NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Soft Error Rate Measurements through ACE Analysis in TLB Structures of CPUs

Konstantinos Marios I. Sgouras

Supervisor: **Dimitris Gizopoulos,** Professor

ATHENS

JUNE 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μετρήσεις του ρυθμού των παροδικών σφαλμάτων, μέσω της ACE ανάλυσης των TLB δομών της CPU

Κωνσταντίνος Μάριος Ι. Σγούρας

Επιβλέπων: **Δημήτρης Γκιζόπουλος,** Καθηγητής

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2024

BSc THESIS

Soft Error Rate Measurements through ACE Analysis in TLB Structures of CPUs

Konstantinos Marios I. Sgouras
S.N.: 1115202000178

SUPERVISOR: **Dimitris Gizopoulos,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μετρήσεις του ρυθμού των παροδικών σφαλμάτων, μέσω της ACE ανάλυσης των TLB δομών της CPU

Κωνσταντίνος Μάριος Ι. Σγούρας
Α.Μ.: 1115202000178

ΕΠΙΒΛΕΠΩΝ: **Δημήτρης Γκιζόπουλος,** Καθηγητής

ABSTRACT

In recent years, there has been a decrease in the minimum feature size of the transistors in integrated circuits. As a result, the vulnerability of CPU components has increased. An increasing rate of defects are present, causing transient faults and permanent faults. These faults, at some point in the execution, manifest into errors that interfere with the correctness of the execution. Thus, new ways need to be explored to (1) detect these defects (2) correct the resulting errors or prevent the execution from being affected by them (e.g., stopping the execution).

To this end, we studied the vulnerability of the TLB (Translation Lookaside Buffer) hierarchy in both the ARM and x86 ISA using the ACE (Architecturally Correct Execution) methodology which applies to transient errors. We chose this component due to its criticality in the correctness of the execution (it is vital for ensuring process isolation and security) and its frequent use, as it is responsible for caching virtual to physical translations and is accessed on every memory reference. The ACE methodology calculates the AVF (Architectural Vulnerability Factor) of an array-based component, focusing on all of its bits and being more pessimistic than fault injection (its alternative and standard in the field). We conducted this study using the gem5 micro-architectural simulator. We calculated the results using benchmarks from the MiBench suite and custom stress marks. We observed that for the x86 ISA, the average TLB hierarchy AVF from our workloads is 30.49% and the average FIT (Failures in Time) rate is 0.0226. For the ARM ISA, the average TLB hierarchy AVF from our workloads is 5.07% and the average FIT rate is 0.0414. The differences between the two ISAs can be attributed to the L2 TLB in the case of the ARM ISA, which drastically decreases the overall AVF but hinders the overall FIT rate due to its size. Finally, the results reveal the pessimistic nature of the ACE methodology.

SUBJECT AREA: Computer Architecture

KEYWORDS: soft errors, vulnerability measurements, ACE, gem5, simulation, TLB

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια, το ελάχιστο μέγεθος των χαρακτηριστικών στα ολοκληρωμένα κυκλώματα έχει μειωθεί. Ως αποτέλεσμα, η ευπάθεια των εξαρτημάτων της κεντρικής μονάδας επεξεργασίας έχει αυξηθεί. Παρατηρούμε οτι ο ρυθμός των ατελειών αυξάνεται, δημιουργώντας προσωρινά και μόνιμα ελαττώματα. Αυτά τα ελαττώματα, σε κάποιο σημείο της εκτέλεσης φανερώνονται με τη μορφή σφαλμάτων τα οποία επηρεάζουν την εκτέλεση. Για αυτό το λόγο, πρέπει να ερευνήσουμε τρόπους για να (1) ανιχνεύσουμε αυτές τις ατέλειες, (2) λύσουμε τα σφάλματα που προκύπτουν ή να αποτρέψουμε την εκτέλεση από το να επηρεαστεί από αυτά (π.χ να σταματήσουμε την εκτέλεση).
Αφορμώμενοι από αυτό, μελετήσαμε την ευπάθεια της ιεραρχίας TLB στα σύνολα εντολών ARM και x86 χρησιμοποιώντας την ACE μεθοδολογία. Διαλέξαμε τα συγκεκριμένα εξαρτήματα λόγο της κρισιμότητα τους στην ορθότητα της εκτέλεσης (είναι σημαντικά εξαρτήματα για την ασφάλεια και τις προσβάσεις στη μνήμη) και τη συχνή του χρήση. Η ACE μεθοδολογία, χρησιμοποιείται για να υπολογίσουμε το AVF κάθε εξαρτήματος βασιζόμενο σε λίστα, η οποία επικεντρώνεται σε όλα τα bit του και είναι πεσιμιστική σε σύγκριση με την ένεση λαθών (η εναλλακτική και πρότυπη μεθοδολογία). Για αυτή τη μελέτη χρησιμοποιήσαμε τον μικροαρχιτεκτονικό προσομοιωτή gem5 [7]. Υπολογίσαμε τα αποτελέσματα χρησιμοποιώντας προγράμματα από την σουίτα MiBench καθώς και δικά μας προγράμματα για έλεγχο αντοχής. Παρατηρήσαμε ότι για το σύνολο εντολών x86 το μέσο AVF της ιεραρχίας των TLB είναι 30.49% και ο μέσος ρυθμός FIT είναι 0.0226. Οι διαφορές ανάμεσα στα δύο σύνολα εντολών οφείλονται στην ύπαρξη του L2 TLB, στην περίπτωση του συνόλου εντολών ARM, το οποίο μειώνει το AVF αλλά αυξάνει δραστικά το FIT λόγο του μεγέθους του. Για το σύνολο εντολών ARM το μέσο AVF της ιεραρχίας των TLB είναι 5.07% και ο μέσος ρυθμός FIT είναι 0.0414. Τέλος, τα αποτελέσματα επιβεβαιώνουν την πεσιμιστική φύση της ACE μεθοδολογίας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αρχιτεκτονική υπολογιστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: παροδικά σφάλματα, ευπάθεια, ACE, gem5, προσομοίωση, TLB

ACKNOWLEDGEMENTS

Για τη εκπόνηση της παρούσας Πτυχιακής Εργασίας, θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου Δημήτρη Γκιζόπουλο καθώς και τον επίκουρο καθηγητή Βασίλειο Καρακώστα και τον μεταδιδακτορικό ερευνητή Γιώργο Παπαδημητρίου για την πολύτιμη συμβολή τους στην ολοκλήρωσή της.

For the completion of this Master's Thesis, I would like to thank my supervisor Professor Dimitris Gizopoulos as well as Assistant Professor Vasilios Karakostas and postdoctoral researcher George Papadimitriou for their valuable contribution.

CONTENTS

1. INTRODUCTION 13

2. BACKGROUND 15

2.1 Faults and Errors 15

2.2 Mean Time To Failure and Failures In Time 15

2.3 Architectural Vulnerability Factor 16

2.4 Statistical Fault Injection 16

2.5 ACE Methodology 17
2.5.1 ACE and un-ACE bits 17
2.5.2 AVF calculation of storage-based component 18

2.6 TLB Hierarchy 18

3. METHODOLOGY 20

3.1 The gem5 simulator 20

3.2 x86 ISA 20
3.2.1 Data Array AVF 20
3.2.2 Tag Array AVF 22

3.3 ARM ISA 23
3.3.1 Data Array AVF 24
3.3.2 Tag Array AVF 25

3.4 Stressmark 26

3.5 Automation Scripts 27

3.6 Experimental Methodology 27

4. EXPERIMENTAL RESULTS 30

4.1 Stressmarks Results 30
4.1.1 x86 Stressmark Results 30
4.1.2 ARM Stressmark Results 31

4.2 x86 MiBench Results 32
4.2.1 AVF calculation 32
4.2.2 FIT rate calculation 35

4.3 ARM MiBench Results 37
4.3.1 AVF calculation 37

4.3.2 FIT rate calculation 39

4.4 Comparing the ARM and x86 ISAs 40
4.4.1 AVF comparison 40
4.4.2 FIT rate comparison 40

5. CONCLUSION AND FUTURE WORK **41**

5.1 Conclusion 41

5.2 Future Work 41

ABBREVIATIONS - ACRONYMS **42**

REFERENCES **45**

LIST OF FIGURES

2.1 ACE and unACE cycles in the ACE methodology 18
2.2 Example TLB hierarchy 19

4.1 Stressmark Results for ARM ISA 32
4.2 Workloads AVF for Data Array (x86 ISA) 33
4.3 Workloads AVF for Tag Array (x86 ISA) 34
4.4 DTLB AVF for x86 ISA 36
4.5 ITLB AVF for x86 ISA 36
4.6 Data Array AVF ARM ISA 37
4.7 Tag Array AVF ARM ISA 38
4.8 Total Array AVF ARM ISA 40

LIST OF TABLES

3.1 x86 Simulation Configuration 28
3.2 ARM Simulation Configuration 28
3.3 Workloads x86 ISA 29
3.4 Workloads ARM ISA 29

4.1 Stressmark Results for x86 ISA 31
4.2 Stressmark Results for ARM ISA 31

PREFACE

Η συγκεκριμένη εργασία δημιουργήθηκε με την καθοδήγηση και επίβλεψη του καθηγητή Δημήτρη Γκιζόπουλου, και τη βοήθεια του επίκουρου καθηγητή Βασίλη Καρακώστα και του μεταδιδακτορικού ερευνητή Γιώργου Παπαδημητρίου στα πλαίσια της Πτυχιακής μου εργασίας.

1. INTRODUCTION

In recent years, transistors become smaller and smaller due to advances in their manufacturing process. The physical laws are now the biggest obstacles to shrinking down these devices further, and thus our scaling rate has decreased (Moore's law [27] is now not applicable). There are three reasons for manufacturing smaller devices:

(1) Transistor density, which gives us the ability to create more complicated designs in general purpose CPUs, SIMD (Single Instruction, Multiple Data) and GPU systems, and smaller devices in general. Design goals vary in different systems and include speed and energy efficiency.

(2) Due to the smaller bridge length of the transistors, the raw speed of the device has increased.

(3) Finally, smaller transistor plates lead to reduced voltage needs and thus energy efficiency.

Despite making great strides in these goals, the scaling comes at a cost. There has been an increase in the vulnerability of integrated circuits [2, 14, 19]. The problem is only tackled in specialized applications (e.g., space computing [26, 21, 48, 41]) but in commercial and cloud devices the problem becomes more and more prominent [20, 16]. Furthermore, current measurement methods often lead to misleading results [35].

During the execution of a workload, there is a chance that a fault will occur inside the computational or memory units (their causes are mentioned in Section 2.1). These faults can be split into two categories, transient faults and permanent faults. With this realization, in this work, we study the first category of faults. Specifically, we are interested in transient faults in the TLB (Translation Lookaside Buffer) structures of the CPU. The TLB is the structure responsible for buffering virtual to physical address translations (as well as required metadata e.g. process identifiers). The choice of component is based on its criticality in the correct execution of the program.

There have been some previous works that study the reliability of the address translation structures and the address translation subsystem. First of all there have been some approaches that focus on the validation of address translation structures [34, 33, 39]. Moreover there are works that focus on adding protection mechanisms in the address translation structures [12, 46]. Finally there are also approaches that tackle memory consistency taking into account the virtual memory address space [42].

