce the time required for a campaign). For this information to be beneficial, there will have to be programs for which the proportion of critical data is significantly less than resident data.

Experimental results have shown a correlation between utilization and radiation cross section [37]. For critical data to be more predictive of radiation sensitivity than resident data, these metrics will have to diverge for some programs.

Table 3 shows the percentage of cache lines that are resident, critical, and dormant critical data throughout each benchmark's run. Dormant critical data is examined in the next experiment. Figure 5 shows these numbers as proportions of the resident data to better visualize the relationships between the metrics.

We can see that the proportion of critical data to resident ranges from almost 0% for benchmarks like Basicmath, ADPCM, and CRC32 to about 50% for QSort and Susan (Bitcount reaches 80%, but uses very little cache data). This spread of ranges shows that the proportion of critical data may be significantly less than the proportion of resident data. For these benchmarks, critical data is often a much tighter bound on cache vulnerability than resident data. The two metrics are also differentiated, so one of them may be a better predictor of cache vulnerability. We are inclined to believe that critical data is the better predictor based on the fact that only critical data can propagate faults, but verifying this and the magnitude to which they differ will require further testing.

6.3 Experiment 3: Effect of Flushes on Critical Data

Asadi et al. [19] explored using cache flushes as a mechanism for decreasing error propagation using the SimpleScalar simulator and SPEC2000 benchmarks. The cache flushes reduce the critical time of data in the cache by forcing more reads from backing memory. They observed that flushing the cache reduced the vulnerability of the cache by approximately 10 times while causing less than a 10% reduction in the instructions per cycle.

Data that is read often will soon be brought back into the cache negating the benefits of this approach, but data that is read after significant time gaps will have a reduced critical time. We call data that is resident in the cache but will not be accessed for at least 10k cycles *dormant critical*.

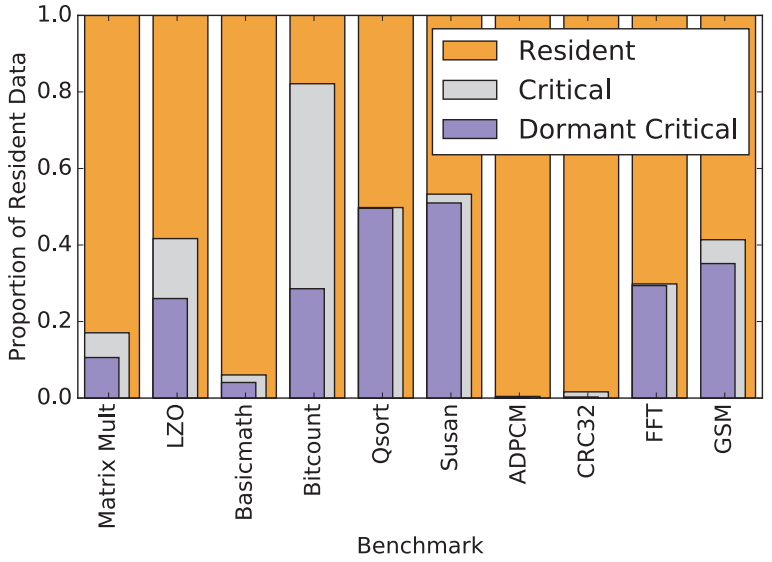

Fig. 5. Critical and dormant critical data shown as a proportion of total resident data (height is scaled proportionally; width is scaled to show subset relationship).

Ten thousand cycles was selected because that is the lowest value of times between cache flushes tested by Asadi. Dormant critical data is a subset of critical data, which is a subset of resident data.

We are interested in seeing if the amount of dormant critical data is consistently a large proportion of the critical data. This would confirm that Asadi's cache flush mechanism will indeed reduce cache vulnerability.

