NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

COMPUTING SYSTEMS: SOFTWARE AND HARDWARE


MASTER THESIS

# Micro-Viruses for Fast and Accurate Characterization of Voltage Margins and Variations in Multicore CPUs


Ioannis S. Vastakis
grad1408

Supervisor: Dimitris Gizopoulos, Professor



ATHENS

JULY 2017

**ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**
**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

**ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ: ΛΟΓΙΣΜΙΚΟ ΚΑΙ ΥΛΙΚΟ**

**ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ**

# ΜΙΚΡΟ-ΙΟΙ ΓΙΑ ΓΡΗΓΟΡΟ ΚΑΙ ΑΚΡΙΒΗ ΧΑΡΑΚΤΗΡΙΣΜΟ ΤΩΝ ΠΕΡΙΘΩΡΙΩΝ ΚΑΙ ΔΙΑΚΥΜΑΝΣΕΩΝ ΤΑΣΗΣ ΣΕ ΠΟΛΥΠΥΡΗΝΟΥΣ ΕΠΕΞΕΡΓΑΣΤΕΣ

**Ιωάννης Σ. Βαστάκης**
**grad1408**

**Επιβλέπων:** **Δημήτρης Γκιζόπουλος,** Καθηγητής

**ΑΘΗΝΑ**

**ΙΟΥΛΙΟΣ 2017**

**MASTER THESIS**

# Micro-Viruses for Fast and Accurate Characterization of Voltage Margins and Variations in Multicore CPUs

**Ioannis S. Vastakis**

**grad1408**

**SUPERVISOR:** **Dimitris Gizopoulos, Professor**

**EXAMINATION COMMITTEE:**

**Dimitris Gizopoulos,** Professor University of Athens

**Antonis Paschalis,** Professor University of Athens

**July 2017**

**ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ**

Μικρό-ιοί για γρήγορο και ακριβή χαρακτηρισμό των περιθωρίων και διακυμάνσεων τάσης σε πολυπύρηνους επεξεργαστές

**Ιωάννης Σ. Βαστάκης**

**grad1408**

**ΕΠΙΒΛΕΠΩΝ:** **Δημήτρης Γκιζόπουλος,** Καθηγητής

**ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΣΤΡΟΠΗ:**

**Δημήτρης Γκιζόπουλος,** Καθηγητής Πανεπιστήμιο Αθηνών

**Αντώνης Πασχάλης**, Καθηγητής Πανεπιστήμιο Αθηνών

**Ιούλιος 2017**

# ABSTRACT

Energy-efficient computing can be largely enabled by fast and accurate identification of the pessimistic voltage margins of multicore CPU designs and in particular the unveiling of voltage margins variability among cores and among chips. In multi-socketed systems with multiple CPUs each of which consists of several cores, the core-to-core and the chip-to-chip voltage margin variability can be effectively utilized by software layers for diligent power-saving threads scheduling.

Massive but straightforward characterization of the voltage margins of different CPU chips and their different cores is an excessively long and thus unaffordable in most cases process if it is naively based on publicly available benchmarks or other in-house programs with large execution times.

In this thesis, we follow a different strategy for the characterization of the voltage margins of multicore CPUs and the measurement of the voltage variability among chips and cores: we propose the employment of fast targeted programs (diagnostic micro-viruses) that aim to stress individually the main hardware components of a multicore CPU architecture which are known to determine the limits of voltage reduction, i.e. the $V_{min}$ values. We describe the development of the micro-viruses which target separately the three different cache memory levels and the main processing components, the integer and the floating-point arithmetic units. The combined execution of the micro-viruses takes very short time compared to regular programs and extensively stress the CPU cores to reveal their voltage limits when they operate below the nominal voltage levels. To demonstrate the effectiveness of the synthetic micro-virus programs, we compare the safe Vmin reported by the combined micro-viruses characterization against the corresponding safe Vmin values of SPEC CPU2006 benchmarks. The micro-viruses based characterization flow requires orders of magnitude shorter time while it delivers very close results to the excessively characterization campaign (in most cases identical, at most 2% divergences) in terms of: (a) Vmin values for the different CPU chips (b) Vmin values for the different cores within a chip, (c) core-to-core and chip-to-chip voltage margins variability.

We evaluate the proposed micro-viruses based characterization flow (and compare it to the SPEC-based flow) on three different chips (a nominal grade and two corner parts) of AppliedMicro's X-Gene 2 micro-server family (8-core, ARMv8-based CPUs manufactured in 28nm); the reported results validate the speed and accuracy of the proposed method.

**SUBJECT AREA**: Dependable and energy efficient computer architectures

**KEYWORDS**: Design Margins Characterization, Energy-Efficient Computing, Multicore CPUs, Undervolting, Diagnostic Viruses, ARMv8

# ΠΕΡΙΛΗΨΗ

Οι ενεργειακά-αποδοτικοί υπολογισμοί είναι δυνατοί μέσω του γρήγορου και ακριβούς προσδιορισμού των δυσοίωνων περιθωρίων τάσης σε σχεδιάσεις πολυπύρηνων επεξεργαστών και πιο συγκεκριμένα με την αποκάλυψη της διακύμανσης των περιθωρίων τάσης ανάμεσα σε πυρήνες και επεξεργαστές. Σε συστήματα με πολλαπλές υποδοχές για πολυπύρηνους επεξεργαστές, με κάθε έναν να διαθέτει πολλαπλούς πυρήνες, η μεταβλητότητα των περιθωρίων μεταξύ πυρήνων και μεταξύ επεξεργαστών μπορεί να αξιοποιηθεί αποτελεσματικά μέσω στρωμάτων λογισμικού για ενεργειακά-αποδοτική χρονοδρομολόγηση των νημάτων.

Ο μαζικός αλλά ειλικρινής χαρακτηρισμός των περιθωρίων τάσης διαφορετικών επεξεργαστών και των πυρήνων τους είναι μια υπερβολικά χρονοβόρα και συνεπώς μη προσιτή διαδικασία στις περισσότερες περιπτώσεις, αν αυτή βασιστεί σε δημόσια διαθέσιμα προγράμματα αναφοράς με μεγάλους χρόνους εκτέλεσης.

Στη παρούσα διπλωματική εργασία, ακολουθούμε μια διαφορετική στρατηγική για τον χαρακτηρισμό των περιθωρίων τάσης πολυπύρηνων επεξεργαστών και για την μέτρηση της διακύμανσης των περιθωρίων αυτών ανάμεσα σε διαφορετικούς επεξεργαστές και πυρήνες: προτείνουμε την υιοθέτησή γρήγορων και στοχευμένων προγραμμάτων (διαγνωστικοί μικρό-ιοί) των οποίων στόχος είναι να πιέσουν ξεχωριστά τα βασικά συστατικά ενός πολυπύρηνου επεξεργαστή τα οποία είναι γνωστό πως καθορίζουν τα όρια στην μείωση της τάσης, με άλλα λόγια την ελάχιστη δυνατή τάση $V_{min.}$ Περιγράφουμε την ανάπτυξη των διαγνωστικών μικρό-ιών που στοχεύουν ξεχωριστά τα τρία επίπεδα των κρυφών μνημών και τις βασικές μονάδες επεξεργασίας, την αριθμητική μονάδα ακέραιων αριθμών και την αριθμητική μονάδα αριθμών κινητής υποδιαστολής. Ο συνδυαστικός χρόνος εκτέλεσης όλων των διαγνωστικών μικρό-ιών είναι σημαντικά συντομότερος από αυτών κανονικών προγραμμάτων, με συνέπεια οι πυρήνες ενός επεξεργαστή να πιέζονται εκτεταμένα ώστε να αποκαλύψουν τα όρια τάσης όταν λειτουργούν κάτω από την ονομαστική τους τάση. Για να επιδείξουμε την αποτελεσματικότητα των μικρό-ιών μας, συγκρίνουμε την ελάχιστη δυνατή τάση λειτουργίας $V_{min}$ που αυτοί προσδιορίζουν με αυτή που προσδιορίζουν υπερβολικά χρονοβόρες καμπάνιες προσδιορισμού με τη βοήθεια των προγραμμάτων αναφοράς SPEC CPU2006. Οι μικρό-ιοί που αναπτύχθηκαν απαιτούν τάξεις μεγέθους μικρότερο χρόνο εκτέλεσης, ενώ την ίδια στιγμή τα αποτελέσματα τους απέχουν ελάχιστα (στις περισσότερες περιπτώσεις είναι πανομοιότυπα, με αποκλίσεις της τάξης του 2%) στον προσδιορισμό (α) της ελάχιστης τάσης λειτουργίας $V_{min}$ για διαφορετικούς επεξεργαστές (β) της ελάχιστης τάσης λειτουργίας $V_{min}$ μεταξύ των διαφορετικών πυρήνων του ίδιου επεξεργαστή και (γ) της μεταβλητότητας των περιθωρίων τάσης από επεξεργαστή σε επεξεργαστή και από πυρήνα σε πυρήνα.

Τέλος, αξιολογούμε πειραματικά την ροή χαρακτηρισμού με τους προτεινόμενους μικρό-ιούς (συγκρίνοντας την με αυτή που βασίζεται στα προγράμματα αναφοράς) σε τρεις διαφορετικούς επεξεργαστές (έναν με nominal grade και δύο που ανήκουν στα corner parts) σε έναν server της οικογένειας X-Gene 2 της εταιρείας AppliedMicro (8 πυρήνες, ARMv8 επεξεργαστής κατασκευασμένος με την διαδικασία 28nm). Τα αποτελέσματα επαληθεύουν την ταχύτητα και την ακρίβεια της προτεινόμενης μεθόδου.



**ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ**: Αρχιτεκτονική Υπολογιστών

**ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ**: Χαρακτηρισμός Περιθωρίων Σχεδίασης, Ενεργειακά-Αποδοτικοί Υπολογισμοί, Πολυπύρηνοι Επεξεργαστές, Undervolting, Διαγνωστικοί Ιοί, ARMv8

*To my beloved late grandmother Kiki and to my dear late uncle Manoli.*

## AKNOWLEDGMENTS

First of all, I would like to thank my advisor, professor Dimitris Gizopoulos and his PhD candidates, Athanasios Chatzidimitriou, Manolis Kaliorakis and George Papadimitriou. Without their help and steering none of this would be possible. Secondly, I would like to thank my family and friends for their support all these years in my academic journey. Lastly, I would like to thank professor Panagiotis Rondogiannis for his guidance throughout my undergraduate and graduate academic career.

# CONTENTS

1. INTRODUCTION ..... 14

2. MICROPROCESSOR ARCHITECTURE ..... 16

2.1 Cache Entry ..... 19

2.2 Associativity ..... 19

2.3 Replacement Policy ..... 20

2.4 Write Policy ..... 21

2.5 Virtually and Physically Addressed Caches ..... 22

2.6 Page Coloring ..... 24

2.7 Data Prefetching ..... 25

2.8 Error Correction and Error Detection ..... 26

3. DIAGNOSTIC MICRO-VIRUSES OVERVIEW ..... 28

3.1 L1 Data Cache ..... 28

3.2 L1 Instruction Cache ..... 30

3.3 Unified L2 Cache ..... 32

3.4 Unified L3 Cache ..... 35

3.5 ALU ..... 36

3.6 FPU ..... 37

3.7 Pipeline ..... 38

4. EXPERIMENTAL EVALUATION ..... 39

4.1 Benchmarks vs Diagnostic Micro-Viruses ..... 39

4.2 Observations ..... 44

5. MICRO-VIRUS DEVELOPMENT & VALIDATION ON RASPBERRY PI ..... 46

5.1 L1 Data Cache ..... 48

5.2 L1 Instruction Cache ..... 48

5.3 Unified L2 Cache ..... 48

6. RELATED WORK ..... 52

7. CONCLUSION AND FUTURE WORK ..... 53

ABBREVIATIONS - ACRONYMS ..... 55

ANNEX I ..... 56

ANNEX II ..... 82

REFERENCES ..... 85

# LIST OF FIGURES

Figure I: X-Gene 2 micro-server power domains block diagram. The outlines with dashed lines present the independent power domains of the chip. .....17

Figure II: Virtual-to-Physical Address Translation .....24

Figure III: Cache lines' 64 bits are zeros .....29

Figure IV: Cache lines' 64 bits are ones. .....30

Figure V: Cache lines with checkerboard pattern .....30

Figure VI: A 256KB 32-way set associative L2 cache .....34

Figure VII: Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed Diagnostic Micro-Viruses for TTT (nominal) chip .....40

Figure VIII: Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed Diagnostic Micro-Viruses for TFF (corner part) chip .....40

Figure IX: Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed Diagnostic Micro-Viruses for TSS (corner part) chip .....41

Figure X: Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed L3 Diagnostic Micro-Viruses for all chips. Undervolting experiments occurred in SoC domain where the L3 cache takes place .....41

Figure XI: Detailed Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed Diagnostic Cache Micro-Viruses for TFF (corner part) chip .....42

Figure XII: Detailed Maximum Vmin among 10 SPEC CPU2006 benchmarks and all the proposed Diagnostic Micro-Viruses for TFF (corner part) chip .....42

Figure XIII: Maximum Vmin for the proposed L1 Instruction Cache Micro-Virus for TFF (corner part) chip and two different CPU frequencies .....43

Figure XIV: Guardband for all chips and all cores based on SPEC CPU 2006 benchmarks .....44

Figure XV: Guardband for all chips and all cores based on the proposed micro-viruses .....44

# LIST OF TABLES

Table I: Basic Characteristics of X-Gene 2. .....16

Table II: X-Gene 2 Caches Characteristics. .....18

Table III: ARMv8 Instructions used in the L1 Instruction Diagnostic Micro-Virus. The rightmost column presents the encoding of each instruction in bit granularity to demonstrate that all the bits in the cache line get flipped.....31

Table IV: L2 Cache Blocks Refills for 4KB and 64KB physical pages of a user-space variation of the L2 cache micro-virus during test phase. .....34

Table V: Basic Characteristics of Raspberry Pi 3 Model B. .....46

Table VI: Raspberry Pi 3 Model B Caches Characteristics. .....47

Table VII: L1 Data Accesses and Refills for X-Gene 2 and Raspberry Pi 3 Model B L2 micro-virus with a stride accessing of the array with a block stride of one block. .....49

Table VIII: Block Accessed for block stride of one block and eight blocks.....50

Table IX: L1 Data Accesses and Refills L2 micro-virus of Raspberry Pi 3 Model B with a stride accessing of the array with a block stride of eight blocks. .....51

# 1. INTRODUCTION

Transistors miniaturization due to the capabilities provided by the modern manufacturing process is also accompanied by process variations that affect transistor features (length, width, oxide thickness etc.) and consequently, affect their operation. Apart from these variations that are classified as static because they remain unchanged after the end of the fabrication period, transistor aging and dynamic variations in supply voltage caused by different workloads interactions can also affect the functionality of modern multicore CPU chips. To protect the chips from such unpredicted phenomena, microprocessor architects choose to introduce pessimistic voltage and frequency margins that lead to significant energy waste.

The minimization of such energy waste without sacrificing performance is a primary concern of microprocessor designers, but it requires a deep understanding of the voltage and frequency margins in order to ensure the correct execution of a chip in off-nominal conditions. The characterization procedure to identify these margins becomes more and more difficult and time consuming in the modern multicore CPU chips, as the systems become more and more complex and non-deterministic and the population of cores is rapidly increasing. In such multicore CPUs, there are significant opportunities for energy savings, because the divergences of the safe margins are remarkable among the cores of the same chip, among the different workloads that can be executed on different cores of the same chip and among the chips of the same type.

Trying to benefit from these divergences, several techniques have been proposed until now either at the software or at the hardware level aiming to reach the best trade-off between energy efficiency and performance. For instance, in Dynamic Voltage and Frequency Scaling (DVFS) [1], voltage and frequency are scaled during epochs where peak performance is not required, while some other techniques try to predict the safe percentage of chip undervolting without corrupting the correct execution of the program [2]. Moreover, some system level approaches (that have been evaluated only in simulators) propose scheduling algorithms in multicore chips that are based on the variation of the lowest safe voltage margin ($V_{min}$) of the different cores of the same chip to effectively allocate hardware resources to software tasks [3] [4] [5] [6]. Finally, at the hardware level, several methods have been proposed to reach the lowest safe voltage limit of the chip, without significant loss of performance [7] [8]. Unfortunately, these methods demand significant area, design, test and characterization overheads. None of these methods can be implemented without accurate identification of the safe voltage and frequency limits that differ from core-to-core, chip-to-chip and workload-to-workload.

To accurately identify these limits of a real multicore system requires multiple execution of a large number of real workloads in all the cores of the chip (or in all different chips of a system), for different voltage and frequency values. For instance, to identify the $V_{min}$ of each of the eight cores of the AppliedMicro's (APM) X-Gene 2 micro-server that is the experimental vehicle of this study, we used 10 SPEC CPU2006 benchmarks with 20 minutes (on average) execution time and we repeated each experiment 10 times starting from the nominal voltage value (980mV) until their crash voltage value (~870mV), we needed about 6 months for a complete characterization for all the cores of three different chips. This excessively long time makes the characterization of each chip that is ready to be released to the market infeasible, forcing the manufactures to introduce the same pessimistic guardband for all the cores of the same multicore chips. Therefore, the need to create benchmarks that are able to reveal the $V_{min}$ of each core

of a multicore chip (or the $V_{min}$ of different chips) in short time, in conjunction with all the insights needed to understand the behaviour of a system in off-nominal conditions is indisputable.

In this thesis, we propose the development of dedicated programs (diagnostic micro-viruses) that aim to stress the fundamental hardware components of three different chips (one “nominal” and two corner parts) of Applied Micro’s (APM) X-Gene 2 micro-server family, that are ARMv8-based multicore CPUs manufactured in 28nm. With our proposed diagnostic micro-viruses, we effectively stress all the main components of the chip:

a) the caches (the first level data and instruction caches, the unified L2 caches and the last level L3 cache of the chip)

b) two main components of the pipeline (the ALU and the FPU).

These diagnostic micro-viruses are executed in very short time (4 days for the entire massive characterization campaign of the three 8-core chips) compared to normal benchmarks such as those of the SPEC CPU2006 suite. The micro-viruses purpose is to reveal the safe voltage margins of each core of the multicore chip and also all the insights of the behaviour of the chips when they operate in unsafe voltage conditions.

The rest of the thesis is organized as follows: in Section II, we describe all the details of the microprocessor architecture, in Section III we discuss all the details of the diagnostic micro-viruses that were developed in this study. Section IV presents our findings and the comparison between the SPEC benchmarks and the proposed diagnostic micro-viruses, Section V presents the side story of developing and porting the diagnostic micro-viruses on a Raspberry Pi 3, and finally in the two final sections we present the related work and we conclude our study and future works, respectively.

# 2. MICROPROCESSOR ARCHITECTURE

In this section, we present all the details of the system architecture of the APM X-Gene 2 micro-server (voltage and frequency domains) along with all its microarchitectural details. The APM X-Gene 2 micro-server consists of eight 64-bit ARMv8-compliant cores. The basic microarchitectural details of the pipeline of each core are summarized in Table I.