To understand how vulnerable these components are, we use the ACE methodology [8, 28], which calculates the Architectural Vulnerability Factor (AVF)[29] of different components. The AVF is a metric that shows the probability of a fault in a specific component affecting the result of the execution. During the study, we explore the advantages and disadvantages of this methodology. The ACE methodology is used to calculate reliability metrics through simulation by taking into account every bit of the array (more about the ACE methodology in Section 2.5).

For all the simulations, we employ the gem5 microarchitectural simulator [7]. We conducted the study both in the ARM and the x86 ISA's TLB hierarchies, due to the diversity of fault behavior in different ISAs [15, 13]. We used various workloads from the MiBench suite as well as some stress marks that we created. We observed that for the x86 ISA,

the average TLB hierarchy AVF from our workloads is 30.49% and the average FIT (Failures in Time) rate is 0.0226. For the ARM ISA, the average TLB hierarchy AVF from our workloads is 5.07% and the average FIT rate is 0.0414.

2. BACKGROUND

As mentioned before, our goal is to use the ACE methodology to measure the reliability of the TLB hierarchy to soft errors. To proceed, we need to understand these vulnerabilities, how we can express them in a metric, the methodologies used to calculate these metrics, the role of the TLBs and why we study them.

2.1 Faults and Errors

First, we need to define what reliability problem computers are facing. During the execution of a program, there is a possibility of a fault happening. Faults are not user-visible. Some of their causes include manufacturing defects, hardware bugs or bit-flips. They are split into three categories based on their persistence: permanent, intermittent or transient. Permanent faults are always present (e.g., oxide wearout). Intermittent faults frequently appear and disappear. Usually, they are early forms of permanent faults (e.g., partial wear out). Transient faults are temporary errors, mostly bit flips or gate malfunctions caused by alpha particle strikes, neutron strikes, or electromagnetic effects.

Although faults are invisible to the user, they usually cause a user-visible error. The definition of user-visible in the context of a program execution is an alternation of its result. There are two types of errors regarding their persistence: (1) soft errors (temporary) and (2) hard errors (permanent). In this study, we are focusing on soft errors [9, 10].

Soft errors can also be categorized based on the user's knowledge of their existence. There have been proposed mechanisms which can detect (Error Correction Code - ECC) and correct a fault (e.g., Single-bit Error Correction - SEC). The Single-bit Error Correction, Double-bit Error Detection (SECDED) is the error correcting code used for standard ECC protected SDRAM [18]. If a fault is detected but not corrected, it is called Detected Unrecoverable Error (DUE). If the error detection happens at the software level (e.g. Segmentation Fault), it is called a crash. If a soft error is not detected but still affects the execution, it is called silent data corruption (SDC). SDCs are far more dangerous than crashes because the user does not know their existence. The user trusts the result of an integrated circuit, which turns out wrong, with no sign of malfunction [38, 40, 43, 17].

2.2 Mean Time To Failure and Failures In Time

To study how the faults affect the execution, we need to use a metric that defines how often they occur. The first metric is Mean Time To Failure (MTTF), which expresses the mean time elapsed between two individual faults. A similar metric is the Failures In Time (FIT), which expresses how many errors happen in a system in (10^9) hours. The FIT metric has an additive nature when it comes to calculating the FIT of a system using the FIT of its components; thus it is preferred by engineers.

$$FITrate_{system} = \sum_{i=0}^{n} FITrate_i$$

2.3 Architectural Vulnerability Factor

In this study, we focus on the vulnerability of array-based components. These components contain many storage cells; thus we focus on their bit-flips (faults on storage cells). To calculate the FIT of user-visible errors in storage cells there are three factors we need to take into account: (1) The chance a bit-flip will happen in a specific time period (intrinsic FIT), (2) the fraction of this time period that the component is vulnerable to bit-flips (Timing Vulnerability Factor or TVF), (3) the probability that a bit-flip will result in a user-visible error. The first two factors can be calculated by physical experiments taking into account the chip technology, its packaging and the environment in which it is functioning. It is worth mentioning, that a typical intrinsic FIT rate for an SRAM array bit ranges from 10^{-6} to 10^{-5} (for our calculations, we will use an intrinsic FIT rate of $5 * 10^{-6}$). The last factor is calculated using a metric called Architectural Vulnerability Factor (AVF). Due to the addictive nature of the intrinsic rate we can calculate the total intrinsic FIT rate of a component using the equation:

$$FITintrinsic_{system} = \sum_{i=0}^{n} FITintrinsic_i = FITintrinsic_bit * TotalNumberofBits$$

Finally, the FIT formula that includes the intrinsic FIT rate and the AVF is:

$$FITrate = FITintrinsic_{bit} * TotalNumberofBits * AVF$$

As mentioned before, AVF expresses the possibility that a bit-flip in a storage cell results in a user-visible error. The calculation of this metric is very complicated due to the interactions of many architectural and micro-architectural components. Thus, simulation software is used to calculate it. Furthermore, it can be split into two categories: (1) SDC AVF and (2) DUE AVF. In this study, our goal is to calculate this metric for different virtual translation components to make observations about the vulnerability. We will be using the combined SDC and DUE AVF. Finally, there are two methodologies that are widely used to acquire the AVF of a component using simulation: (1) Statistical Fault Injection and (2) Architecturally Correct Execution (ACE).

2.4 Statistical Fault Injection

The first methodology used to compute the AVF of a specific component is called Statistical Fault Injection (SFI) [25]. In this methodology, many simulations of the execution of a particular program take place. In each execution using a statistical model, a fault is injected (either in a storage cell or in an RTL component). At the end of the execution, the correctness of its result is assessed (compared with an execution with no injected faults). If the result is not correct, then an error is considered detected. Then, the AVF is calculated using the expression:

$$AVF = \frac{ErrorsDetected}{FaultsInjected}$$

SFI is the most accurate way to calculate the AVF of a component. The main issue is the time required to simulate all these executions. First, the execution needs to finish observing an outcome. Then, to have a statistically correct result, many repetitions are necessary. This creates a lengthy simulation for RTL (Register-Transfer Level) and microarchitectural simulators using performance models (which are magnitudes faster). The poor performance of this methodology leads to long AVF calculations for new components, although some works aim to reduce this latency [37]. Moreover, SFI may not consider every single bit of an array-based component (given its statistical nature). The calculation solely considers the random bits that were injected at the cycle they were chosen to be injected. Thus, despite examining the outcome of some faults until the end of the execution, some potentially hurtful faults may be omitted. To this end, the amount and the distribution of faults are of paramount importance for high statistical significance of the final measured AVF [25, 36].

2.5 ACE Methodology

The second methodology is called ACE because it uses Architecturally Correct Execution (ACE) [8, 28] principles to compute per-structure AVF. To explain the methodology, we need to define what ACE and un-ACE terms are.

2.5.1 ACE and un-ACE bits

A storage cell is considered ACE in every cycle in which a change in the storage cell's state will result in a user-visible error. During every other cycle of the execution, the bit is considered un-ACE (a visual representation of the characterization of a bit's cycles in the ACE methodology can be found in Figure 2.1). For example, let us consider an execution that lasts 10 million cycles. If, during the first 2 million cycles, a change in the storage cell's value results in a user-visible error, the bit will be labeled as ACE for these 2 million cycles. If, during the other 8 million cycles, the change in its value does not affect the execution, then the bit will be labeled as un-ACE for these 8 million cycles. The formula to calculate the AVF for this specific bit using this information is straightforward:

$$AVF = \frac{ACECycles}{TotalCycles}$$

In our example, the AVF of the bit is 0.2 or 20%. This definition of the ACE bit creates a problem. In which cycles will the bit in question be labeled ACE, when its effect on the execution's outcome is unknown? To solve this problem, the ACE methodology employs a pessimistic approach to reduce the simulation's length. It assumes that a bit is ACE when it can potentially create a user-visible error in the future. The methodology examines each stage of the ”path” in the bit's ”journey” separately, thus adopting a pessimistic approach and labeling every potential error-prone bit as ACE. Finally, the AVF calculation, despite taking into account every bit in a component, is less accurate (pessimistic) compared to Fault Injection.

2.5.2 AVF calculation of storage-based component

The AVF calculation of a storage-based, using the ACE methodology, is an additive version of the single bit's expression. First, we simulate each cycle, labeling every bit of the structure as ACE or un-ACE. It is important to note that the distinction happens based on the current stage of the bits datapath, i.e., we consider only the propagation of a potential fault. Then, we use the following expression to calculate the AVF:

$$AVF = \frac{TotalACECycles}{NumberOfBits * TotalSimulationCycles}$$

During this study, we also utilize an expression that uses the number of un-ACE cycles because it is sometimes easier to label a bit as un-ACE rather than ACE:

$$AVF = 1 - \frac{TotalUnACECycles}{NumberOfBits * TotalSimulationCycles}$$

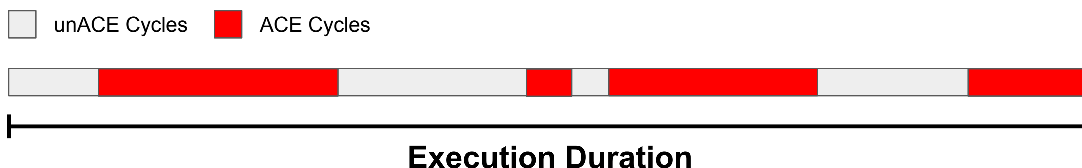

Figure 2.1: ACE and unACE cycles in the ACE methodology

2.6 TLB Hierarchy

In this study, we calculate the AVF of the components in the TLB Hierarchy. The Translation Lookaside Buffer (TLB) is a component that stores virtual to physical address translations as TLB entries. It contains two arrays: (1) a data array and (2) a tag array. The data array stores the physical frame to which the associated virtual page points. The tag array stores the entire Virtual Page Number (VPN) or part of it (depending on the associativity of the TLB) , a subset of the virtual address that is used to match with the given virtual address to determine whether the translation is in the buffer.
In a TLB, because of the common virtual address space between processes, we need to be able to differentiate between the processes' translations. So there are two ways to distinguish the processes' translations: (1) either to concatenate the identifier of the process in the TLB's tag (e.g Address Space Identifier [11], Process Context Identifier [23]) or (2) flush each time the OS performs a context switch. In either of these techniques, the TLB plays a critical role in keeping each process' data separate. A TLB can be fully associative, n-way associative or directly mapped (usually fully associative or n-way associative). It is worth mentioning that there are two TLBs in a modern system: (1) a data TLB and (2)

an instruction TLB. Finally, the TLB's purpose is to make the virtual address translation process faster.

So, what is the process of virtual translation? When a virtual address has to be translated, the hardware first checks the TLB for a potential match using the VPN. In the case it matches, the corresponding physical frame number (PFN) is used. In the case of a miss, the OS (operating system) or the hardware performs a page table walk (PTW). Its result is cached in the TLB. In modern systems, a hierarchy of TLBs is used (similar to the cache hierarchy) to avoid even more virtual address translations (usually, a shared L2 TLB is used for both instructions and data as shown in Figure 2.2). //The page table walk is a latency-inducing process. Thus here are two approaches to reduce its impact on system performance: (1) increasing the reach of the TLB by increasing the page size [22, 47] or by using range translations [31, 32] and (2) reducing the latency of the TLB misses by faster PTWs [44, 45] and caching mechanisms such as Page Walk Caches (PWC) [4].