The data shows that a significant amount of critical data is dormant: if data is to be used again (and thus a fault may propagate), it is very likely to not be used within the next 10k cycles. Dormant critical data is a significant proportion of critical data for most benchmarks, especially those such as QSort, which have a high ratio of critical to resident data. This lends evidence to the generality of Asadi's results [19] showing that cache flushes reduce vulnerability by using a real hardware target instead of a simulator and a different set of benchmarks. A flushing mechanism could be implemented using ARM's "clean and invalidate" instructions, and further investigation is needed concerning the associated performance costs. Flushing the data reduces the vulnerability by removing dormant critical data from the cache.

7 CASE STUDY: REDUCING FAULT INJECTIONS

This case study demonstrates how we can reduce the number of fault injections required to characterize a system's cache vulnerability using an application's critical data measurement. This can save teams significant amounts of time in future fault injection campaigns.

The purpose of a fault injection campaign is to determine the probability that a fault (e.g., a bit-flip) will result in a failure of the system (e.g., corrupted data). Using brute force to determine this probability is impractical because the number of possible faults grows exponentially as the bits in the system and runtime increase. Instead, a reduced number of injections are performed to provide an estimate with a specific confidence and error margin.

Leveugle et al. [52] explored how to apply statistical methods to calculate how many fault injections need to be performed. Their technique makes the number of required fault injections tractable and is not sensitive to increases in the number of fault locations once it has become

significantly large (as any non-trivial system will). The number of fault injections is dependent on an estimate for the probability that a fault results in a failure: it is this estimate that our tool can refine.

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times \hat{p} \times (1-\hat{p})}}. \quad (1)$$

Equation (1) from Leveugle et al. outputs n; the number of fault injections required to give an accurate measure of the actual probability that a fault will become a failure (p). The formula requires the user to provide an initial guess for what p is ($\hat{p}$). N is the number of possible faults; in our case every bit of the data section of the L2 cache multiplied by the number of cycles the program executes. t is the standard deviations of the mean for the desired confidence level: $t = 1.96$ corresponds to 95% confidence and $t = 2.58$ corresponds to 99% confidence. The margin of error is e: after running n fault injections the reported result will be within $\pm e$ from the actual value with the given confidence.

The worst-case scenario is that p is 0.5: there is an equal probability that a fault will become a failure as not. Thus, without prior knowledge of p, $\hat{p}$ should be set to 0.5, which maximizes the number of fault injections that will be executed. Intuitively, this can be framed as follows: it is more difficult to determine if a coin is fair than if it is biased.

OCD Cache Tracer places limits on the possible values of p, which lets us select a value for $\hat{p}$ that results in significantly fewer fault injections being required. The limits are derived from the critical data measurement, with the assumption that only faults in cache data that will be used again can become a failure. For example, if 20% of the cache data is critical, then p may range from 0 (no faults in the critical section become failures) to 0.2 (every fault in the critical section of the cache results in a failure).

We must conservatively select $\hat{p}$ with 0.5 representing the worst-case scenario. From the limits set by the amount of critical data, we select the value closest to 0.5 to ensure that enough fault injections will be run to yield accurate results. For example, if the cache profile shows that 80% of the cache is critical, then we select a $\hat{p}$ of 0.5 since that results in the most fault injections being performed.

Cache utilization is at most 24% for the benchmarks and input data we have tested. This is in part due to the limited input sizes we can use because the cache profiling takes a long time. Critical cache data was less than half of the utilization for all benchmarks with any significant cache usage. If this trend holds for programs with higher utilization, then our technique will still offer improvements.

Refining the initial estimate for $\hat{p}$ allows us to achieve equivalent results for estimating p via a fault injection campaign with significantly fewer fault injections. This is beneficial because each fault injection may take a significant amount of time (e.g., DrSEUs using Simics takes about 5 minutes per injection [26]).