**Table I: Basic Characteristics of X-Gene 2.**

| Parameter | Configuration |
|---|---|
| **CPU** | 8 Cores, 2.4GHz |
| **ISA** | ARMv8 (AArch64, AArch32, Thumb) |
| **Pipeline** | 64-bit OoO |
| **Issue Queue** | 4-issue |
| **ALU** | 1 single & 1 simple/complex integer arithmetic instructions |
| **FPU** | IEEE 754 (single and double precision) |
| **Integer physical register file** | 80 entries, 64-bit |
| **Floating point and SIMD physical register file** | 80 entries, 64-bit |
| **Page Size** | 4KB |

The X-Gene 2 system consists of a Power Management processor (PMpro) and a Scalable Lightweight Intelligent Management processor (SLIMpro). The first is a 32-bit dedicated processor that provides advanced power management capabilities such as multiple power planes and clock gating, thermal protection circuits, Advanced Configuration Power Interface (ACPI) power management states and external power throttling support, while the latter is a 32-bit dedicated processor that monitors system sensors, configures system attributes (e.g. regulate supply voltage, change DRAM refresh rate etc.) and accesses all error reporting infrastructure, using an integrated I2C controller as the instrumentation interface between the X-Gene 2 cores and this dedicated processor. SLIMpro can be accessed by the system's running Linux Kernel.

X-Gene 2 has three independently regulated power domains that are presented in Figure I.

1. **PMD (Processor Module)**: Each PMD contains two ARMv8 cores. Each of the two cores has separate instruction and data L1 caches, while they share a unified L2 cache. The operating voltage of all four PMDs together can change with a granularity of 5mV beginning from 980mV. While PMDs operate at the same voltage, each PMD can operate in a different frequency. The frequency can range from 300 MHz up to 2.4 GHz with 300 MHz steps.
2. **PCP (Processor Complex)/SoC**: It contains the L3 cache, the DRAM controllers, the central switch and the I/O bridge. The PMDs do not belong to the PCP/SoC power domain. The voltage of the PCP/SoC domain can be independently scaled downwards with a granularity of 5mV beginning from its nominal value (950mV).
3. **Standby Power Domain**: This includes the Power Management processor (PMpro) and a Scalable Lightweight Intelligent Management processor (SLIMpro) microcontrollers and interfaces for I2C buses, which monitor and regulate the voltage of the X-Gene 2 microprocessor.

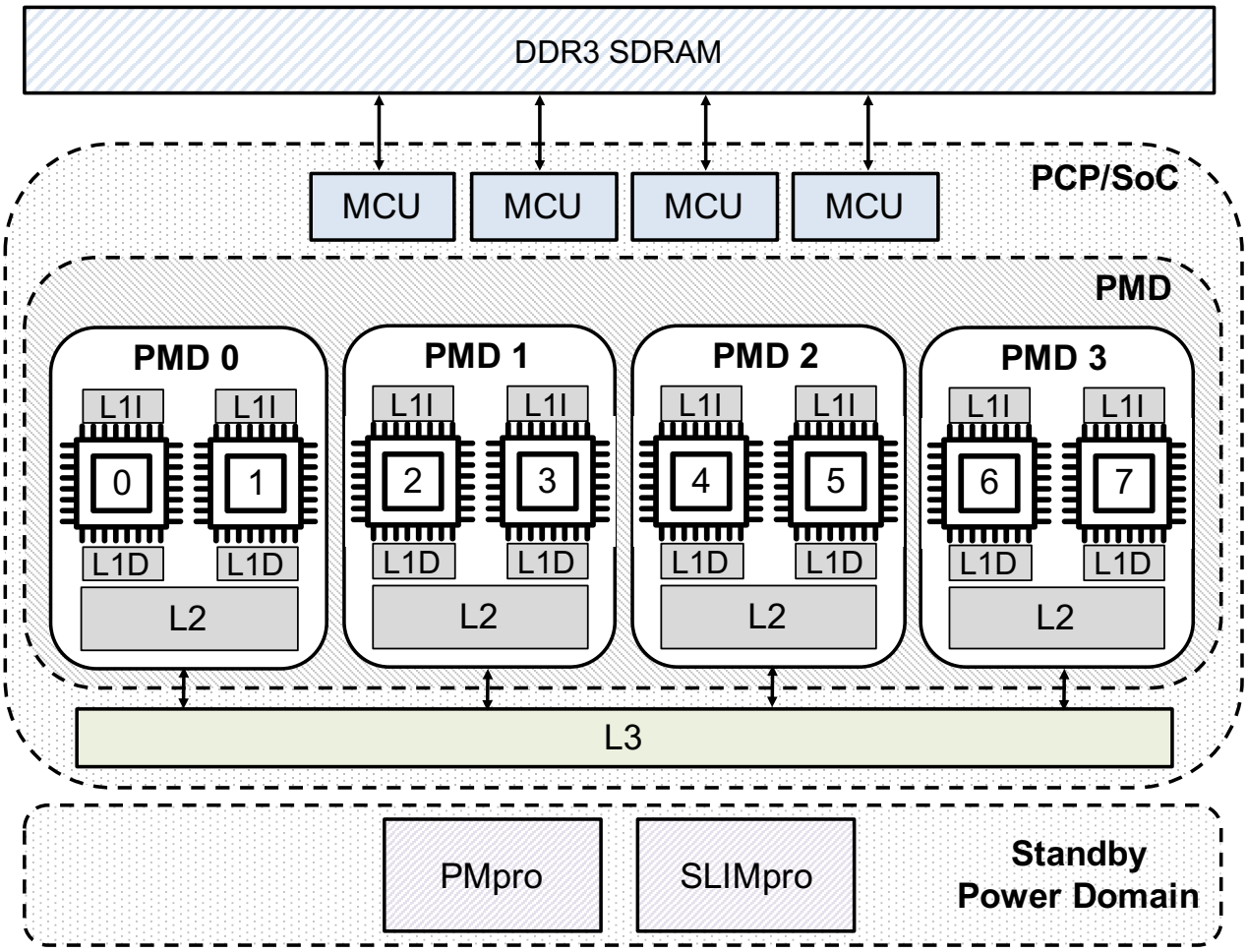


**Figure I: X-Gene 2 micro-server power domains block diagram. The outlines with dashed lines present the independent power domains of the chip.**

The basic features of the all the caches of the X-Gene 2 are summarized in Table II.All the characteristics of the X-Gene 2 microprocessor architecture are very important for the construction of the diagnostic micro-viruses that we present in the following section.

**Table II: X-Gene 2 Caches Characteristics.**

| | L1 Instr | L1 Data | L2 | L3 |
|---|---|---|---|---|
| **Size** | 32 KB | 32 KB | 256 KB | 8 MB |
| **# of Ways** | 8 | 8 | 32 | 32 |
| **Block Size** | 64 B | 64 B | 64 B | 64 B |
| **# of Blocks** | 512 | 512 | 4096 | 131,072 |
| **# of Sets** | 64 | 64 | 128 | 4096 |
| **Write Policy** | - | Write-Through | Write-Back | - |
| **Write Miss Policy** | No-write allocate | No-write allocate | Write allocate | - |
| **Organization** | PIPT[1] | PIPT | PIPT | PIPT |
| **Prefetcher** | YES | YES | YES | NO |
| **Scope** | Per Core | Per Core | Per PMD | Shared |
| **Protection** | Parity Protected | Parity Protected | ECC Protected | ECC Protected |

Before proceeding to the next section, a quick overview of the above cache characteristics is presented for the shake of completeness.

[1] Physically Indexed, Physically Tagged

## 2.1 Cache Entry

Even when one requests a single memory word, i.e. when accessing the first element of an integer array, data are transferred between the main memory and the caches in blocks of fixed size, called cache blocks or cache lines (this is the minimum unit of information that exists in a cache memory) [34]. When a cache line is copied from the main memory into the cache, a cache entry is created; this entry consists of not only the actual requested data, but also an upper portion of the requested memory address (called tag) and some flag bits (such as the valid bit that indicates whether an entry contains a valid address or not). When there is a request for a memory address, the CPU has to check for the presence of a corresponding cache entry in the cache. The cache checks for the requested data in all the cache lines that might contain it (more on this in the next subsection). If the processor finds the corresponding cache entry in the cache, a cache hit has occurred (or in other words the request hits the cache) and after that the CPU reads the data from the cache. However, if the processor does not find the memory location in the cache, a cache miss has occurred (or the request misses the cache) and the CPU has to allocate a new entry in the cache in order to copy the data from the main memory or from a lower-level cache (allocate-on miss).

## 2.2 Associativity

This is a placement policy that decides where in the cache a copy of a particular entry of main memory (block) can reside [34]:

a) If a block can be placed in any location in the cache, the cache is called fully associative (in other words a block in memory may be associated with any entry in the cache). To find a given block in a fully associative cache, all the entries in the cache must be searched because a block can be placed in any one of them. Practically, a parallel search is employed with a comparator associated with each cache entry. However, these comparators significantly increase the hardware cost, effectively making fully associative placement efficient only for caches with small numbers of blocks.

b) If each entry in main memory can go in just one place in the cache, the cache is called direct mapped (in other words there is a direct mapping from any block in memory to a single entry in the cache). This placement organization, as each location of the main memory can go in only one place in the cache, can also be called "one-way set associative". Moreover, a direct mapped cache does not have a replacement policy, since there is no choice of which cache entry's contents to evict; if two memory addresses map to the same cache entry, they will keep knocking each other out.

c) The middle range of designs between a direct mapped cache and a fully associative cache is called N-way set associative, in which each entry in main memory can map to any one of N places in the cache. For instance, an eight-way set associative cache can hold up to eight blocks in each set. The number of blocks in a set is also known as the cache associativity or set size. Each block in each set has a stored tag which identifies each block. First, the (set) index of the address of a request is used in order to access the correct set. Then, comparators are used to compare all the tags of the selected set with the incoming tag from the request. If a match is found, the corresponding location is accessed, otherwise, as before, an access to the main memory or lower-level caches is made.