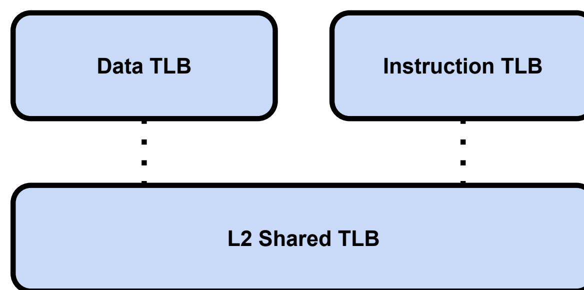

Figure 2.2: Example TLB hierarchy

Another characteristic of the TLB is its safety-critical role. If a fault were to occur in the TLB, a process could access or change the data of a different process. That has potential correctness issues for the entire system as a whole (not as a single entity running by itself as mentioned in the bibliography) and also possible security threats if these vulnerabilities are used maliciously (an example of an attack using bit flips in the TLB entries using the RowHammer attack [24] has been discovered in [49]).
Finally, it is worth mentioning that there is another type of TLB called nested TLBs [3, 6] that are used to cache the translations of virtual machines in a system. The focus of this thesis is to evaluate the vulnerability of the TLBs in native execution, i.e., a non-virtualized system.

3. METHODOLOGY

To calculate the AVF of a variety of TLB hierarchies we needed to make a few choices: (1) what simulator to use, (2) the characteristics of the simulated system and (3) what benchmarks to use. First, the simulator used in this study is the gem5 microarchitectural simulator (version 22.1.0.0). We used a default full-system simulation and a customizable TLB hierarchy. We use the ACE methodology [8, 28] applied to the TLBs. Finally, we used the MiBench benchmark suite and created our customizable stressmark to test the limits of our methodology.

3.1 The gem5 simulator

The gem5 simulator is a micro-architectural simulator that can execute full-system simulations [7]. It offers the ability to run the entire Linux kernel and a chosen Linux image and to be accessed by attaching a terminal (telnet). This simulation is called Full System simulation (FS) and is the one we are using for this study. Also, we used two of the ISAs provided: (1) x86 and (2) ARM. We wanted to measure the vulnerability of the TLB hierarchy in systems with different ISAs, so we used the two most commercially used. Since the gem5 code for TLBs is ISA-specific, we needed to add the ACE methodology to every ISA's code separately.

3.2 x86 ISA

Our goal is to calculate the AVF of the data and the tag array (for both instruction and data TLBs) separately and combine them later (we will present the computation formula of their combined AVF later).

3.2.1 Data Array AVF

To implement the ACE methodology in the data array, we needed: (1) to change the simulation's TLB entry to store some extra variables and to create a way for the TLB to keep track of every un-ACE bit. We decided to count all the un-ACE cycles instead of the ACE cycles because they were a minority and had a more defined definition, thus making them easier to detect.

The entries of the TLB required an extra field while the simulation occurred. So, for every TLB entry, we tracked the clock cycle in which they were last accessed. Next, we need to create a cycle pool. We use an un-ACE cycles pool because they are the minority and easier to calculate. It is important to mention that when we add cycles in the un-ACE pool, we add them on a per-entry basis rather than per bit. Later, we will explain how the AVF computation happens using this counter. This variable is updated in these cases:

1. During the initialization of the TLB. This initialization is redundant, since we will set the beginning of the simulation in the following case.
2. When the simulation starts. Specifically, we consider this point the first time a virtual translation happens (the first time the TLB is accessed).

3. When an LRU eviction occurs, the last accessed time of the invalidated entry is the clock cycle of the eviction (we will use this functionality in the un-ACE cycle calculation).
4. When an insert happens, the ”last accessed time” of the inserted entry is the clock cycle of its insertion.
5. When a lookup happens, if there is a hit in an entry, this entry’s last accessed time is updated. So every time a TLB entry is accessed, then its last time of access is the clock cycle of the access.
6. When a global flush happens, the ”last time of access” of the invalidated entries is when the global flush occurred.
7. When a non-global flush happens, the last time of access of the invalidated entries is the time the non-global flush occurred.
8. When a demap happens, the last time of access of the invalidated entry is when the demap happened.

Furthermore, we tracked the number of memory accesses that hit that specific entry (we used this metric for debugging).

Then we implemented the counting of the un-ACE cycles. To detect which cycles would be considered un-ACE, we needed to define several rules. It is worth mentioning that when we add cycles in the un-ACE pool, we add them on a per-entry basis rather than per bit. Later, we will explain how the AVF computation happens using this counter:

1. During the period between the last accessed time of an entry and its LRU eviction, any fault on any bit of the entry would not propagate through the execution. During the LRU eviction, the entry would be deleted (or replaced) and thus its contents would be irrelevant. So every time an LRU eviction occurs, we add the cycles from the last accessed time of the entry to the eviction cycle in the un-ACE cycles pool.
2. During the period between an LRU eviction and the insertion of an entry, any fault that occurs on the bits of the entry cannot propagate. This period’s cycles are irrelevant because the entry is invalid during this period and is re-validated during the insert. This case, in the gem5, does not usually happen because the LRU eviction occurs only when an entry is about to be inserted in a full TLB, but we add the functionality for the case of a newly flushed TLB. So, the cycles from an LRU eviction to an insertion of a new valid entry in the same slot are added to the un-ACE cycles pool.
3. When a global flush happens, for every entry, the cycles between their last accessed time and the flashed clock cycle are irrelevant (any fault during these cycles would be flushed). It is important to note that this happens for every entry (it will be important in the calculation later). So for every entry we add in the un-ACE pool, the cycles from the clock cycle it was last accessed to the clock cycle the global flush happens.
4. When a non-global flush happens, for every entry, the cycles between their last accessed time and the flashed clock cycle are irrelevant (any fault during these cycles would be flushed). It is important to note that this happens for every entry (it will be important in the calculation later). So for every entry we add in the un-ACE pool, the cycles from the clock cycle it was last accessed to the clock cycle the non-global flush happens.

5. During the period between the last accessed time of an entry and its demap, any fault on any bit of the entry would not propagate through the execution. During the demap, the entry would be deleted and thus its contents are irrelevant. So every time a demap occurs, we add the cycles from the last accessed time of the entry to the demap cycle in the un-ACE cycles pool.

6. At the end of the execution, for every entry, the cycles from the last accessed time of the entry to the end of the execution. These bits are un-ACE because they will not be used again. So, for every entry, we add to the un-ACE cycle pool the entries from their last accessed time to the end of the execution.

Next, we need to calculate the data array AVF. Considering we have a per-entry counter, the calculation happens slightly differently. First, let's look at the original formula:

$$AVF = 1 - \frac{TotalUnACECycles}{NumberOfBits * TotalSimulationCycles}$$

In this formula, we observe it includes the total number of bits and un-ACE cycles. First, let's expand the formula for our case:

$$AVF = 1 - \frac{TotalUnACECycles}{NumberOfEntries * BitsPerEntry * TotalSimulationCycles}$$

Then, we need to break down the total un-ACE cycles in per entry cycles:

$$AVF = 1 - \frac{PerEntryUnACECycles * BitsPerEntry}{NumberOfEntries * BitsPerEntry * TotalSimulationCycles}$$

This modification is possible because we consider the entire entry as one unit. This simplification is valid unless we consider the effect of the altered pointer in that clock cycle's memory snapshot. Because this calculation is outside of this study's scope, the simplification is valid. We further simplify the equation to have the final formula we used for our calculation:

$$AVF = 1 - \frac{PerEntryUnACECycles}{NumberOfEntries * TotalSimulationCycles}$$

We can observe that the size of the entry is missing for the formula. Every part of the formula has already been calculated throughout the execution.

3.2.2 Tag Array AVF

Next, we need to calculate the AVF of the tag array. We first need to keep track of when the TLB was last accessed. So, at the beginning of the simulation (in the first TLB access) and for every access since, we register the clock cycle that the access happens. To make the calculation, we need to count either the un-ACE or the ACE cycles. In this situation, we will count the ACE cycles because they are the minority and are easier to count. It is

important to mention that we will count cycles on a per-bit basis.

Next, we need to understand when a bit is considered ACE. Each time an access happens, we compare the virtual page's number (VPN) to every entry in the tag array. There are two cases in which a fault can affect the execution:

1. If the translation is available in the TLB and there is a fault in the corresponding tag entry, we are guaranteed to have a mismatch. Despite having a miss when the intended result should have been a miss this case cannot be classified as an error. After the miss, the execution will proceed with the page table walk. The resulting translation will be correct despite costing many more cycles compared to a TLB hit, and the execution will be flawless. Thus, the fault only contributed to the latency of the execution and did not affect its correctness.
2. The second case is a tag entry mismatch. Assuming only a single bit of a tag entry has changed, we need to check whether any possibly faulty entries are a match for our desired VPN. To make this comparison, we calculate the Hamming distance (how many bits need to change for the tags to be the same) between any tag entry of the TLB and the desired VPN. If the Hamming distance is 1, a single bit out of the entry was characterized as ACE since the last TLB access. Thus, we add the cycles since the last translation clock cycle to the ACE-cycles pool. This process is repeated for every entry in the tag array, so for each translation, we can contribute to the ACE-cycles pool more than once.

Now that we have calculated the ACE cycles, we need to calculate the AVF. This time, the original formula using the ACE bits is:

$$AVF = \frac{TotalACECycles}{NumberOfBits * TotalSimulationCycles}$$

This time, we need to take the entry's length into account. Each tag entry is the same size as the VPN. The VPN is the entire Virtual Address without the bits used to access the physical page (offset). In the x86 architecture, the offset is 12, and the size of the Virtual Address is 64. The VPN size is $VPN_Size = VirtualAdressSize - OffsetSize = 64 - 12 = 52$. Making the alterations in our formula, we have:

$$AVF = \frac{TotalACECycles}{NumberOfEntries * EntrySize * TotalSimulationCycles}$$

Substituting the entry size:

$$AVF = \frac{TotalACECycles}{NumberOfEntries * 52 * TotalSimulationCycles}$$

Finally, this computation happens at the end of every execution.

3.3 ARM ISA

As mentioned in the x86 ISA, we need to calculate the Data Array and Tag Array AVF separately for the entire TLB hierarchy.

3.3.1 Data Array AVF

First, we will calculate the AVF of the data array. Again, we will implement the same TLB entry attribute we created in the x86 version. The attribute marks for every entry the last time (clock cycle) it was accessed. To create this attribute, we need to set some cases in which we will update it:

1. First of all, when the first translation happens, we update every entry's "last accessed time" to the first lookup clock cycle.
2. If in a TLB lookup, we have a match, we note the matched entries "last accessed time" to be equal to the lookup access time.
3. In the case of an insert, we set the inserted entry's "last accessed time" to be equal to the insert clock cycle.
4. When a flush of a TLB entry happens, we set the entry's "last access time" to be equal to the flush clock cycle.
5. When a flush of all the TLB entries happens, we set all the entries' "last access time" to the flush clock cycle.

Next, we need to create a cycle pool. As with the x86 data array case, we use an un-ACE cycles pool because they are the minority and easier to calculate. It is important to note that we will add the cycles on a per-entry basis (the reason will become apparent in the AVF calculation formula). We put cycles in this pool based on the following rules:

1. When a new entry is inserted, the period between the last time the empty entry was accessed (which is known as its eviction time) and the time of insertion is known as un-ACE because any fault that may occur during this time does not affect the execution, since it is masked. Subsequently, we add to the un-ACE cycles pool the cycles between the eviction of the empty entry and the insert of the new entry.
2. When a flush on all entries happens, for every entry the time since it was last accessed, can be considered un-ACE because the entry is wiped during the flush. Any fault happening during that period will not propagate to become an error. So, for every entry, we add the cycles between the time the entry was last accessed and the flush clock cycle to the un-ACE pool.
3. When a flush on a single entry happens, the time since it was last accessed, can be considered un-ACE because the entry is wiped during the flush. Any fault happening during that period will not propagate to become an error. So we add the cycles between the time the entry was last accessed and the flush clock cycle to the un-ACE pool.