Table 4 shows the number of fault injections required to achieve 99% confidence with an error margin of 1%. The number of injections given three values of $\hat{p}$ is shown: 0.5 (required if nothing about p is known), based on the cache utilization, and based on the critical cache data (see Table 3). The injections when setting $\hat{p}$ to 0.5 are constant because the fault space is so large that the different runtimes of the benchmarks have no appreciable effect.

In all cases, the reduction in the number of required fault injections to estimate p is significant. The worst-case reduction, setting $\hat{p}$ to the proportion of resident data for LZO, results in a 28% reduction in the number of faults injected. Using the proportion of critical data (which is a subset of resident data), results in a worst-case reduction of 64% for QSort. For some benchmarks, the reduction was extreme due to the proportion of critical data being less than the selected error margin.

Table 4. The Number of Fault Injections Required to Estimate p with a Confidence of 99% and an Error Margin of 1% with Different Values of $\hat{p}$

Benchmark	$\hat{p} = 0.5$	$\hat{p}$ = Resident	$\hat{p}$ = Critical
Matrix Mult	16,587	731	126
LZO	16,587	11,936	5,867
Basicmath	16,587	2,652	167
Bitcount	16,587	37	30
QSort	16,587	10,532	5,893
Susan	16,587	1,901	1,027
ADPCM	16,587	4,801	22
CRC32	16,587	7,330	136
FFT	16,587	6,766	2,202
GSM	16,587	507	211

$\hat{p}$ is set to 0.5, resident proportion, and critical proportion (from Table 3).

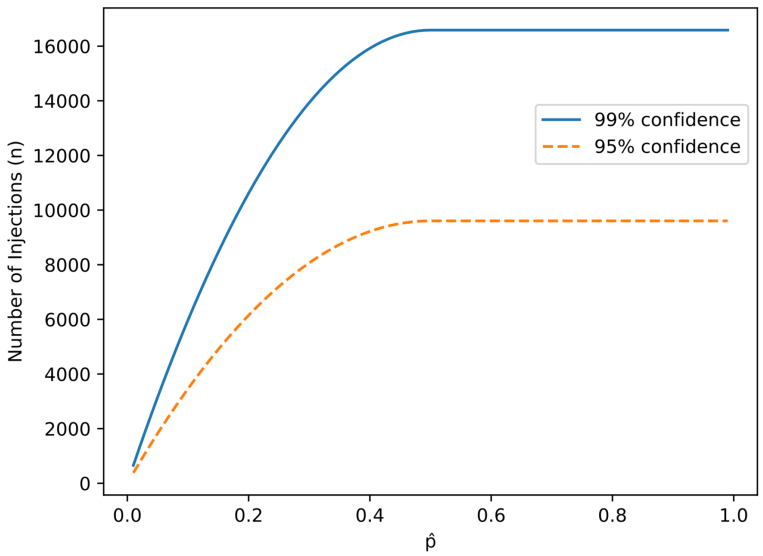

Fig. 6. The number of fault injections required (as calculated by Equation (1)) as $\hat{p}$ changes. Without prior knowledge of the benchmark, $\hat{p}$ is set to 0.5. This chart shows that the number of fault injections may be reduced when $\hat{p}$ is set based on the critical data measurement from the cache profile.

The improvement shown will still be realized if the confidence and error margins are changed. Figure 6 illustrates this with two lines representing the number of required fault injections on the y-axis and the value of $\hat{p}$ based on the critical data measurement on the x-axis. Both lines used the same parameters as we did for our benchmarks, except the dashed line has a confidence of 95% instead of 99%. Using a confidence of 95% requires fewer fault injections, but refining the estimate for p is still beneficial. The number of required fault injections is constant from 0.5 to 1 as the $\hat{p}$ must be conservatively set to 0.5 for critical data greater than 0.5, as discussed earlier.

We can use OCD Cache Tracer to significantly reduce the number of required fault injections as calculated by this method by determining the limits for the probability that a fault will result in a

failure. For the benchmarks examined in Section 6, the reduction in fault injections is at least 2/3. This represents potentially *days* less time required for a fault injection campaign due to the large number of fault injections and the time required for a single fault injection.