Based on the above, choosing the right value of associativity involves a trade-off between performance efficiency and area cost. If there are eight places to which the placement policy could have mapped a memory location, then to check for a corresponding cache entry in the cache, eight cache entries must be searched in parallel. On the one hand, employing more comparators requires more power and die area, while on the other, caches with higher associativity suffer from fewer conflict misses and thus offer better performance [34].

## 2.3 Replacement Policy

In order to make room for a new entry on a cache miss (if the corresponding set is full), the cache may have to evict one of the existing entries (of this particular set). The heuristic that it uses to choose the entry of the set to evict is called cache replacement policy (except for caches with direct mapping, which does not need a replacement algorithm) [34]. The basic problem with any replacement policy is that it must predict which existing cache entry is least likely to be used in the future. However, predicting future access patterns is difficult, so there is no perfect way to choose among the great variety of replacement policies available. Nevertheless, a good replacement algorithm can yield significantly higher performance than a bad replacement algorithm and thus the choice is critical for the cache performance of modern systems. Moreover, the replacement mechanism must be implemented totally in hardware, preferably such that the selection (of the soon to be evicted block) can be performed before the new block is fetched from lower-level caches or the main memory (in order to maintain high performance) [34].

One of the most popular and widely adopted replacement policies is the so called least-recently used (LRU), which (as its name suggest) replaces the least recently accessed block [34]. The least recently used (LRU) algorithm can only be implemented fully when the number of cache blocks is small (more about that in the following paragraphs).

Another replacement policy, that in practice appears to have good performance is the random replacement policy (RR): candidate blocks are randomly selected, possibly using some hardware assistance, with no regard to previous memory references or previous evictions. The biggest advantage of random replacement policy in comparison to the LRU replacement policy is its significantly lower implementation cost while preserving a good miss rate (for a two-way set-associative cache, random replacement has a miss rate about 1.1 times higher than LRU replacement) [34].

Due to the high cost of LRU and the lack of solid performance guarantees of random replacement, Pseudo-LRU was introduced. Pseudo-LRU (or PLRU) is a family of replacement algorithms which improves on the performance of the Least Recently Used (LRU) algorithm by selecting which block to evict based on approximate measures of age rather than maintaining the exact age of every cache block [34]. On the one hand, PLRU typically has a slightly worse miss ratio due to employing approximate measures of age, while on the other it has a slightly better latency, consumes slightly less power than LRU and imposes lower overheads compared to LRU. One of the many implementations of PLRU is Bit-PLRU, which stores one status bit for each cache line; these bits are called MRU-bits. Every access to a line sets its corresponding MRU-bit to 1, indicating that the line was recently used. Whenever the last remaining 0 bit of a set's status bits is set to 1, all other bits are reset to 0. At cache misses, the line with lowest index whose MRU-bit is 0 is replaced.

To better understand the aforementioned problem regarding the cost of the LRU, let’s assume we have an 8-way set-associative cache: A traditional LRU replacement algorithm would essentially be assigning each cache line an exact index in the order of

usage. One can also think of that as an "age". So, each of the 8 cache lines in each set would require an index of 3 bits (since we need to count 8 distinct ages) stating its location in the LRU order - this means 3 bits * 8 ways, per each set of the cache. In the general case of n ways, a standard implementation of the LRU replacement policy would need $\log_2 n$ bits per line, or $n*\log_2 n$ bits per set. On the contrary, a typical implementation of the Bit-Pseudo-LRU replacement policy would only require 8 bits for the entire set (1 bit per cache line of the set) in this particular case (or in general: #ways bits). All in all, the bigger the associativity of a cache, the more prohibitively expensive is the cost of implementing LRU replacement policy.

## 2.4 Write Policy

When a program stores data to the memory, the write policy defines how data consistency is maintained among different levels of the memory hierarchy. This is typically done by deciding when should the new or updated data be pushed to the lower memory levels, offering different performance options [34]. The two basic write policies are the following ones:

a) Write-through: stores update the cache where the request hits and are also forwarded to the next memory level, ensuring that there is always an up-to-date copy of the data in the lower memory hierarchy.

b) Write-back (also called write-behind): writes are not immediately mirrored to the lower levels of the memory hierarchy; the cache instead tracks which locations have been written over, marking them as dirty with the help of the appropriate dirty bit. The write to the backing store is delayed until the dirty cache blocks containing the updated data are about to be evicted from the cache by new cache lines. Based on the above, a read miss in a write-back cache may sometimes require not only a read access but also a write access: before reading the new block from memory, the cache has to write the evicted dirty block to main memory.

In general, write-back schemes can improve performance when processors can generate writes as fast or faster than the writes can be handled by main memory; a write-back scheme is, however, more complex to implement than a write-through one [34].

One should also note that a cache can be write-through, but the writes may be held in an intermediate store data queue (also called write buffer), so that multiple stores can be processed together (which can improve bus utilization and thus the overall performance of a cache subsystem). However, if the rate at which the memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help, because writes are being generated faster than the memory system can accept them [34].

Furthermore, apart from the aforementioned write hit policies there are also two approaches for situations of write-misses [34]:

a) Write allocate: the block corresponding to the missed-write memory address is copied to the cache and this is followed by a write-hit; thus, write misses are treated like read misses in this scheme.

b) No-write allocate: the block corresponding to the missed-write memory address is not copied to cache, and data is forwarded directly to the lower memory level. In this approach, writes are not cached until a read operation has been performed to this particular cache block.

Both write-through and write-back policies can use either of these write-miss policies, but usually they are paired in the following way [34]:

a) A write-back cache employs a write allocate miss policy, hoping for subsequent writes to the same location, which is now cached. This will eliminate extra memory accesses and results in very efficient execution in comparison to a write-back cache with no-write allocate.

b) A write-through cache uses no-write allocate. In this case, subsequent writes have no advantage, since they still need to be written directly to the backing store.

## 2.5 Virtually and Physically Addressed Caches

Caches can be categorized based on whether the (set) index or tag are derived from the physical or the virtual addresses:

a) Physically indexed, physically tagged (PIPT) caches use the physical address to determine both the (set) index and the tag. While this is the simplest organization and avoids synonyms problem (where several virtual addresses map to the same physical addresses), it is also slow, as the virtual address has to be translated into a physical address (which could potentially involve a TLB miss and thus an access to main memory) before the cache lookup can take place (even for determining the appropriate set). However, this problem is considerably alleviated with good TLB hit rates as a successful TLB lookup can be completed very fast. A high-level representation of the virtual-to-physical translation that takes place before the cache lookup is presented in Figure II. This figure assumes 4KB pages, 64-bit virtual addresses and 39-bit physical addresses.

b) Virtually indexed, virtually tagged (VIVT) caches use the virtual address to determine both the (set) index and the tag. On the one hand, this caching scheme can result in much faster lookups in comparison to a PIPT one, since the Memory Management Unit (MMU) does not need to be consulted first to determine the physical address for a given virtual address, while on the other VIVT suffers from aliasing problems, where several different virtual addresses may refer to the same physical address (i.e. in Unix, executable code is typically mapped into a region shared between all processes that execute the same program and most multiprocessing systems support the creation of shared memory where the same physical memory can be mapped to multiple processes). The result of the aforementioned problem is that such addresses would incorrectly be cached separately despite referring to the same physical memory, causing coherency problems. Another problem in this organizations is homonyms, where the same virtual address maps to several different physical addresses (which is a standard situation in modern multitasking systems with virtual memory). In order to make things even more difficult, it is not possible to distinguish these mappings just by looking at the virtual index itself. According to the literature, potential solutions for the homonyms problem are among others: flushing the entire cache after a context switch (which is pretty costly), forcing address spaces to be non-overlapping, tagging the virtual address with an address space ID (ASID), or employing physical tags. Additionally, there is a problem that virtual-to-physical mappings can gradually change, which would require flushing a portion (if not all) of the cache, as the virtual addresses would no longer be valid. All in all, there are hardware workarounds for some of these problems, but the most efficient solution is to simply use VIPT or PIPT caches.

c) Virtually indexed, physically tagged (VIPT) are a natural evolution of VIVT caches. VIPT caches use the virtual address to determine the (set) index and the physical address to determine the tag. The main advantage over a PIPT scheme is lower latency, as the appropriate set can be resolved in parallel with the virtual to physical address translation, however the tag cannot be compared until the physical address is available. The advantage over VIVT is that since the tag has the physical address, the cache can detect homonyms. In general, this organization is a compromise between a PIPT and VIVT organization.

d) Physically indexed, virtually tagged (PIVT) caches are often claimed in literature to be non-existing. However, the MIPS R6000 employs this organization and is the sole known implementation of a PIVT cache [38]. The R6000 is implemented in emitter-coupled logic, which is an extremely fast technology not suitable for large memories such as a TLB. The R6000 solves the issue by putting the TLB memory into a reserved part of the second-level cache having a tiny, high-speed TLB "slice" on chip. The cache is indexed by the physical address obtained from the TLB slice. However, since the TLB slice only translates those virtual address bits that are necessary to index the cache and does not use any tags, false cache hits may occur, which is solved by tagging with the virtual address [38].

ARMv7 and ARMv8 processors have PIPT data caches (or at least are required to behave as if they do). That is, they have physically indexed, physically tagged data caches, and no page coloring restrictions apply. However, one should note that PIPT caches could (in certain situations) still benefit from page coloring as it can improve cache line eviction behavior (more on this in the upcoming subsection). Moreover, they can employ VIPT instruction caches (i.e. Cortex-A8 and Cortex-A9 both have VIPT instruction caches) and even VIVT instruction caches are allowed to an extent. Lastly, they can also have PIPT instruction caches.