Finally, we need to proceed with the AVF calculation. Because we have a per-entry un-ACE cycles pool, we need to make some alterations to the formula (the same as with the x86 case). First, let's see the original formula:

$$AVF = 1 - \frac{TotalUnACECycles}{NumberOfBits * TotalSimulationCycles}$$

Then, let's expand the formula for our case:

$$AVF = 1 - \frac{TotalUnACECycles}{NumberOfEntries * BitsPerEntry * TotalSimulationCycles}$$

Then, we need to take into account the per-entry nature of our un-ACE cycles pool:

$$AVF = 1 - \frac{PerEntryUnACECycles * BitsPerEntry}{NumberOfEntries * BitsPerEntry * TotalSimulationCycles}$$

As mentioned in the x86 case, this modification is possible because we consider the entire entry one unit. This simplification is valid unless we consider the effect of the altered pointer in that clock cycle's memory snapshot, but this calculation is outside this study's scope. We further simplify the equation to have the final formula we used for our calculation:

$$AVF = 1 - \frac{PerEntryUnACECycles}{NumberOfEntries * TotalSimulationCycles}$$

Finally, at the end of the execution, we calculate the AVF of the data array.

3.3.2 Tag Array AVF

Next, we need to calculate the AVF of the tag array. We first need to keep track of when the TLB was last accessed. So, at the beginning of the simulation (in the first TLB access) and for every access since, we register the clock cycle that the access happens. To make the calculation, we need to count either the un-ACE or the ACE cycles. In this situation, we will count the ACE cycles because they are the minority and are easier to count. It is important to mention that we will count cycles on a per-bit basis.
Next, we need to understand when a bit is considered ACE. Each time an access happens, we compare the virtual page's VPN (Virtual Page Number) to every entry in the tag array. There are two cases in which a fault can affect the execution:

1. If the translation is available in the TLB and there is a fault in the corresponding tag entry, we are guaranteed to have a mismatch. Despite having a miss when the intended result should have been a miss this case cannot be classified as an error. After the miss, the execution will proceed with the page table walk. The resulting translation will be correct despite costing many more cycles compared to a TLB hit, and the execution will be flawless. Thus, the fault only contributed to the latency of the execution and did not affect its correctness.

2. The second case is a tag entry mismatch. Assuming only a single bit of a tag entry has changed, we need to check whether any possibly faulty entries are a match for our desired VPN. To make this comparison, we calculate the Hamming distance (how many bits need to change for the tags to be the same) between any tag entry of the TLB and the desired VPN. If the Hamming distance is 1, a single bit out of

the entry was characterized as ACE since the last TLB access. Thus, we add the cycles since the last translation clock cycle to the ACE-cycles pool. This process is repeated for every entry in the tag array, so for each translation we can contribute to the ACE-cycles pool more than once.

Now that we have calculated the ACE cycles, we need to calculate the AVF. This time, the original formula using the ACE bits is:

$$AVF = \frac{TotalACECycles}{NumberOfBits * TotalSimulationCycles}$$

This time, we need to take the entry's length into account. Each tag entry is the same size as the VPN. The VPN is the entire Virtual Address without the bits used to access the physical page (offset). In the ARM architecture, the offset is 12, and the size of the Virtual Address is 64. The VPN size is $VPN_Size = VirtualAdressSize - OffsetSize = 64 - 12 = 52$. Making the alterations in our formula, we have:

$$AVF = \frac{TotalACECycles}{NumberOfEntries * EntrySize * TotalSimulationCycles}$$

Substituting the entry size:

$$AVF = \frac{TotalACECycles}{NumberOfEntries * 52 * TotalSimulationCycles}$$

Finally, this computation happens at the end of every execution.

3.4 Stressmark

We decided to create a stressmark for two reasons:

1. First, we needed to evaluate the correctness of our implementation. A reliable and controllable stressmark is vital because we can compare the simulation results with our expected results.
2. We needed to check the extremes of the methodology. Our workloads are not going to express the maximum or minimum TLB vulnerability. That is why we need to fabricate a way to test the extremes and see which is the maximum (or minimum) vulnerability factor that the TLB can have.

The stressmark is a simple program that allocates M number of pages and then accesses them sequentially, for N times (we provide M, N as arguments). The pseudocode for the stressmark can be found in Algorithm 1. Through this stressmark, we are implementing the most efficient user space program for controlling the TLB entries. We will use this stressmark to allocate different numbers of pages and to understand the condition in which the TLB is most vulnerable.

Algorithm 1 Stressmark Pseudocode

```
1:  procedure Stressmark(Pages, WarmupRepeats, ROIRepeats)
2:      PageSize ← GetPageSize();
3:      memory ← mmap(PageSize * Pages);
4:      for i ← 0, WarmupRepeats do                          ▷ Used to warmup the TLBs
5:          memory[(i mod PageSize) * PageSize] = RandomCharacter;
6:      end for
7:      StartROI();
8:      for i ← 0, Repeats do
9:          memory[(i mod PageSize) * PageSize] = RandomCharacter;
10:     end for
11:     EndROI();
12:     return
13: end procedure
```

3.5 Automation Scripts

We also implemented several automation scripts to automate the simulation process. The gem5 full-system simulation uses a telnet terminal to connect to the simulated terminal. To make an accurate simulation, we need to gather several gem5 checkpoints from the beginning of our region of interest (ROI) in our workloads and separately run the instructions from the checkpoint to the end of our ROI. So, we created automation scripts to gather the necessary checkpoints (both for the stressmark and the workloads) and run the experiments. We then gathered the results and presented them.

3.6 Experimental Methodology

The x86 ISA in gem5 has a small degree of configurability. We created a single level of TLBs with seperate ITLB and DTLB. We chose a size of 64 entries (standard for modern systems). In the ARM ISA, the gem5 simulator offered larger configurability. It has a hierarchy of TLBs, both shared and dedicated (data and instruction). We chose a two-level hierarchy with 64 entry L1 TLBs for data and instructions and a shared L2 TLB with 1280 entries. It is worth mentioning that the ARM ISA offers a different kind of TLB called nested TLBs (in the ARM ISA context, they are called stage 2 TLBs) used for virtualized environment translations. Our methodology is functional in these TLBs, but we do not use any virtualized workloads, so we do not present AVF calculations for these components. The simulation configuration for each ISA can be found in Table 3.1 and Table 3.2.

Table 3.1: x86 Simulation Configuration

x86 Simulation Configuration	
TLB Hierarchy	
ITLB	64 entry, fully-assocative, LRU replacement policy
DTLB	64 entry, fully-associative, LRU replacement policy
Cache Hierarchy	
L1 Data	32kB, 4-way associative, LRU Replacement Policy
L1 Instruction	32kB, 2-way associative, LRU Replacement Policy
L2 Shared	1024kB, 16-way associative, LRU Replacement Policy
Core Characteristics	
Type	BaseO3CPU (Out Of Order CPU)
Branch Predictor	Tournament Branch Predictor
Memory	
Size	512MB
Linux Kernel Info	
Version	Version 5.4.49
Gem5 Characteristics	
Version	Version 22.1.0.0
Simulation Type	Full-System Simulation

Table 3.2: ARM Simulation Configuration

TLB Hierarchy	
L1 ITLB	64 entry, fully-assocative, LRU replacement policy
L1 DTLB	64 entry, fully-associative, LRU replacement policy
L2 Shared TLB	1280 entry, fully-associative, LRU replacement policy
Stage 2 ITLB	64 entry, fully-associative, LRU replacement policy
Stage 2 DTLB	64 entry, fully-associative, LRU replacement policy
Cache Hierarchy	
L1 Data	32kB, 4-way associative, LRU Replacement Policy
L1 Instruction	32kB, 2-way associative, LRU Replacement Policy
L2 Shared	1024kB, 16-way associative, LRU Replacement Policy
Core Characteristics	
Type	BaseO3CPU (Out Of Order CPU)
Branch Predictor	Tournament Branch Predictor
Memory	
Size	512MB
Linux Kernel Info	
Version	Version 3.16.0-rc6
Gem5 Characteristics	
Version	Version 22.1.0.0
Simulation Type	Full-System Simulation

We used workloads from the MiBench [5] suite. To better understand the nature of our workloads, we measured their memory footprint using Valgrind [1, 30] and the Massif toolset. In Tables 3.3 and 3.4, we can see information about the workloads we will use for the x86 and ARM experiments. All the stressmarks executed 10000 repeats and accessed a unique number of 4KB pages each. Specifically, for the x86 ISA, we used 1, 32, 64, 128, 63 number of pages. For the ARM ISA, we used 1, 4, 16, 32, 48, 60, 64, 128, 256, 512,

Table 3.3: Workloads x86 ISA

Name	Peak Memory Usage
basicmath_small	1.1MiB
basicmath_large	1.1MiB
bitcount_small	988 KiB
bitcount_large	988 KiB
qsort_small	1.1 MiB
qsort_large	2.2 MiB
susan_small	1 MiB
susan_large	1.3 MiB
jpeg_small	4.8 MiB
jpeg_large	5.5 MiB
dijkstra_small	1.1 MiB
dijkstra_large	1.1 MiB
patricia_small	2.2 MiB
patricia_large	7.8 MiB
sha_small	4.6 MiB
sha_large	4.6 MiB

Table 3.4: Workloads ARM ISA

Workload Name	Peak Memory Usage
susan_c_small	1.0 MiB
susan_c_large	1.9 MiB
basicmath_large	1.1MiB
qsort_large	2.2 MiB
susan_e_large	1.7 MiB
susan_s_small	1 MiB
bitcount_large	988 KiB
basicmath_small	1.1MiB
susan_s_large	1.3 MiB
qsort_small	1.1 MiB
susan_e_small	1.0 MiB
bitcount_small	988 KiB

and 1024 number of pages. We conducted more stressmark experiments on the ARM ISA to test our methodology for our multilevel TLB hierarchy.

4. EXPERIMENTAL RESULTS

After explaining how the simulation and methodology work, we present and examine our results. First, we evaluate our methodology and find the best and worst-case vulnerability scenarios using our stressmark. Then, we analyse the workload-based results and evaluate them.

4.1 Stressmarks Results

First, we are going to take a look at the Stressmark results. For each ISA, we will (1) Evaluate the correctness of our methodology and (2) examine the extreme case scenarios for the AVF of each TLB. We created the stressmark to test the data array portion of our methodology, so we will not include a total AVF calculation.

4.1.1 x86 Stressmark Results

In our simulation, the current implementation of gem5 for the x86 ISA supports only a single-level TLB. As is the state-of-practice, we use separate data and instruction TLBs. We separate each TLB into its Data Array and its Tag Array. Then, we calculate their AVF. Finally, we combine them to calculate the total AVF of each component.