These results are applicable to many fault injection tools. For example, the cache profile tool could be implemented for a simulation-based fault injector to reduce the number of fault injections required.

8 FUTURE WORK

Our profiling tool may be applied to specific domains to investigate how they utilize data. For example, a wide range of fault vulnerability has been observed between different configurations of neural networks [53], with the use of buffers relating to an increase in failures in time. Our work could support specialized hardware if a JTAG interface is provided.

A high amount of critical data indicates possible vulnerability to faults; we intend to implement a JTAG-based fault injector that uses the cache profile to accurately simulate cache faults. This will allow us to observe how faults propagate into failures and will provide insights into software-based mitigation techniques.

An important avenue of future work is to explore how this tool and fault injection tools can effectively target multi-threaded applications executing on multi-core architectures. The challenges are significant, but the work is necessary to fully take advantage of the performance of modern processors.

Finally, OCD Cache Tracer may be used to validate that cache models, such as those used by simulators, match the hardware that they are modeling. The outputs of the cache model may be compared to hardware behavior observed by our tool.

9 CONCLUSIONS

This work presents a tool that collects cache usage information at an instruction granularity and demonstrates the benefits of the information. We have evaluated the tool using a set of 10 benchmarks to compare the expected and actual behavior of the L2 cache of a Zynq-7000. The information collected about *critical data* (cache data that is resident and will be used again) has yielded three results:

- Critical data is a 50% tighter bound on cache vulnerability than cache utilization for several benchmarks, and may make for a more accurate predictor of cache vulnerability (Section 6.2).
- Confirmation of prior research that cache flushes can decrease cache vulnerability (Section 6.3).
- The cache profile may be used to decrease the number of required fault injections (by at least 2/3 for the benchmarks examined) to determine fault behavior (Section 7).

The cache profile may be used to correctly target instructions when using OCD-based fault injection techniques. This is an improvement upon previous techniques that introduce artificial biases when simulating cache faults.

ACKNOWLEDGMENTS

We would like to thank the Center for Space, High-performance, and Resilient Computing (SHREC), in particular Edward Carlisle IV, for help with modifying DrSEUs.

REFERENCES

[1] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*.

[2] Whitney Q. Lohmeyer, Kerri Cahoy, and Shiyang Liu. 2013. Causal relationships between solar proton events and single event upsets for communication satellites. In *AeroConf*.

[3] Robert E. Lyons and Wouter Vanderkulk. 1962. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development* 6, 2 (1962), 200–209.

[4] Sammy Kayali, William McAlpine, Heidi Becker, and Leif Scheick. 2012. Juno radiation design and implementation. In *AeroConf*.

[5] Hank Heidt, Jordi Puig-Suari, Augustus Moore, Shinichi Nakasuka, and Robert Twiggs. 2000. CubeSat: A new generation of picosatellite for education and industry low-cost space experimentation. In *SmallSat*.

[6] M. A. Swartwout. CubeSat Database. Retrieved April 7, 2016 from https://sites.google.com/a/slu.edu/swartwout/home/cubesat-database.

[7] Rex Ridenoure, Riki Munakata, Alex Diaz, Stephanie Wong, Barbara Plante, Doug Stetson, Dave Spencer, and Justin Foley. 2015. LightSail program status: One down, one to go. In *SmallSat*.

[8] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. 2009. PLR: A software approach to transient fault tolerance for multicore architectures. *TDSC* 6, 2 (2009), 135–148.

[9] Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. 2015. dOSEK: The design and implementation of a dependability-oriented static embedded kernel. In *RTAS*.

[10] David M. Hiemstra and Allan Baril. 1999. Single event upset characterization of the Pentium (R) MMX and Pentium (R) II microprocessors using proton irradiation. *TNS* 46, 6 (1999), 1453–1460.