A rather interesting case is the one of the ARM Cortex-A73, where the L1 data cache is organized as a Virtually Indexed, Physically Tagged (VIPT) cache. However, the ARMv8 technical manual states that "In the L1 data memory subsystems, aliases are handled in hardware and from the programmer's point of view, the data cache behaves like an eight-way set associative PIPT cache (for 32KB configurations) and a 16-way set associative PIPT cache (for 64KB configurations)."

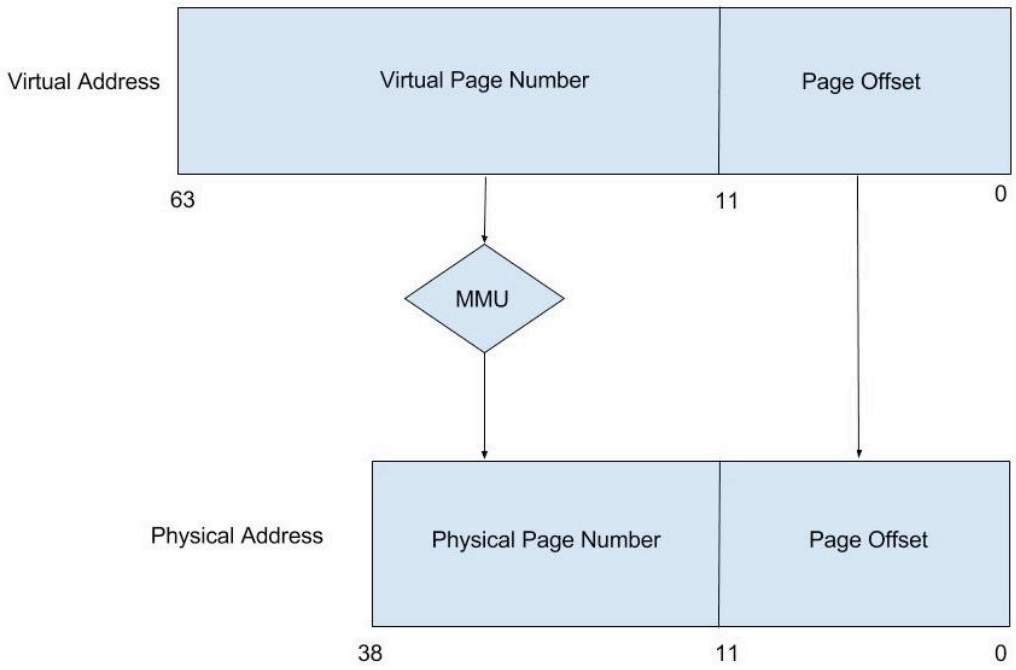


**Figure II: Virtual-to-Physical Address Translation**

## 2.6 Page Coloring

Page coloring (or cache coloring) is a software technique that operates at the level of the operating system (OS) or virtual machine monitor by influencing the translation of virtual addresses to physical ones. The color of a physical page is defined as the bits in the intersection of the physical page number field and the cache (set) index field of the page address. Page coloring aims to provide a uniform distribution of page colors for the physical pages assigned to a set of virtual pages, which in turn ensures maximum utilization of all the available cache sets (or in other words 100% occupancy) [40].

Assuming that in the physical address layout we find that the page number field overlaps the cache index field by 2 bits, giving 4 possible page colors. If the kernel does not implement a page coloring technique and unintentionally mapped virtual pages $V_0$, $V_1$, $V_2$, …, $V_N$ to physical pages $P_0$, $P_1$, $P_2$, …, $P_3$, respectively, and those physical pages happened to be of the same page color, then cache lines underlying pages $V_0$, $V_1$, $V_2$, …, $V_N$ would reside in just one quarter of the available cache sets, underutilizing parts of the cache and creating a “hot spot” in other sectors. Page coloring attempts to avoid such unfavorable assignments of physical pages to virtual addresses (i.e. a simple page coloring algorithm could achieve the aforementioned goal by enforcing that the page color of each virtual page belonging to a set of contiguous virtual pages should match the page color of the corresponding physical page).

As one can easily understand after the example presented above, a virtual memory subsystem that lacks cache coloring is less deterministic in terms of cache performance, as differences in page allocation from one program run to another can lead to significant differences in terms of performance due to variation in conflict misses. Afek et al. in [40] identify the problem of inter-block cache index conflict misses arising from excessive regularity in addresses returned by memory allocators on a Linux/x64 Nehalem system and show that the placement policies of malloc (and other related

allocators), by virtue of conflict miss rates, can have a significant impact on application performance.

All in all, cache coloring does not alter the way the cache works at all; it is just an abstraction to describe a finer-grained control of how the virtual-to-physical memory mapping is set up in order to maximize the number of pages cached by the CPU and thus make virtual memory as deterministic as possible regarding cache performance. However, one should note that a page coloring algorithm adds a significant amount of complexity to the virtual memory allocation subsystem.

While page coloring is employed in some operating systems such as Solaris and FreeBSD [39], the Linux community decided not to implement a page coloring algorithm in the Linux kernel. According to Linus Torvalds' emails found in the UseNet archives "*There have been at least four different major cache coloring trials for the kernel over the years. This discussion has been going on since the early nineties. And none of them have worked well in practice.*"

Moreover, Larry McVoy, CEO of BitMover (the company that developed BitKeeper, a versioning control system that was used from February 2002 to early 2005 to manage the source code of the Linux kernel), states in a mail that can be also found in the UseNet archives "*Linus doesn't like it because it adds cost to the page free/page alloc paths, they now have to go put/get the page from the right bucket. He also says it's pointless because the caches are becoming enough associative that there is no need and he's mostly right. Life can really suck on small cache systems that are direct mapped, as are some embedded systems, but that's life. It's a tradeoff.*"

## 2.7 Data Prefetching

Data prefetching is a widely-adopted optimization technique, of the standard demand fetching cache algorithm, that aims to improve the cache performance by decreasing the number of compulsory misses (or cold start misses) of an executing program. The way to achieve the aforementioned goal is by fetching instructions and data from their original storage (in slower memory) to a faster cache memory before they are actually requested; hence the term prefetch [35]. The source for the prefetch operation is the backing store (main memory or lower memory levels in the cache hierarchy).

There are two main types of cache prefetching [37]:

a) Hardware based prefetching: Hardware based prefetching is typically accomplished by having a dedicated hardware mechanism in the processor that observes the stream of instructions or data being requested by the executing program, recognizes which instructions or data in nearby memory locations the program might access shortly after (spatial locality) and prefetches them into the processor's cache.

b) Software based prefetching: Software based prefetching is solely accomplished with the help of the compiler; during the compilation phase of the program, the compiler performs some kind of (static) analysis of the code and accordingly inserts "prefetch" instructions wherever it deems fit.

One of the most widely employed hardware based prefetching technique in use these days is stream buffers. This technique was originally proposed by Norman Jouppi in 1990 [36] and numerous variations and improvements of this method have been proposed since. The cornerstone of all these variations is that the cache miss address (and k subsequent memory addresses) are fetched into a separate buffer of depth k. This buffer is called a stream buffer and is a separate entity from the cache. If the address associated with the prefetched blocks matches the requested address

generated by the executing program, then the CPU consumes data or instructions from the stream buffer.

Whenever the prefetch mechanism detects a miss on a memory block, say A, it allocates a stream to begin prefetching successive blocks from the missed block onward. If the stream buffer can hold up to four blocks, then we would prefetch A+1, A+2, A+3, A+4 and hold those in the allocated stream buffer. If the processor consumes A+1 next, then it shall be moved "up" from the stream buffer to the processor's cache. The first entry of the stream buffer would now be A+2 and so on. This pattern of prefetching successive blocks is called Sequential Prefetching and it is mainly employed when contiguous locations are to be prefetched. For example, it is used when prefetching instructions.

This mechanism can be scaled up by adding multiple stream buffers; each of these buffers would maintain a separate prefetch stream. For each new miss, there would be a new stream buffer allocated and it would operate in a similar way as described above. The ideal depth of the stream buffer is something that is subject to experimentation against various benchmarks and depends on the rest of the microarchitecture involved.

An example of such a mechanism is given in the ARMv8 Manual [32] for ARM Cortex-A53. According to the manual, the L1 data cache implements an automatic prefetcher that monitors cache misses in the core (the L1 data cache is private for each core). When an access pattern is detected, the prefetcher starts linefills in the background. The prefetcher is able to recognize a sequence of data cache misses at a fixed stride pattern that lies in four cache lines, plus or minus. Moreover, any intervening stores or loads that hit in the data cache do not interfere with the recognition of the aforementioned cache miss pattern. Finally, the prefetcher supports, by default, two independent data prefetch streams (the maximum number of independent streams for Cortex-A53 is four).

Having analyzed the most common hardware based prefetching, up next is compiler directed prefetching [37]. Compiler directed prefetching is widely used within loops with a large number of iterations that access contiguous memory elements. In this technique, the compiler predicts future access patterns and inserts a prefetch instruction based on the miss penalty and execution time of the instructions. These prefetches are non-blocking memory operations, i.e. these memory accesses do not interfere with the actual memory accesses and they do not change the state of the processor or cause page faults (the latter is quite important). The main advantage of software prefetching is that it reduces the number of compulsory (or cold-start) cache misses.

All in all, software prefetching works well only with loops where there is regular array access pattern as the programmer or the compiler has to manually insert the prefetch instructions. On the contrary, hardware prefetchers work dynamically based on the program's runtime behavior and are able to detect even more complex access patterns. Moreover, it is widely accepted that hardware prefetching imposes significantly less CPU overhead when compared to software prefetching [37].

## 2.8 Error Correction and Error Detection

Error Detecting Code (EDC) is a code that enables the detection of an error in data, but not the precise location and, hence, correction of the error [34].

Error Correcting Code (ECC) is a code that is able not only to detect errors in data but also correct (some of) them. ECC memory can be found mostly in computers where data corruption is not tolerable due to the nature and the importance of the performed computations.