First, we need to understand what we expect from the simulation. In Section 3.2.1, we discuss that the deletion and insertion of new entries in the TLB, generate un-ACE cycles. If we have several pages that are constantly reused, there will not be as many un-ACE cycles. As a result, we will have a larger AVF. So, let's look at the case that the pages used do not fill the entire TLB (remember the TLB size is 64 entries). In this case, we expect a linear increase in the AVF as the number of pages increases until we reach the 63rd page (the 64th page is allocated regardless of data outside the heap). If we look at the data from Table 4.1, the results seem to be the same as our prediction. In the case of the ITLB (Instruction TLB), we only use a single page of instructions. Thus, the AVF is around 7-9%. In the case of the DTLB (the component that the stressmark tests), we see a linear increase in the AVF from 26.14% when 1 page is used (2 with the page used for data outside the heap) up to 99.49% when we use the entirety of the TLB.
Next, let's see what happens when we use more than 63 pages. As we employ a sequential access pattern, we constantly evict the LRU page. Thus, most of the execution cycles are labeled as un-ACE. So, we observe that the AVF drops significantly. It is important to note that the length of the simulation plays a part in this result because the potential ACE cycles of the bonus page (caused by its reuse) become irrelevant compared to the total number of cycles. In Table 4.1, we also include the total number of simulation cycles to illustrate this point.

Table 4.1: Stressmark Results for x86 ISA

Pages Used	AVF_DTLB	AVF_ITLB	Simulation Cycles
1	26.14%	8.31%	817,857,000
32	84.45%	7.72%	38,639,666,500
63	99.49%	9.17%	139,252,445,000
64	1.71%	7.86%	7,266,505,000
128	1.70%	7.79%	154,523,141,000

4.1.2 ARM Stressmark Results

For the ARM ISA, we are using a multilevel TLB hierarchy. For this reason, we are conducting more experiments. Our expected data array AVF follows the same rules as the x86 case. In Table 4.2, we observe the same linearity in the L1 DTLB AVF from 3.49% up to 95.19% for the 60-page experiment. For more than 60 pages, we observe an average AVF of 1.8%. The L1 ITLB AVF ranges from 1.61% to 6.23%. That happens because we use a small number of pages.

The intriguing feature of the ARM ISA simulation is the shared L2 TLB. As mentioned in Section 3.3, the L2 shared TLB has 1280 entries. So, we expect the AVF to increase linearly as the number of pages increases. Subsequently, the AVF should become relatively benign. In Table 4.2, we validate these assumptions as the AVF linearly increases from 0.005% up to 80.28% for the 1024-page experiment. Finally, in the 2048-page experiment, the AVF drops to 0.07%. To better portray the AVF linearity, we created Figure 4.1.

Table 4.2: Stressmark Results for ARM ISA

Pages Used	L1 DTLB	L1 ITLB	L2 SHARED TLB
1	3.4936%	1.7806%	0.0050%
4	7.9346%	1.6619%	0.0019%
16	26.5521%	1.6290%	0.0011%
32	51.4068%	1.6303%	0.0011%
48	76.3206%	1.6102%	0.0008%
60	95.1920%	2.7167%	1.3041%
64	1.8032%	1.6279%	4.9647%
128	1.7828%	3.4996%	10.2860%
256	1.7622%	4.8583%	20.5193%
512	1.7521%	5.6636%	40.5342%
1024	1.7490%	6.1218%	80.2886%
2048	1.7412%	6.2308%	0.0786%

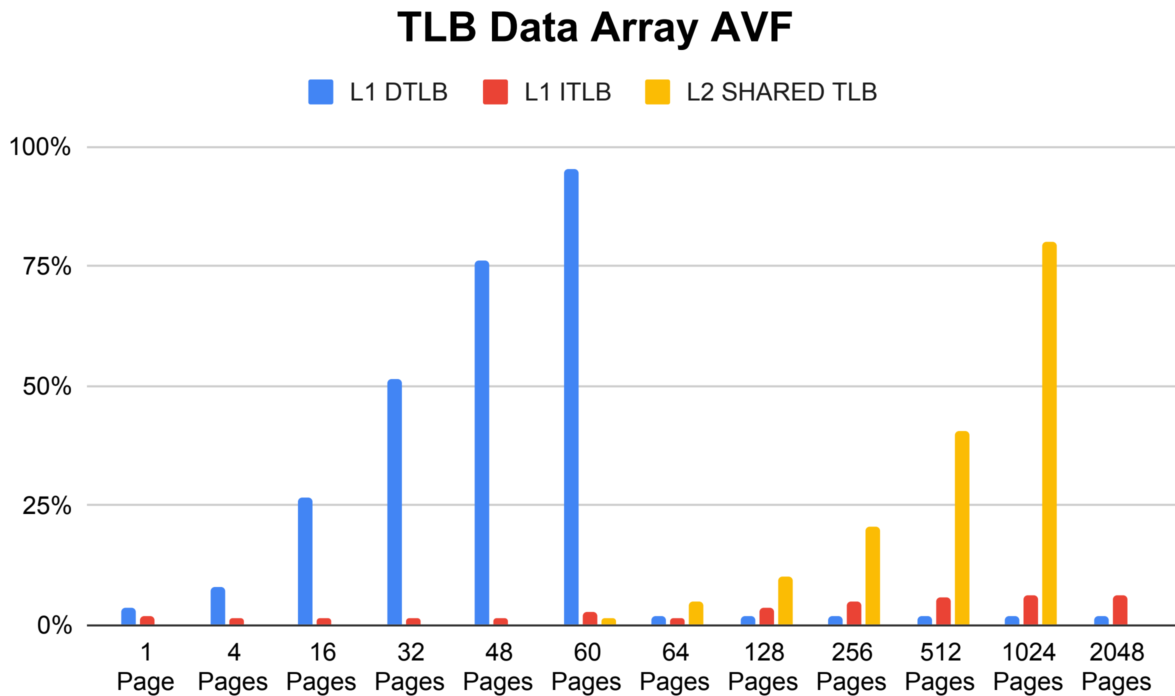

Figure 4.1: Stressmark Results for ARM ISA

4.2 x86 MiBench Results

4.2.1 AVF calculation

Next, we need to analyze the workload results. First, we will take a look at the x86 TLB hierarchy. We will examine the data array results, then the tag array results and finally, we will calculate the total AVF for each TLB and the entire TLB hierarchy.

Let us first analyze the vulnerability of the data array. In Figure 4.2, we see the AVF of the DTLB and ITLB for all our workloads. For the DTLB, we observe larger AVF values than the ITLB. There are many reasons why this could be the case. As shown in 3.3, our workloads contain enough data to fill the DTLB. Subsequently, as the AVF is 70.36% on average, we assume there is enough TLB entry reuse so that a significant percentage of the cycles are labeled as ACE. On the contrary, in the ITLB, we observe 45.76% AVF on average. That can be caused either (1) by a partially empty TLB (around half) that the bits of the invalidated entries are considered un-ACE throughout the execution or (2) by a lower page reuse compared to the DTLB.

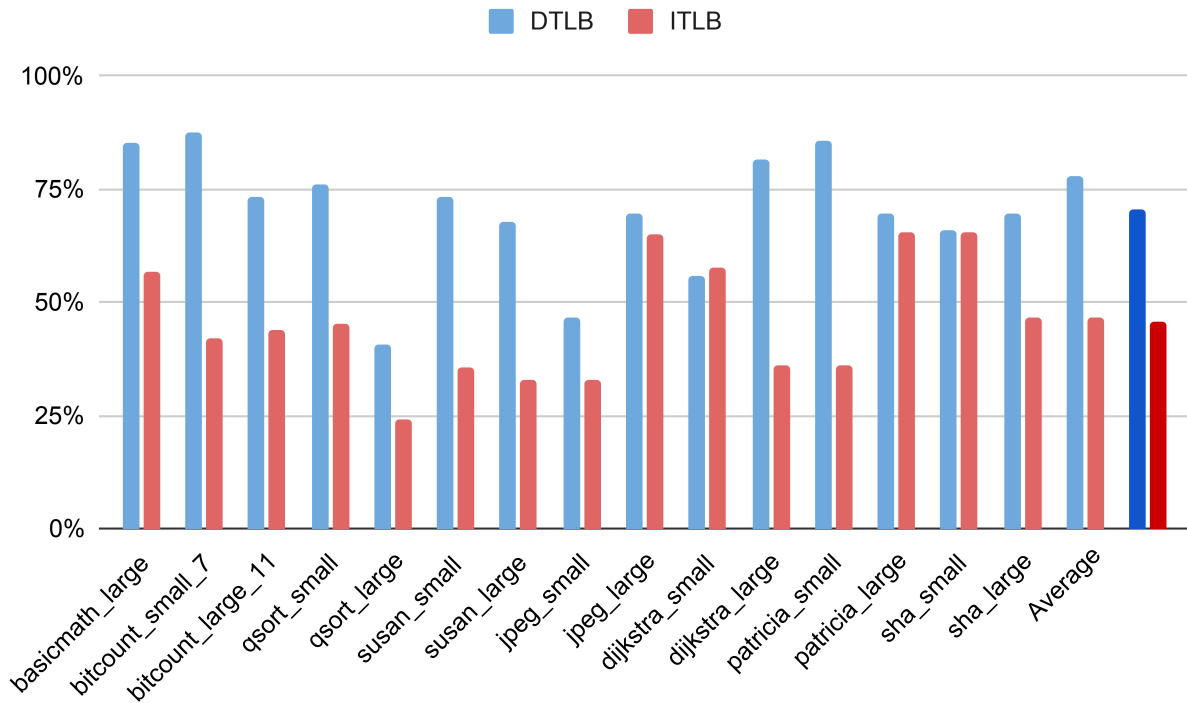

Figure 4.2: Workloads AVF for Data Array (x86 ISA)

Next, we will examine the vulnerability of the tag array of the two TLBs. First, in Figure 4.3 we notice that the AVF of the tag array is tiny compared to the one on the data array for both components. That makes sense because the probability of a tag array bit being ACE is much lower compared to a data array bit because (1) there is a low probability that a hamming distance one match exists during a TLB access and (2) even if that is the case only one bit in the entire tag is considered ACE (as opposed to the per entry nature of the data array). All in all, the DTLB has an average tag array AVF of 0.04% and the ITLB has an average tag array AVF of 0.09% (the linearity of the instructions' accesses causes slightly more hamming distance one matches).

The next step is to calculate the total vulnerability of each of the two TLBs. To calculate the total AVF, we multiply each array's AVF by the percentage of its bits out of the total bits of the component. We can achieve that using this formula:

$$AVF_{total} = AVF_{DataArray} * \frac{DataArrayBits}{TotalBits} + AVF_{TagArray} * \frac{TagArrayBits}{TotalBits}$$

$$\begin{aligned} AVF_{total} = AVF_{DataArray} * \frac{DataArrayBits}{DataArrayBits + TagArrayBits} \\ + AVF_{TagArray} * \frac{TagArrayBits}{DataArrayBits + TagArrayBits} \end{aligned}$$

We then simplify the formula, trying to eliminate the number of entries of the TLB (this will be useful for the ARM ISA L2 TLB later).