[11] Farokh Irom. 2008. *Guideline for Ground Radiation Testing of Microprocessors in the Space Radiation Environment*. Technical Report. Pasadena, CA: JPL, NASA.

[12] Haissam Ziade, Rafic A. Ayoubi, Raoul Velazco, et al. 2004. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.* 1, 2 (2004), 171–186.

[13] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A. Abraham, and Subhasish Mitra. 2013. Quantitative evaluation of soft error injection techniques for robust system design. In *DAC*.

[14] Anna Thomas and Karthik Pattabiraman. 2013. LLFI: An intermediate code level fault injector for soft computing applications. In *SELSE*.

[15] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling soft-error propagation in programs. In *DSN*.

[16] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. 2017. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In *DSN*.

[17] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. 2014. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *DSN*.

[18] IEEE 1149.1 Working Group. IEEE Std. 1149.1 - Standard Test Access Port and Boundary-Scan Architecture. Retrieved March 9, 2017 from http://grouper.ieee.org/groups/1149/1/.

[19] G.-H. Asadi, V. S. Mehdi, B. Tahoori, and David Kaeli. 2005. Balancing performance and reliability in the memory hierarchy. In *ISPASS*.

[20] Andreas Heinig, Ingo Korb, Florian Schmoll, Peter Marwedel, and Michael Engel. 2013. Fast and low-cost instruction-aware fault injection. In *GI-Jahrestagung*.

[21] Nicholas Wulf, Grzegorz Cieslewski, Ann Gordon-Ross, and Alan D. George. 2011. SCIPS: An emulation methodology for fault injection in processor caches. In *AeroConf*.

[22] Edward Carlisle, Nicholas Wulf, James MacKinnon, and Alan George. 2016. DrSEUs: A dynamic robust single-event upset simulator. In *AeroConf*.

[23] Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. 2011. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In *CODES + ISSS*.

[24] Raphael R. Some, Won S. Kim, Garen Khanoyan, Leslie Callum, Anil Agrawal, and John J. Beahan. 2001. A software-implemented fault injection methodology for design and validation of system fault tolerance. In *DSN*.

[25] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. 2015. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *DSN*.

[26] Edward Carlisle and Alan D. George. 2018. Cache fault injection with DrSEUs. In *AeroConf*.

[27] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. 2014. Sources of error in full-system simulation. In *ISPASS*.

[28] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, Nikos Foutris, and Dimitris Gizopoulos. 2015. Differential fault injection on microarchitectural simulators. In *IISWC*.