Typically, ECC memory maintains a memory system immune to single-bit errors: the data that is read from each word is guaranteed to be always the same as the data that had been written to it, even if one (and sometimes more) stored bits have been flipped to the wrong state.

Most non-ECC memory cannot even detect errors. However, some non-ECC memory with parity protection allows error detection but not error correction; a parity bit is an extra bit added to a word (or a cache line depending on the implementation) in order to ensure that the total number of ones is even or odd. Generally speaking, parity bits are used as the simplest form of error detecting code.

For instance, in the case of a single-bit memory error in the L1 data cache (which is parity protected), the entire cache line is invalidated and it is refetched from the L2 cache or the external memory. However, if there is a two-bit memory error in the L1 data cache, then the 1-bit parity scheme (like the one in the X-gene 2) will not detect the error. (Actually, a 1-bit parity scheme can detect any odd number of errors; however, the probability of having three errors is much lower than the probability of having two, so, in practice, a 1-bit parity code is limited to detecting a single bit of error.)

The way a corrected error is displayed in an ARMv8 compliant CPU, like the one in the X-Gene 2, is the following:

---

May 7 21:10:54 tigs-c0-021 kernel: {2}[Hardware Error]: Hardware error from APEI Generic Hardware Error Source: 8

May 7 21:10:54 tigs-c0-021 kernel: {2}[Hardware Error]: It has been corrected by h/w and requires no further action

May 7 21:10:54 tigs-c0-021 kernel: {2}[Hardware Error]: event severity: corrected

May 7 21:10:54 tigs-c0-021 kernel: {2}[Hardware Error]: Error 0, type: corrected

May 7 21:10:54 tigs-c0-021 kernel: {2}[Hardware Error]: section_type: general processor error

May 7 21:10:54 tigs-c0-021 kernel: {2}[Hardware Error]: error_type: 0x01

May 7 21:10:54 tigs-c0-021 kernel: {2}[Hardware Error]: cache error

May 7 21:10:54 tigs-c0-021 kernel: {2}[Hardware Error]: level: 0

May 7 21:10:54 tigs-c0-021 kernel: {2}[Hardware Error]: processor_id: 0x0000000000000003

---

The above messages are part of the output of the dmesg command. Dmesg (display message or driver message) is a command on most Unix-like operating systems that prints the message buffer of the kernel and the output of this command typically contains the messages produced by the device drivers. These messages inform the user that a hardware error occurred, it was a cache error with error_type 0x01 and level 0 and it has been corrected and requires no further action. Based on the level attribute the user can infer that this error occurred in the L1 cache (however there is no way to tell if the error occurred in the L1 data cache or in the L1 instruction cache).

# 3. DIAGNOSTIC MICRO-VIRUSES OVERVIEW

For the construction of the proposed micro-viruses we followed two different principles for the tests that target the caches and the pipeline respectively:

1) **Caches**: For all levels of caches the goal of the developed micro-viruses is to flip all the bits of the structures with zeros and ones and to read the same hardware entries during the undervolting procedure in order to identify any corruptions of the written values compared to the golden values, which are not detected by the detection mechanisms of the microprocessor, such as the SECDED ECC.
2) **Pipeline**: For the pipeline, we developed dedicated benchmarks that stress: (i) the Floating-Point Unit (FPU), (ii) the integer Arithmetic Logical Units (ALUs) and (iii) the entire pipeline using a combination of loads, stores, branches, arithmetic and floating-point unit operations. The goal is to trigger the critical paths that could possibly lead to an error during off-nominal voltage conditions.

We developed all diagnostic micro-viruses by using a mix of C language and ARMv8 assembly instructions, to achieve the best execution of the code in the actual hardware of the machine. We describe the major aspects of this challenging micro-viruses design process in the following sub-sections. Moreover, the source code for all the micro-viruses can be found in ANNEX I.

## 3.1 L1 Data Cache

For the first level data cache of each core, we statically allocate an array in memory with the same size as the L1 data cache. As the L1 data cache is no-write allocate, before the first write of the desired pattern in all the words of the structure we had to read them first in order to bring all the blocks in the first level of data cache. Otherwise, the blocks would remain in the L2 cache and we would have only write misses in the L2 cache.

Moreover, due to the pseudo-LRU policy that is used in the L1 data cache, we had to read all the words of the cache three consecutive times (before the test begins), in order to ensure that all the blocks with the desired patterns are allocated in the first level data cache. We experimentally observed that a safe number of iterations is $\log_2$(# of ways) to guarantee that the L1 data cache is filled only with the data of the diagnostic micro-virus. With these steps, we are able to achieve 100% read hit in the L1 data cache during the execution of the L1D micro-virus in undervolting conditions. The way we measure the L1 data cache accesses and refills (along with other micro-architectural events) in order to calculate the above read hit rate is by leveraging the built-in performance counter[2].

Performance Monitor Unit (PMU) is a new feature in ARMv8-A architecture and it includes a 64-bit cycle counter along with a number of 32-bit event counters and control component. From a programmer's point of view, it is a handy tool for low-level performance monitoring, as one can easily access the processor status, like cycles, instructions executed, branch taken, cache refills and hits for different level of caches and many more from these PMU event counters. The PMU uses event ids to identify different events which can be categorized into architectural, microarchitectural and implementation specific. The actual events which are available is again implementation defined and thus one should refer the ARMv8 specification to get the complete list of events supported by the PMU. A complete list with an extensive description of the

[2] We developed a kernel module able to grant us access to the performance counters from user space.

architectural and microarchitectural events we monitored during the development of the diagnostic micro-viruses can be found in ANNEX II. The main advantage of employing this technique for performance monitoring over the traditional *perf* tool lies in the ability to monitor a particular piece of code (or various pieces of code) and not necessarily the whole program. Moreover, because ARMv8 allows user space access of the PMU counters from EL0, we developed a kernel module that enables user-mode access to these counters. These PMU counters and their associated control registers are accessible in the AArch64 Execution state with *MRS* and *MSR* instructions (*MRS* copies the value of a system register into a general-purpose register and *MSR* copies the value of a general-purpose register into a system register).

The L1 Data diagnostic micro-virus fills the L1 Data cache with three different patterns, each of which corresponds to a different virus test. These tests are the all-zeros (Figure III), the all-ones (Figure IV), and checkerboard (Figure V), as presented below. The checkerboard pattern differs from the all-zeros and all-ones patterns due to the fact that each cell's four neighbors (top, bottom, left, right) have different value from the cell itself; thus, with this pattern we can detect bridging faults (bridging faults are a subclass of coupling faults where a transition in cell x causes unwanted change in a neighbor cell y). To enable the self-checking property of the micro-virus (correctness of execution is determined by the virus itself and not externally), at the end of the test we check if each fetched word is equal to the expected value (the one stored before the test begins). If not, then we have identified Silent Data Corruption (SDC) compared to the program's output during nominal voltage conditions.

**Pattern 1**

| | way 0 | way 1 | way 2 | way 3 | ••• | way 7 |
|---|---|---|---|---|---|---|
| set 0 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 |
| set 1 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 |
| set 2 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 |
| ⋮ | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 |
| set 63 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 | 0x000…00 |

**Figure III: Cache lines' 64 bits are zeros**

**Pattern 2**

| | way 0 | way 1 | way 2 | way 3 | ••• | way 7 |
|---|---|---|---|---|---|---|
| set 0 | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF |
| set 1 | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF |
| set 2 | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF |
| ⋮ | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF |
| set 63 | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF | 0xFFF…FF |

**Figure IV: Cache lines' 64 bits are ones.**

**Pattern 3**

| | way 0 | way 1 | way 2 | way 3 | ••• | way 7 |
|---|---|---|---|---|---|---|
| set 0 | 0xAAA…AA | 0xAAA…AA | 0xAAA…AA | 0xAAA…AA | 0xAAA…AA | 0xAAA…AA |
| set 1 | 0x555…55 | 0x555…55 | 0x555…55 | 0x555…55 | 0x555…55 | 0x555…55 |
| set 2 | 0xAAA…AA | 0xAAA…AA | 0xAAA…AA | 0xAAA…AA | 0xAAA…AA | 0xAAA…AA |
| ⋮ | | | | | | |
| set 63 | 0x555…55 | 0x555…55 | 0x555…55 | 0x555…55 | 0x555…55 | 0x555…55 |

**Figure V: Cache lines with checkerboard pattern**

### 3.2 L1 Instruction Cache

The concept behind the L1 Instruction Cache micro-virus is to flip all the bits of the instruction encoding in the cache line. In the ARMv8 ISA there is no single pair of instructions that can be utilized to invert all 32 bits of an instruction word in the cache, so to achieve this we had to employ multiple instructions. The instructions listed in Table III are able to flip all the bits in the instruction cache from 0 to 1 and vice versa according to the Instruction Encoding Section of the ARMv8 Manual [32]. For instance, the first two instructions are able to flip all the bits of the immediate field [10:21], all the

bits of the destination [0:4] and source [5:9] register, and the fourth most important bit which in this case dictates if the instruction is an addition or a subtraction. Having flipped the aforementioned bits, the goal is to flip the most significant bits of the encoding that are mostly control bits, like bits [22:23] which are the ones that define the type of shift in the so called add with shifted register; this instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. For an arithmetic right shift these two bits equal to 10, while for a logical shift right these two bits equal to 10.