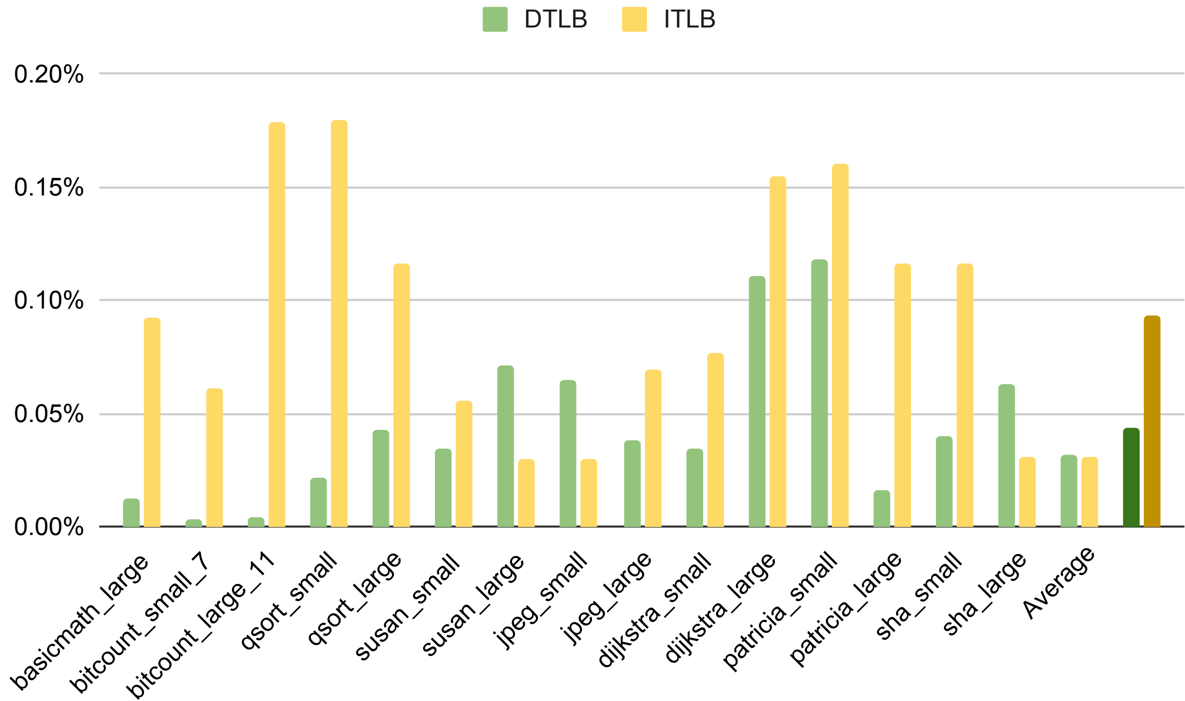

Figure 4.3: Workloads AVF for Tag Array (x86 ISA)

$$
\begin{aligned}
AVF_{total} = AVF_{DataArray} * \frac{DataSize * TLB_Entries}{DataSize * TLB_Entries + TagSize * TLB_Entries} \\
+AVF_{TagArray} * \frac{TagSize * TLB_Entries}{DataSize * TLB_Entries + TagSize * TLB_Entries}
\end{aligned}
$$

$$
\begin{aligned}
AVF_{total} = AVF_{DataArray} * \frac{DataSize * TLB_Entries}{(DataSize + TagSize) * TLB_Entries} \\
+AVF_{TagArray} * \frac{TagSize * TLB_Entries}{(DataSize + TagSize) * TLB_Entries}
\end{aligned}
$$

Finally, with some simplifications:

$$
AVF_{total} = AVF_{DataArray} * \frac{DataSize}{DataSize + TagSize} + AVF_{TagArray} * \frac{TagSize}{DataSize + TagSize}
$$

Specifically, in the x86 ISA, we have a 52-bit tag and a 64-bit data entry:

$$
AVF_{total} = AVF_{DataArray} * \frac{64}{52 + 64} + AVF_{TagArray} * \frac{52}{52 + 64}
$$

$$
AVF_{total} = AVF_{DataArray} * \frac{64}{116} + AVF_{TagArray} * \frac{52}{116}
$$

Next, we apply this formula for the DTLB and ITLB. Also, we calculate the total AVF for all workloads and the average. We plot it in Figures 4.4 and 4.5. The low tag array AVF brings the total AVF lower for both components. For the DTLB, the total average AVF is 36.94%. For the ITLB, the total average AVF is 24.05%.

Finally, we will calculate the average AVF for the entire TLB hierarchy. To make this calculation, we need to take into consideration the AVF of each TLB proportionally to each size. Because both TLBs have the same size each AVF will be multiplied with a factor of 0.5 as so:

$$AVF_{hierarchy} = 36.94 * \frac{DTLB_{size}}{Total_{size}} + 24.05 * \frac{ITLB_{size}}{Total_{size}} = 36.94 * 0.5 + 24.05 * 0.5 = 30.495\%$$

Thus, the AVF of the entire TLB hierarchy is 30.495%.

4.2.2 FIT rate calculation

Next, we will calculate the FIT rate for the entire TLB hierarchy. We will be utilizing the formulas and the intrinsic FIT rate an SRAM array bit ($5 * 10^{-6}$) mentioned in Section 2.2 and Section 2.3 respectively. Due to the additive nature of the FIT rate (and thus the intrinsic FIT rate), we can calculate the intrinsic FIT rate of the entire TLB hierarchy.

$$FITintrinsic_{system} = \sum_{i=0}^{n} FITintrinsic_i$$

The number of bits of the entire TLB hierarchy are the bits of the ITLB and the DTLB combined. Both TLBs have 116 bits per entry (64 bits of data and 52 bits of tag) and 64 entries. Thus the entire TLB hierarchy has $116 * 64 * 2 = 14848bits$

Next, we calculate the FIT rate considering the components are constantly fault-prone:

$$FITrate_{hierarchy} = FITintrinsic_{bit} * TotalNumberofBits * AVF_{hierarchy}$$

$$FITrate_{hierarchy} = 5 * 10^{-6} * 14848 * 0.30495 = 0.0226$$

Finally, the FIT rate for the entire TLB hierarchy is 0.0226. Consequently, 0.0226 errors will occur in a period of (10^9) hours on average.

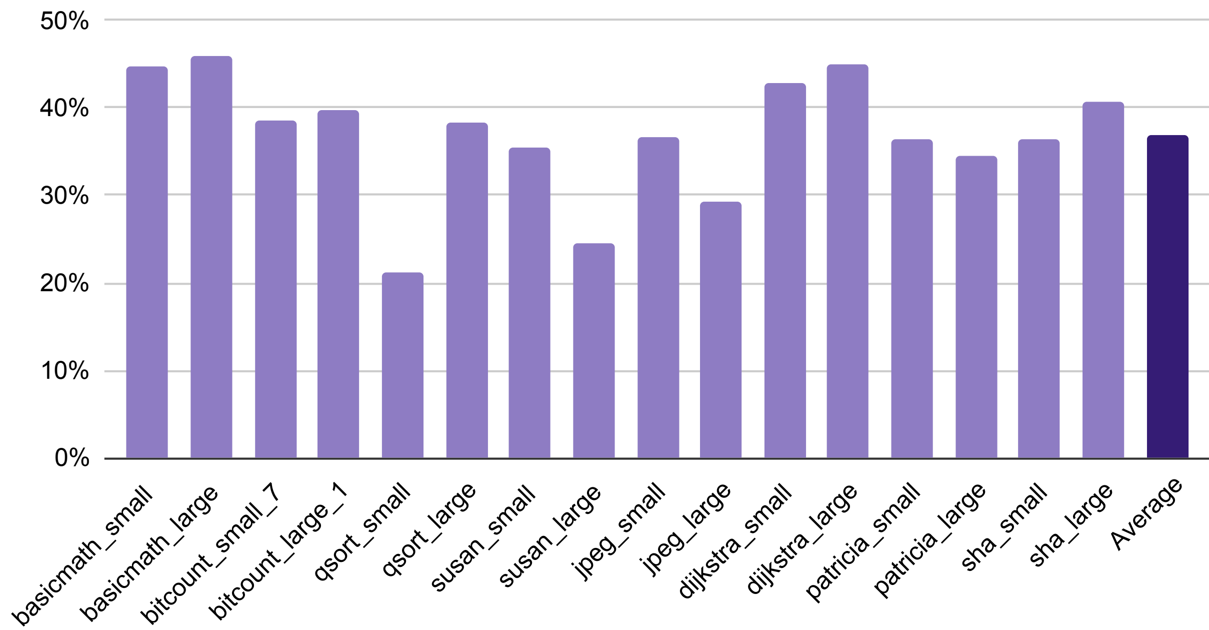

Figure 4.4: DTLB AVF for x86 ISA

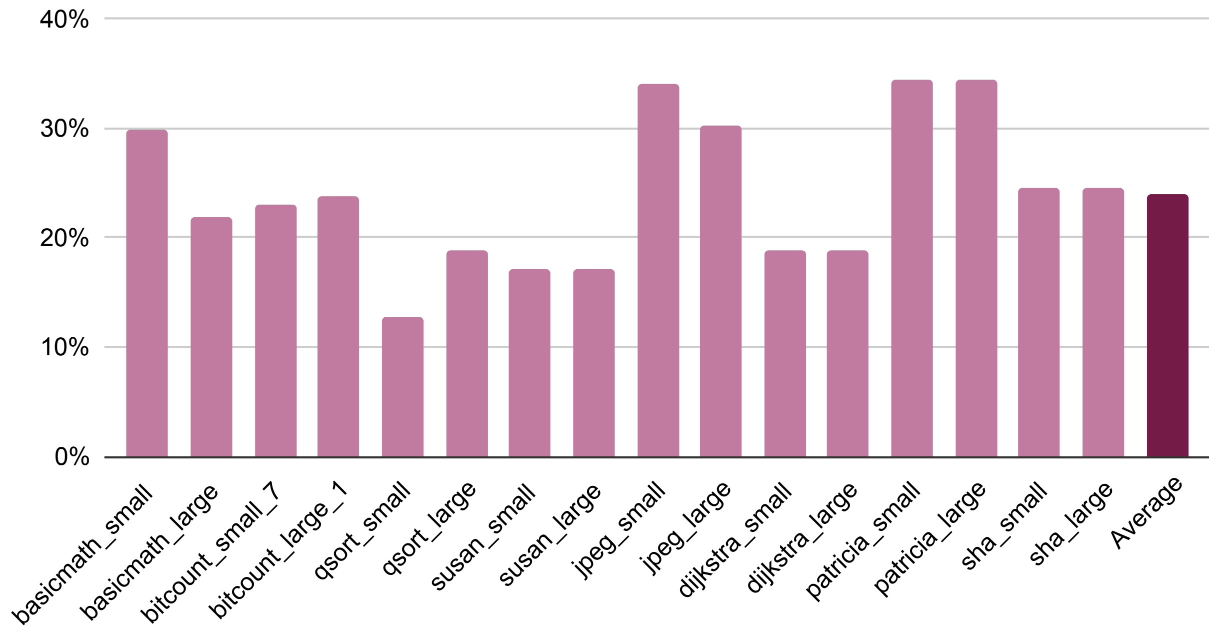

Figure 4.5: ITLB AVF for x86 ISA

4.3 ARM MiBench Results

4.3.1 AVF calculation

The ARM ISA has a more complicated TLB hierarchy. This time, we need to analyze the results for the L1 DTLB, L1 ITLB and the L2 Shared TLB. First, we will examine the data arrays, the tag arrays and the total AVF of each TLB and the entire TLB hierarchy.

Let us first analyze the vulnerability of the data array. In Figure 4.6, we can find the results for all three components. First, the DTLB has an average AVF of 48.5%. It is not as large as the DTLB in the x86 ISA but still the largest compared to the other two TLBs. The ITLB has an average AVF of 38.21%. That is closer to the x86 case because, despite being different architectures, we are using the same workloads and the same number of instructions. Finally, the L2 shared TLB has an average AVF of 6.3%. The AVF is smaller because the L2 TLB is never through the execution filled with entries (it contains 1280 entries). We can observe that in the workloads that utilize more memory (qsort_large, susan_c_large), the AVF is larger. That follows the pattern mentioned in Section 4.2.