[29] Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. 2014. gem5, GPGPUsim, McPAT, GPUWattch, "your favorite simulator here" considered harmful. In *11th Annual Workshop on Duplicating, Deconstructing and Debunking*.
[30] Hossein Asadi, Vilas Sridharan, Mehdi B. Tahoori, and David Kaeli. 2006. Vulnerability analysis of L2 cache elements to single event upsets. In *DATE*.
[31] Luis Entrena, Mario Garcia-Valderas, Raul Fernandez-Cardenal, Almudena Lindoso, Marta Portela, and Celia Lopez-Ongil. 2012. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. *IEEE Trans. Comput.* 61, 3 (2012), 313–322.
[32] Maurizio Rebaudengo and M. Sonza Reorda. 1999. Evaluating the fault tolerance capabilities of embedded systems via BDM. In *VLSI Test Symposium*.
[33] Marta Portela-Garcia, Celia Lopez-Ongil, Mario Garcia Valderas, and Luis Entrena. 2011. Fault injection in modern microprocessors using on-chip debugging infrastructures. *TDSC* 8, 2 (2011), 308–314.
[34] Nicholas Nethercote. 2004. *Dynamic Binary Analysis and Instrumentation*. Technical Report. University of Cambridge, Computer Laboratory.
[35] Hadi Brais and Preeti Ranjan Panda. 2019. Alleria: An advanced memory access profiling framework. *TECS* 18, 5s (2019), 1–22.
[36] Alan D. George and Christopher M. Wilson. 2018. Onboard processing with hybrid and reconfigurable computing on small satellites. *Proc. IEEE* 106, 3 (2018), 458–470.
[37] Thiago Santini, Paolo Rech, Luigi Carro, and Flávio Rech Wagner. 2015. Exploiting cache conflicts to reduce radiation sensitivity of operating systems on embedded systems. In *CASES*.
[38] Lucas Antunes Tambara, Fernanda Lima Kastensmidt, Nilberto H. Medina, Nemitala Added, Vitor A. P. Aguiar, Fernando Aguirre, Eduardo L. A. Macchione, and Marcilei A. G. Silveira. 2015. Heavy ions induced single event upsets testing of the 28 nm Xilinx Zynq-7000 all programmable SoC. In *REDW*.
[39] Thiago Santini, Paolo Rech, Gabriel Nazar, Luigi Carro, and Flávio Rech Wagner. 2014. Reducing embedded software radiation-induced failures through cache memories. In *ETS*.
[40] Michael Wirthlin, David Lee, Gary Swift, and Heather Quinn. 2014. A method and case study on identifying physically adjacent multiple-cell upsets using 28-nm, interleaved and SECDED-protected arrays. *TNS* 61, 6 (2014), 3080–3087.
[41] Alex Hands, Paul Morris, Keith Ryden, and Clive Dyer. 2012. Large-scale multiple cell upsets in 90 nm commercial SRAMs during neutron irradiation. *TNS* 59, 6 (2012), 2824–2830.
[42] Eishi Ibe, Hitoshi Taniguchi, Yasuo Yahagi, Ken-ichi Shimbo, and Tadanobu Toba. 2010. Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. *TED* 57, 7 (2010), 1527–1538.
[43] David S. Lee, Gary M. Swift, Michael J. Wirthlin, and Jeffrey Draper. 2015. Addressing angular single-event effects in the estimation of on-orbit error rates. *TNS* 62, 6 (2015), 2563–2569.
[44] Cornelius Dennehy, Kenneth Lebsock, and John West. 2007. GN&C engineering best practices for human-rated spacecraft systems. In *AIAA Guidance, Navigation and Control Conference and Exhibit*.
[45] Dominic Rath. 2005. *OpenOCD: Open On-Chip Debugging*. (2005). Diploma Thesis. FH Augsburg.
[46] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4*.
[47] Markus F. X. J. Oberhumer. LZO real-time data compression library. Retrieved March 12, 2018 from http://www.oberhumer.com/opensource/lzo/.
[48] Heather Quinn, William H. Robinson, Paolo Rech, Miguel Aguirre, Arno Barnard, Marco Desogus, Luis Entrena, Mario Garcia-Valderas, Steven M. Guertin, David Kaeli, et al. 2015. Using benchmarks for radiation testing of microprocessors and FPGAs. *TNS* 62, 6 (2015), 2547–2554.
[49] Digilent Inc. ZYBO FPGA Board Reference Manual. Retrieved July 11, 2917 from https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual.
[50] Xilinx 2016. *Zynq-7000 All Programmable SoC Technical Reference Manual*. Xilinx. v1.11.
[51] Christopher Wilson, Jacob Stewart, Patrick Gauvin, James MacKinnon, James Coole, Jonathan Urriste, Alan George, Gary Crum, Elizabeth Timmons, Jaclyn Beck, et al. 2015. CSP hybrid space computing for STP-H5/ISEM on ISS. In *SmallSat*.
[52] Régis Leveugle, A. Calvez, Paolo Maistri, and Pierre Vanhauwaert. 2009. Statistical fault injection: Quantified error and confidence. In *DATE*.
[53] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. 2017. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *SC*.

Received February 2020; revised September 2020; accepted December 2020