**Table III: ARMv8 Instructions used in the L1 Instruction Diagnostic Micro-Virus. The rightmost column presents the encoding of each instruction in bit granularity to demonstrate that all the bits in the cache line get flipped.**

| Instruction | Encoding |
|---|---|
| add x28, x28, #0x1 | 1001 0001 00 0000 0000 0001 11100 11100 |
| sub x3, x3, #0xffe | 1101 0001 00 1111 1111 1110 00011 00011 |
| madd x28, x28, x27, x27 | 1001 1011 00 0110 1101 1011 11100 11100 |
| add x28, x28, x27, asr #2 | 1000 1011 10 0110 1100 0010 11100 11100 |
| add w28, w28, w27, lsr #2 | 0000 1011 01 0110 1100 0010 11100 11100 |
| nop | 1101 0101 00 0000 1100 1000 00000 11111 |
| bics x28, x28, x27 | 1110 1010 00 1110 1100 0000 11100 11100 |

Each cache line of the L1 instruction cache is able to hold 16 instructions because each instruction is 32-bit in ARMv8 and the L1 Instruction Cache Block Size is 64 bytes. The size of each way of the L1 Instruction Cache is 32KB / 8 = 4KB and thus is equal to the page size which is 4KB. As a result, there should be no conflict misses when accessing a code segment with size equal to the L1 Instruction cache (the same argument holds also for the L1 Data Cache).

The method we perform the self-checking property in the L1 Instruction cache micro-virus is the following: The L1 cache array holds 8192 instruction (64 sets x 8 ways x 8 64-bit words in each cache line = 8192 instructions). We use 8177 instructions to hold the instructions of our diagnostic micro-virus, and the remaining 15 instructions (8177 + 15 = 8192) to compose the control logic of the self-checking property and the loop control. More specifically, we execute iteratively 8177 instructions and at the end of this block of code we expect the destination registers to hold a specific “signature” (the signature is the same for each iteration of the same group of instructions, but different among different executed instructions). If this “signature” is distorted then the diagnostic

micro-virus detects that an error occurred (for instance a bit flip in an immediate instruction resulted in the addition of a different value or a bit flip in one of the source registers resulted in reading wrong value from the register file) and records the location of the faulty instruction as well as the expected and the faulty signature for further diagnosis. We iterate this code multiple times and after that we continue with the next block of code.

As in L1 Data cache micro-virus, due to the pseudo-LRU policy that is used also in the L1 Instruction cache, we fetch all the instructions three consecutive times (log2(# of ways) = log28=3) before the test begins, in order to ensure that all the blocks with the desired instruction patterns are allocated in the first level instruction cache. With these steps, we achieve 100% read hit in the cache (and thus cache stressing) during the undervolting campaign.

### 3.3 Unified L2 Cache

The aim of the L2 diagnostic micro-virus is to stress all the bits of the L2 cache. To achieve this, we take into account the microarchitectural characteristics of X-Gene 2 for the L2 cache, while developing the code. As we have already discussed in Section 2, the L2 cache is a 32-way associative PIPT cache with 128 sets; thus, the bits of the physical address that determine the block placement in the L2 cache are bits [12:6] (as shown in Figure VI). Moreover, the page size we rely on is 4KB and consequently the page offset is consisted of the 12 less significant bits of the physical address. According to this, the most significant bit (bit 12) of the set index (see Figure VI) is not a part of the page offset. If this bit is equal to 1, then the block is placed in a set of the upper half of the cache, and in the same manner if this bit equals to 0, the block is placed in a set of the lower half of the cache. Bits [11:6] which are part of page/frame offset determine all the available sets for each individual half.

In order to guarantee the maximum block coverage (meaning to completely fill the L2 cache array), and thus to fully stress the cache array, the L2 diagnostic micro-virus should not depend on the MMU translations that may result in increased conflict misses. The way to achieve this (meaning maximum block coverage) is by allocating memory that is not only virtually contiguous (as with the standard C memory allocation functions used in user space), but also physically contiguous by using *kmalloc*() function. The kmalloc function's operation is very similar to that of user-space's familiar memory allocation functions, with the main difference that the region of physical memory allocated by *kmalloc*() is physically contiguous. The fact that the memory is physically contiguous guarantees that in one half of the allocated physical pages, the most significant bits of their set index equals to one and in the other half equals to zero.[3]

Given that the replacement policy of the L2 cache is also pseudo-LRU, the L2 diagnostic virus needs to iteratively access five times the allocated data array ($\log_2$(# of ways) = $\log_2 32=5$), to ensure that all the ways of each set contain the correct pattern.

Furthermore, due to the fact that the L1 data cache has write-through write policy and the L2 cache has write allocate write miss policy, the stored data will reside in the L2 cache right after the initial writes (the are no write backs). We also perform a check at the beginning of the test to guarantee that the allocated array is cache aligned (to be block aligned afterwards).

[3] Our kernel was built with the commonly used page size of 4KB; if the page size was 64KB we could use standard C memory allocation functions in user space instead of kmalloc(), due to that the most significant bit of the set index would be a part of the page offset like the rest of the set index bits.

Another thing that we take into consideration is that the L2 diagnostic micro-virus should access the data only from the L2 cache during the test and not from the L1 data cache, to completely stress the former one. The solution to this problem is a stride accessing of the array with a block stride of one block (meaning 8 words each time). Therefore, in the first iteration it accesses the first word of each block, in the second iteration it accesses the second word of each block, and so on. Thus, it always misses the L1 Data cache. Note that the L1 instruction cache can completely hold all the L2 diagnostic micro-virus' instructions, so the L2 cache holds only the data of our test.

To verify the above, we isolated all the system processes by forcing them to run in different cores from the one that executes the L2 diagnostic micro-virus, by setting the system processes' CPU affinity and interrupts to a different core, and we measured the L1 and L2 accesses and misses after we have already "trained" the pseudo-LRU with the initial accesses.

In this particular diagnostic micro-virus, the performance counters show that the L2 diagnostic micro-virus always misses the L1 Data cache and always hits the L1 Instruction cache, and it hits the L2 cache in the vast majority of accesses. Specifically, the L2 cache has 4096 blocks and the maximum number of block misses we observed was 32 at most for each execution of the test. In such a way, we verify that the L2 micro-virus completely fills and stresses the L2 cache.

The L2 diagnostic micro-virus fills the L2 cache with three different patterns, each of which corresponds to a different virus test. These tests are the all-zeros (Figure III), the all-ones (Figure IV), and the checkerboard (Figure V). To enable the self-checking property into this micro-virus, at the end of the test we check if each fetched word is equal to the expected value (the one stored before the test begins).

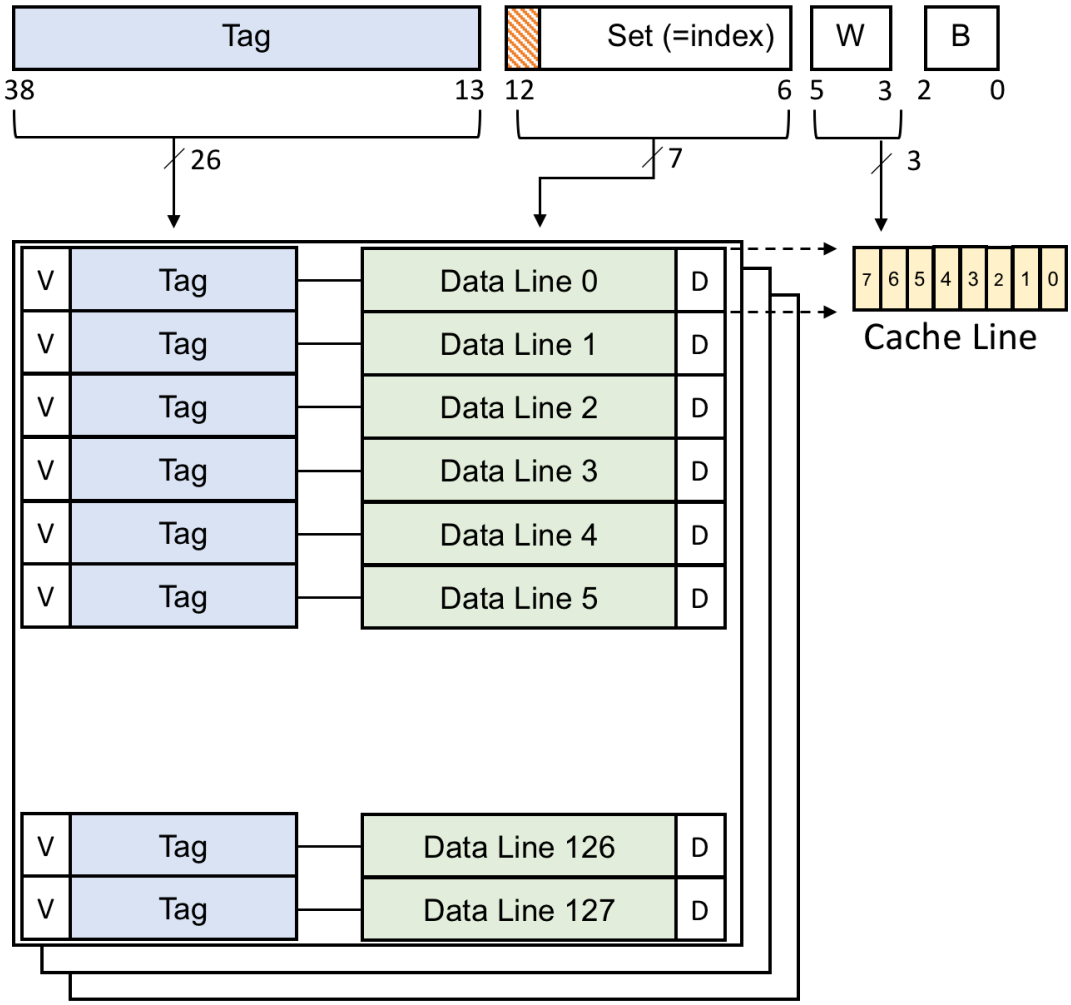


**Figure VI: A 256KB 32-way set associative L2 cache**

Finally, we measured the number of L2 misses, during the testing phase, in a user-space variation of the L2 diagnostic micro-virus (we replaced the call to *kmalloc* with a call to *malloc*) in the X-Gene 2 for 4KB and 64KB physical pages for different executions of the micro-virus. The reason behind this particular experiment is to prove our theory about MMU translations affecting the block coverage in different executions of our program when we employ 4KB physical pages; this is not the case when we employ larger physical pages i.e. 64KB physical pages. As one can easily see, the number of L2 block refills presented in Table IV support our aforementioned theory; there is significantly large variation in the number of L2 cache block refills when employing 4KB physical pages due to the lack of a cache coloring algorithm in our kernel; this is not the case when employing 64KB physical pages. We should also note that we performed more than twenty executions for both page sizes (in order to have a statistically significant sample) and the results are similar to the ones presented in the table below.