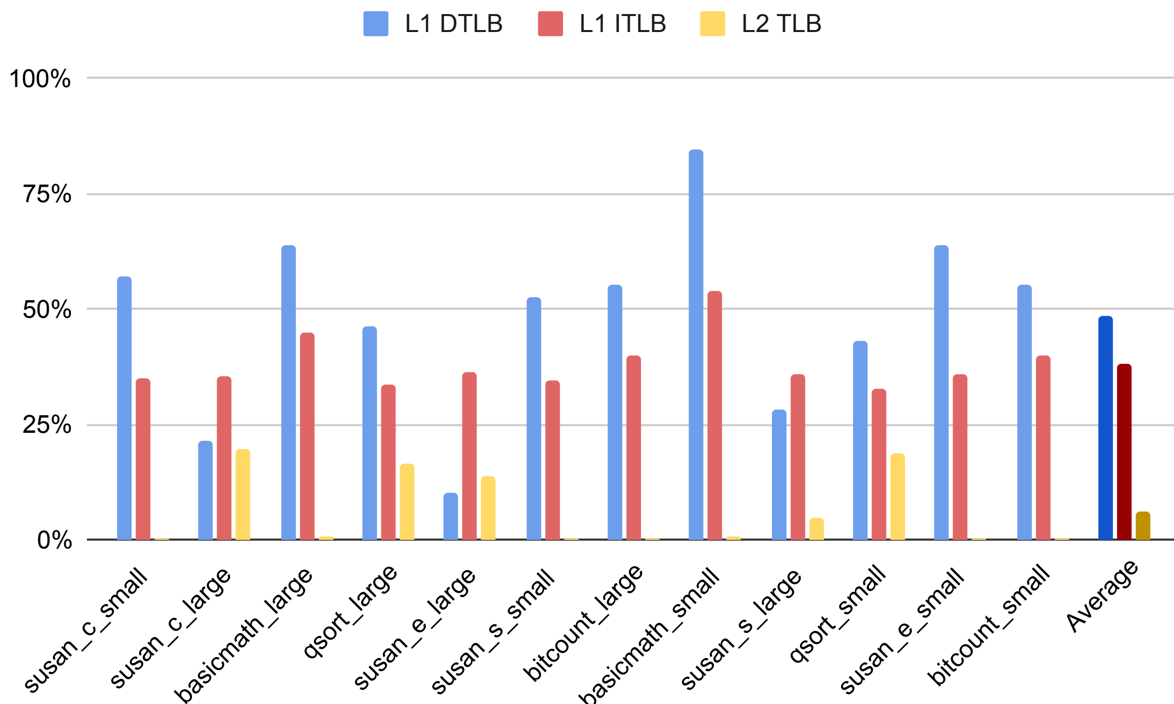

Figure 4.6: Data Array AVF ARM ISA

Next, we need to examine the tag array AVF results. In Figure 4.7 we can see that for all three components, the tag array AVF is tiny compared to the data array AVF. The reason is the scarcity of the ACE bits (the scarcity of hamming distance one matches) in the tag array through the execution. Only the ITLB has a slightly larger AVF due to the linearity of its pages. The average tag array AVF of the DTLB, ITLB and L2 shared TLB is 0.04%, 0.07% and 0.002% respectively.

Next, we need to calculate the total AVF of each of the three components. We can use the formula we calculated in the previous section:

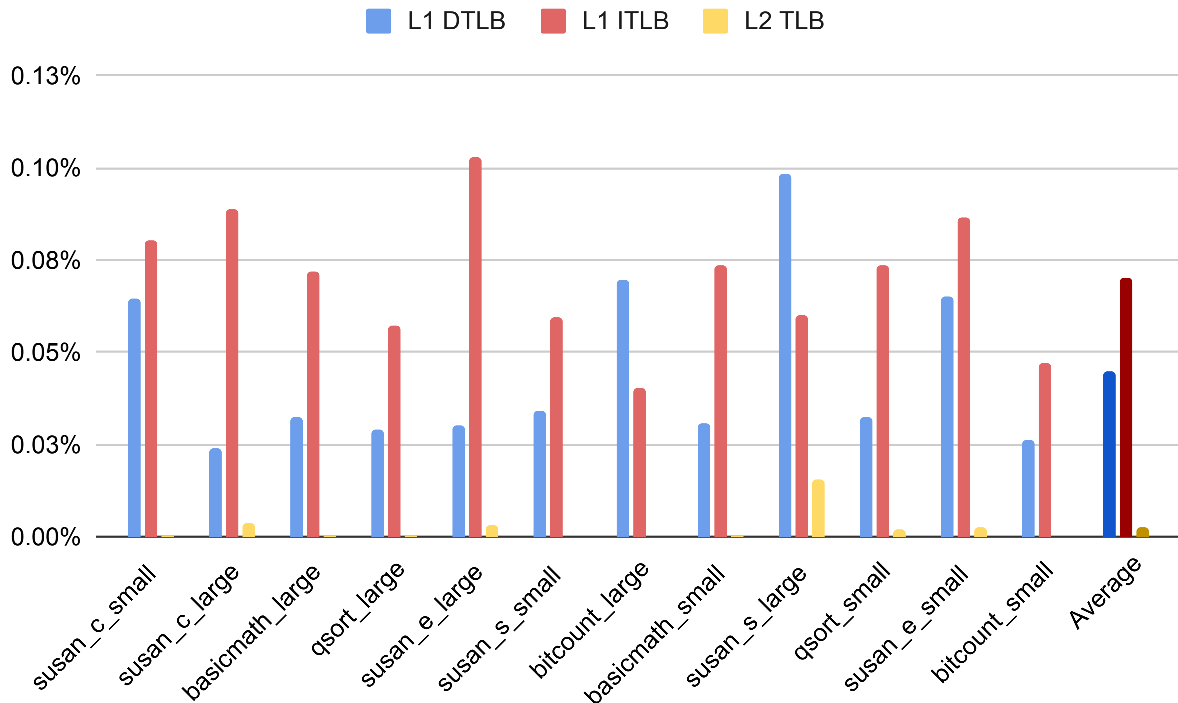

Figure 4.7: Tag Array AVF ARM ISA

$$AVF_{total} = AVF_{DataArray} * \frac{DataSize}{DataSize + TagSize} + AVF_{TagArray} * \frac{TagSize}{DataSize + TagSize}$$

Specifically, in the ARM ISA, we have a 52-bit tag and a 64-bit data entry:

$$AVF_{total} = AVF_{DataArray} * \frac{64}{52 + 64} + AVF_{TagArray} * \frac{52}{52 + 64}$$

$$AVF_{total} = AVF_{DataArray} * \frac{64}{116} + AVF_{TagArray} * \frac{52}{116}$$

We used the final form of this formula to calculate the total AVF for each component. The results, can be found in Figure 4.8. The total AVF for the DTLB, ITLB and L2 shared TLB is 25.46%, 20.08% and 3.31%, respectively. It is worth mentioning that the total AVF of each component is smaller than the AVF of its data array because of the tiny tag array AVF.

Finally, we will calculate the AVF of the entire TLB hierarchy. We can calculate this by proportionally weighting the AVF of each TLB to their sizes as so:

$$AVF_{hierarchy} = AVF_{DTLB} * \frac{DTLBsize}{TotalHierarchySize} + AVF_{ITLB} * \frac{ITLBsize}{TotalHierarchySize} +$$

$$AVF_{L2TLB} * \frac{L2TLBsize}{TotalHierarchySize}$$

Next, we will calculate the total hierarchy bits. For all TLBs, the entry size is 116 (64-bit data 52-bit tag). The total number of entries is $64 + 64 + 1280 = 1408$. Thus, the total hierarchy size is $116*1408 = 163328$ (L1 TLB: $64*116 = 7424$, L2 TLB: $1280*116 = 148480$). Finally, the total AVF of the hierarchy is:

$$AVF_{hierarchy} = 0.2546 * \frac{7424}{163328} + 0.2008 * \frac{7424}{163328} + 0.0331 * \frac{148480}{163328} = 5.07\%$$

Consequently, the AVF of the entire ARM TLB hierarchy is 5.07%.

4.3.2 FIT rate calculation

Next, we will calculate the FIT rate for the entire TLB hierarchy. We will be utilizing the formulas and the intrinsic FIT rate an SRAM array bit ($5 * 10^{-6}$) mentioned in Section 2.2 and Section 2.3 respectively. Due to the additive nature of the FIT rate (and thus the intrinsic FIT rate), we can calculate the intrinsic FIT rate of the entire TLB hierarchy.

$$FITintrinsic_{system} = \sum_{i=0}^{n} FITintrinsic_i$$

The number of bits of the entire TLB hierarchy is 163,328. Next, we calculate the FIT rate considering the components are constantly fault-prone:

$$FITrate_{hierarchy} = FITinstrinsic_{bit} * TotalNumberofBits * AVF_{hierarchy}$$

$$FITrate_{hierarchy} = 5 * 10^{-6} * 163328 * 0.0507 = 0.0414$$

Finally, the FIT rate for the entire TLB hierarchy is 0.0414. Consequently, 0.0414 errors will occur in a period of (10^9) hours on average.

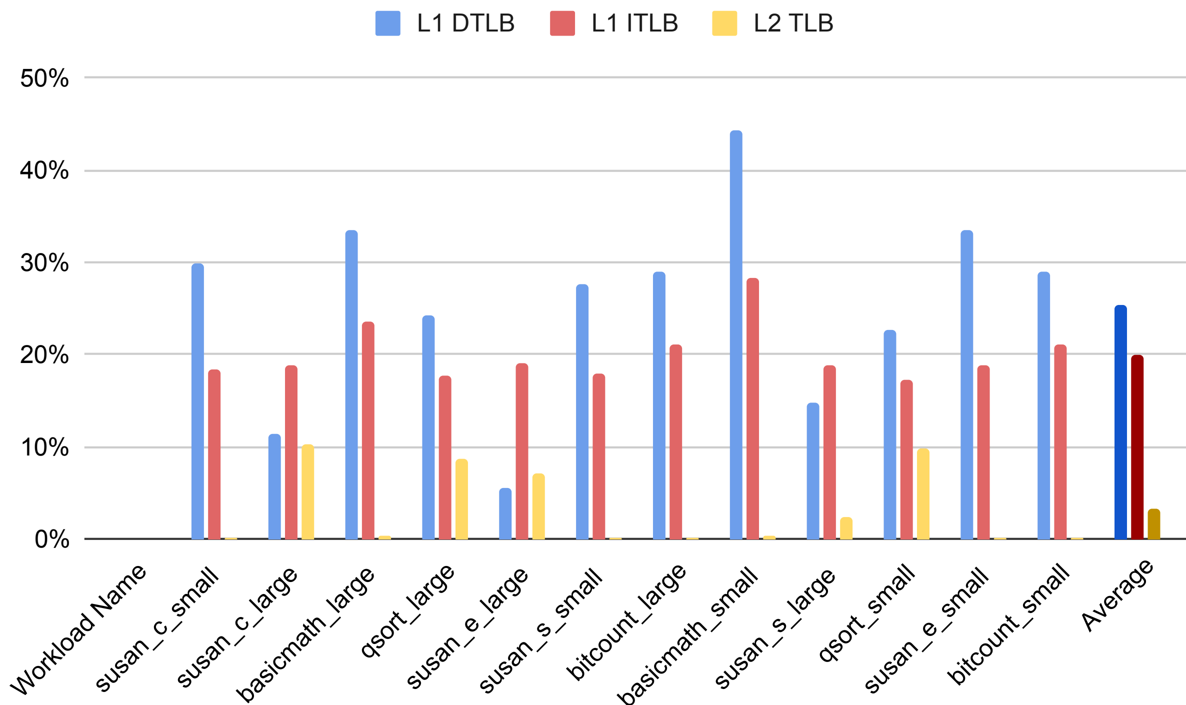

Figure 4.8: Total Array AVF ARM ISA

4.4 Comparing the ARM and x86 ISAs

Having two different ISAs at our disposal allows us to compare them. Unfortunately, due to differences in the organization of the TLB hierarchy of the simulated system for each ISA, we will essentially be comparing the two setups and microarchitectures rather than the ISAs. Regardless, we can extract some insights.

4.4.1 AVF comparison

We will start by comparing the ITLBs and DTLBs. In both cases, the AVFs in the x86 ISA are higher compared to the ARM ISA. This difference is attributed to different hierarchy parameters, such as the associativity and the existence of a second-level TLB. Next, we can examine the AVF of the entire TLB hierarchy. Again, the AVF in the case of the ARM ISA is lower compared to the x86 results. This is heavily attributed to the existence of a huge L2 TLB that has a small AVF (due to the limited size of the workloads), which significantly lowers the total AVF.

4.4.2 FIT rate comparison

Finally, we will examine the total FIT rates of each ISA's TLB hierarchies. This time, the x86 ISA has the lower FIT rate. For this calculation, we consider the number of bits of each TLB hierarchy. In the ARM ISA case, the L2 TLB is huge compared to the entire x86 hierarchy's size. Intuitively, having a large component by area increases the total number of bit flips. Consequently, despite having a small AVF value, the errors occurring within a specific period will be substantial.

5. CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this study, we calculated the AVF of the TLB hierarchies of different systems using the ACE methodology. We used the gem5 simulator to implement the ACE methodology in the x86 and ARM ISA. The gem5 x86 ISA implementation offered a single-level TLB hierarchy and the ARM ISA implementation a multilevel TLB hierarchy with an L2 shared TLB. We also created a stressmark to test our methodology and its limits. Furthermore, we calculated the AVF using workloads from the MiBench benchmark suite. We observed that for the x86 ISA, the average AVF from our workloads of the DTLB is 36.94% and for the ITLB is 24.05%. For the ARM ISA, the average AVF for the L1 ITLB is 20.08%, for the L1 DTLB is 25.46%, and for the L2 shared TLB is 3.31%.

5.2 Future Work

When we decided which cycles would be considered ACE in the data array, we mentioned that the calculation would happen on a per-entry basis. There are two ways we could measure the bits with smaller granularity. The first way is taking into account where the altered bit leads in the main memory. When a bit does not alter the same process's pages it should be considered un-ACE. Taking the calculation a step further, we can count in a separate cycle pool the ACE bits that alter a memory position belonging to a different process on the system. This way, we can create a metric that shows the probability that a fault will become a user-visible error in a different process of the system.

A different way to approach the smaller pool cycle granularity would be to differentiate between the DUE and SDC AVF. That could be possible by tracking the data array entry's path into the system (in the case of a TLB hit) for a small number of instructions. If it is used in a way that could potentially end the execution (e.g., as a pointer to an error page), the bit could be classified as DUE AVF instead of simply AVF. The rest of the bits could be considered ’potentially’ SDC. Finally, due to the critical nature of the TLBs as the systems protection mechanism, there should be more research on their vulnerability and its effects on the execution of the entire system.

ABBREVIATIONS - ACRONYMS

ACE	Architecturally Correct Execution
ARM	Advansed RISC Machine
AVF	Architectural Vulnerability Factor
CPU	Central Processing unit
DTLB	Data Translation Lookaside Buffer
DUE	Detected Unrecoverable Error
ECC	Error Corection Code
FIT	Failure In Time
ISA	Instruction Set Architecture
ITLB	Instruction Translation Lookaside Buffer
L1	Level 1
L2	Level 2
LRU	Least Recently Used
MTTF	Mean Time To Failure
OS	Operating System
ROI	Region Of Interest
SDC	Silent Data Corruption
SECDED	Single-bit Error Correction, Double-bit Error Detection
SEC	Single-bit Error Correction
SFI	Statistical Fault Injection
SIMD	Single Instruction Multiple Data
SRAM	Static Random-Access memory
TLB	Translation Lookaside Buffer
TVF	Timing Vulnerability Factor
VPN	Virtual Page Number
PFN	Physical Frame Number
PWC	Page Walk Cache

BIBLIOGRAPHY

[1] Valgrind Toolset. https://valgrind.org/.

[2] Jacob A. Abraham. Cross-layer resilience: are high-level techniques always better? In *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 78–78, 2016.

[3] AMD. SEV-SNP (Secure Nested Paging). https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/solution-briefs/amd-secure-encrypted-virtualization-solution-brief.pdf.

[4] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 48–59, New York, NY, USA, 2010. Association for Computing Machinery.

[5] Jeremy Bennett. MiBench Suite. https://github.com/embecosm/mibench?tab=readme-ov-file.

[6] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. *SIGARCH Comput. Archit. News*, 36(1):26–35, mar 2008.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[8] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S.S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 532–543, 2005.

[9] Pablo Bodmann, George Papadimitriou, Dimitris Gizopoulos, and Paolo Rech. The impact of soc integration and os deployment on the reliability of arm processors. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 223–225, 2021.

[10] Pablo R. Bodmann, George Papadimitriou, Rubens L. Rech Junior, Dimitris Gizopoulos, and Paolo Rech. Soft error effects on arm microprocessors: Early estimations versus chip measurements. *IEEE Transactions on Computers*, 71(10):2358–2369, 2022.

[11] Spencer W. Ng Bruce Jacob and David T. Wang. *Memory Systems*, chapter 2, pages 79–115. 2008.

[12] Yaman Cakmakci and Oguz Ergin. Exploiting virtual addressing for increasing reliability. *IEEE Computer Architecture Letters*, 13(1):29–32, 2014.

[13] Odysseas Chatzopoulos, George Papadimitriou, Vasileios Karakostas, and Dimitris Gizopoulos. Gem5-marvel: Microarchitecture-level resilience analysis of heterogeneous soc architectures. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 543–559, 2024.

[14] Eric Cheng, Shahrzad Mirkhani, Lukasz G. Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R. Stan, Klas Lilja, Jacob A. Abraham, Pradip Bose, and Subhasish Mitra. Clear: Cross-layer exploration for architecting resilience: Combining hardware and software techniques to tolerate soft errors in processor cores. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[15] Nikos Foutris, Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera, and Antonio Gonzalez. Accelerating microprocessor silicon validation by exposing isa diversity. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 386–397, New York, NY, USA, 2011. Association for Computing Machinery.

[16] Nikos Foutris, Dimitris Gizopoulos, Xavier Vera, and Antonio Gonzalez. Deconfigurable microprocessor architectures for silicon debug acceleration. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 631–642, New York, NY, USA, 2013. Association for Computing Machinery.

[17] Dimitris Gizopoulos, George Papadimitriou, and Odysseas Chatzopoulos. Estimating the failures and silent errors rates of cpus across isas and microarchitectures. In *2023 IEEE International Test Conference (ITC)*, pages 377–382, 2023.

[18] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[19] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, 2013.

[20] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Adam Jacobs, Grzegorz Cieslewski, Alan D. George, Ann Gordon-Ross, and Herman Lam. Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive fpga-based space computing. *ACM Trans. Reconfigurable Technol. Syst.*, 5(4), dec 2012.

[22] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 66–78, 2015.

[23] Linux Kernel. Process Context Identifiers (PCID). https://www.kernel.org/doc/Documentation/x86/pti.txt.

[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[25] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 502–506, 2009.

[26] Ian Vince McLoughlin and Timo Rolf Bretschneider. Reliability through redundant parallelism for micro-satellite computing. *ACM Trans. Embed. Comput. Syst.*, 9(3), mar 2010.

[27] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[28] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 29–40, 2003.

[29] S.S. Mukherjee, C.T. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. Measuring architectural vulnerability factors. *IEEE Micro*, 23(6):70–75, 2003.

[30] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007.

[31] Ashish Panwar, Sorav Bansal, and K. Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 347–360, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. *SIGPLAN Not.*, 53(2):679–692, mar 2018.

[33] George Papadimitriou, Athanasios Chatzidimitriou, Dimitris Gizopoulos, and Ronny Morad. Isa-independent post-silicon validation for the address translation mechanisms of modern microprocessors. In *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 72–77, 2016.

[34] George Papadimitriou, Athanasios Chatzidimitriou, Dimitris Gizopoulos, and Ronny Morad. An agile post-silicon validation methodology for the address translation mechanisms of modern microprocessors. *IEEE Transactions on Device and Materials Reliability*, 17(1):3–11, 2017.

[35] George Papadimitriou and Dimitris Gizopoulos. Demystifying the system vulnerability stack: transient fault effects across the layers. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 902–915. IEEE Press, 2021.

[36] George Papadimitriou and Dimitris Gizopoulos. Anatomy of on-chip memory hardware fault effects across the layers. *IEEE Transactions on Emerging Topics in Computing*, 11(2):420–431, 2023.

[37] George Papadimitriou and Dimitris Gizopoulos. Avgi: Microarchitecture-driven, fast and accurate vulnerability assessment. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 935–948, 2023.

[38] George Papadimitriou and Dimitris Gizopoulos. Silent data corruptions: Microarchitectural perspectives. *IEEE Transactions on Computers*, 72(11):3072–3085, 2023.

[39] George Papadimitriou, Dimitris Gizopoulos, Athanasios Chatzidimitriou, Tom Kolan, Anatoly Koyfman, Ronny Morad, and Vitali Sokhin. Unveiling difficult bugs in address translation caching arrays for effective post-silicon validation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 544–551, 2016.

[40] George Papadimitriou, Dimitris Gizopoulos, Harish Dattatraya Dixit, and Sriram Sankar. Silent data corruptions: The stealthy saboteurs of digital integrity. In *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–7, 2023.

[41] Noah Perryman, Nicholas Franconi, Gary Crum, Christopher Wilson, and Alan D. George. Spacecube ghost: A resilient processor for low-power, high-reliability space computing. In *2024 IEEE Aerospace Conference*, pages 1–11, 2024.

[42] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. Specifying and dynamically verifying address translation-aware memory consistency. *SIGPLAN Not.*, 45(3):323–334, mar 2010.

[43] Adit Singh, Sreejit Chakravarty, George Papadimitriou, and Dimitris Gizopoulos. Silent data errors: Sources, detection, and modeling. In *2023 IEEE 41st VLSI Test Symposium (VTS)*, pages 1–12, 2023.

[44] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1093–1108, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Jovan Stojkovic, Namrata Mantri, Dimitrios Skarlatos, Tianyin Xu, and Josep Torrellas. Memory-efficient hashed page tables. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1221–1235, 2023.

[46] Alfonso Sánchez-Macián, Luis Alberto Aranda, Pedro Reviriego, Vahdaneh Kiani, and Juan Antonio Maestro. Enhancing instruction tlb resilience to soft errors. *IEEE Transactions on Computers*, 68(2):214–224, 2019.

[47] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. Exploiting page table locality for agile tlb prefetching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 85–98, 2021.

[48] Christopher Wilson, Alan George, and Ben Klamm. A methodology for estimating reliability of smallsat computers in radiation environments. In *2016 IEEE Aerospace Conference*, pages 1–12, 2016.

[49] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 28–41, Los Alamitos, CA, USA, oct 2020. IEEE Computer Society.