**Table IV: L2 Cache Blocks Refills for 4KB and 64KB physical pages of a user-space variation of the L2 cache micro-virus during test phase.**

| Execution | 4KB physical page | 64KB physical page |
|---|---|---|
| 1 | 1400 | 29 |
| 2 | 3789 | 32 |
| 3 | 2983 | 26 |
| 4 | 1143 | 31 |
| 5 | 2092 | 28 |
| 6 | 3252 | 19 |
| 7 | 3476 | 25 |
| 8 | 2255 | 19 |
| 9 | 2289 | 25 |
| 10 | 3435 | 27 |
| 11 | 473 | 30 |
| 12 | 1340 | 29 |
| 13 | 4294 | 27 |
| 14 | 3391 | 32 |
| 15 | 4312 | 29 |
| 16 | 3169 | 19 |
| 17 | 2255 | 22 |
| 18 | 2791 | 27 |
| 19 | 2604 | 25 |
| 20 | 1283 | 29 |

### 3.4 Unified L3 Cache

As in the L2 diagnostic micro-virus, the aim of the L3 diagnostic micro-virus is to stress and test all the bits of the L3 cache (shared among the eight cores). For the development of this test we take into consideration the microarchitectural characteristics for the L3 cache of the X-Gene 2. The L3 cache is a 32-way associative PIPT cache with 4096 sets and is organized in 32 banks; so, each bank has 128 sets and 32 ways. Moreover, the bits of the physical address that determine the block placement in the L3 cache are the bits [12:6] (for choosing the set in a particular bank) and the bits [19:15]

for choosing the correct bank. Based on the above, in order to fill the L3 cache we allocate physically contiguous memory with *kmalloc*() (as we described in subsection 3.3).

However, *kmalloc*() has an upper limit of 128 KB in older kernels and 4MB in newer kernels (like the one we are using). This upper limit is a function of the page size and the number of buddy system freelists (MAX_ORDER). The workaround to this problem is to allocate two arrays with two calls to *kmalloc*() and each array's size should be half the size of the L3 cache. The reason that this workaround will result in full block coverage in the L3 cache is that 4MB chunks of physically contiguous memory gives us contiguously the 22 less significant bits, while we need contiguously only the 20 less significant (for the set index and the bank index).

Moreover, we should highlight that the L3 cache behaves as a non-inclusive victim cache. In response to an L2 cache miss from one of the PMDs, agents forward data directly to the L2 cache of the requestor, bypassing the L3 cache. Afterwards, if the corresponding fill replaces a line in the L2 cache, a write-back request is issued, and the evicted line is allocated into the L3 cache. On a request that hits the L3 cache, the L3 cache forwards the data and invalidates its copy, freeing up space for future evictions. Since data may be forwarded directly from any L2 cache, without passing through the L3 cache, the behavior of the L3 cache increases the effective caching capacity in the system.

Based on the above, and the fact that, like in the L2, the replacement policy is pseudo-LRU it designed accordingly the write and read operations (five sequential writes to "fix" the ways, and the read operations are performed by stride of 1 block). The L3 diagnostic micro-virus fills the L3 cache with three different patterns, each of which corresponds to a different virus test. These tests are the all-zeros (Figure III), the all-ones (Figure IV), and the checkerboard (Figure V). To enable the self-checking property into this micro-virus, at the end of the test we check if each fetched word is equal to the expected value (the one stored before the test begins).

However, in contrast to the L2 diagnostic micro-virus we do not have the necessary tools to prove our thesis (full coverage of the L3 cache) due to the fact that there are no built-in performance counters that correspond to the L3 accesses and misses; with the events that correspond to the L1 and L2 accesses, misses and write backs what we can prove is that all the requests are refills in the L1 and L2 cache. Finally, we should highlight that the shared nature of the L3 cache encouraged us to try to minimize the number of the running daemons in the system in order to reduce the noise in the L3 cache from their access to it.

### 3.5 ALU

X-Gene 2 features a 4-issue out-of-order superscalar microarchitecture. It has one integer scheduler and two different integer pipelines, a Simple Integer Pipeline and a Simple and Complex Integer pipeline. The integer scheduler can issue two integer operations per cycle; each of the other schedulers can issue one operation per cycle (the integer scheduler can issue 2 simple integer operations per cycles; for instance, 2 additions, or 1 simple and 1 complex integer operation; for instance, 1 multiplication and 1 addition).

The execution units are fully pipelined for all operations, including multiplications and multiply-add instructions. Simple ALU operations are single-cycle. The fetch stage can fetch up to 4 instructions from the 64-byte fetch buffer (that has the same size as a cache line) per cycle. The fetch buffer contains 16 instructions from the same cache line

and its contents are cache-block aligned. These 16 instructions (or cache block) are read from the instruction cache before being available for processing by the pipeline.

After taking into account the aforementioned, we developed a self-testing micro-virus, which avoids data and control hazards and iterates 1000 times over a block of 16 aligned instructions (that resides in the fetch buffer, and thus the L1 instruction and data cache are not involved in the stress testing process) and after completing 1000 iterations, it checks the value of the registers involved in the calculations (by comparing them with the expected values). After re-initializing the values of the registers, we repeat the same test 35M times, which is approximately 30 seconds of total execution. Therefore, we execute code that resides in the instruction buffer for 1000 iterations of our loop and then we execute code that resides in 1 block of the cache after the end of these 1000 iterations.

The cache block that is placed in the fetch buffer is consisted of 15 arithmetic and logical instructions (multiplication, addition, subtraction, bitwise logical and, bitwise exclusive or, bitwise inclusive or, bitwise exclusive or not and logical shift left) and one conditional branch in the end of it for controlling the number of the iterations.

Because the instructions are issued and categorized in groups of 4 (X-Gene 2 issues 4 instructions), and the integer scheduler can issue 2 of them per cycle we can't achieve the theoretical optimal IPC of 4 instructions per cycle only with integer operations. Furthermore, we try to have in each group of 4 instructions, instructions that "stress" all the units of all the issue queues like the adder, the shifter and the multiplier. Specifically, the IPC of this test is 1.95, and it consists of 94% integer operations and 6% branches. The fact the IPC is close to 2 (which is the theoretical optimal for this particular micro-virus because we only stress the ALU that is able to issue 2 instructions per cycles) suggests that the chosen combination of instructions and registers avoids data hazards and keeps the involved computation units stressed as much as possible throughout the execution of the micro-virus (one needs to take into consideration that complex integer instructions like multiply may need more operations in order to calculate their result in comparison to simple integer instructions like addition).

Finally, we ensure that the first instruction of the execution block is cache aligned, so we ensure that a cache block is located to the instruction buffer each time.

### 3.6 FPU

To completely stress and diagnose the Floating-Point Unit (FPU), we perform a mix of diverse floating-point operations, by avoiding data hazards (as much as possible) among the instructions and using different inputs to test as many bits and combinations as possible. To implement the self-checking property of the micro-virus, we execute the floating-point operations twice, with the same input registers and different result registers. If the destination registers of these two similar operations have different result, our self-test notifies that an error occurred during the computations. For every iteration, the values of the registers (for all of the FPU operations) are increased by a non-fixed stride that is based on the calculations that take place. The values in the registers of each loop are distinct between them and between every loop.

Moreover, due to the restriction imposed by the size of the instruction buffer, the choice of the involved floating point instructions is even more important. For this reason, we employ instructions like floating multiply add, floating division and floating square root that traditionally involve complex computations and units for the calculation of their result. Based more on intuition rather than solid proofs, errors in such floating-point computations are harder to be cascaded due to the nature of the involved instructions (the same error has to occur twice in the same iteration in two individual instructions in

order to be cascaded). One should also note that if the instruction buffer was able to hold even more cache lines, we would be able to perform the same floating operations thrice in order to decrease even further the chances of encountering cascading errors (that we would otherwise miss).

Finally, we ensure that the first instruction of the execution block is cache aligned, so we ensure that a cache block is located to the instruction buffer each time.

### 3.7 Pipeline

Apart from the dedicated micro-viruses that stress only the ALU and the FPU of the X-Gene 2, we have also constructed a diagnostic micro-virus to stress simultaneously all the issue queues of the pipeline.

The idea is the following: between two consecutive “heavy” (high activity) floating-point instructions of the FPU test (like the consecutive *multiply add*, or the *fsqrt* which follows the *fdiv*) we add a small iteration over 24 array elements of an integer array and a floating-point array. To this end, during these iterations, the “costly” instructions such as *multiply add* have more than enough cycles to calculate their result, while at the same time we perform load, store, integer multiplication, exclusive or, subtractions and branches. All instructions and data of this micro-virus are located in L1 cache in order to fetch them at the same cycle to avoid high cache access latency. As a result, the “pipeline” micro-virus has a great variety of instructions which stress in parallel all integer and floating-point units.

The pipeline micro-virus has a low IPC equal to 0.86. This virus consists of 65% integer operations and 23.1% floating point operations, while the rest are loads, stores and branches. All in all, this particular micro-virus has the greatest diversity in terms of executed instructions among all the ones we developed.

# 4. EXPERIMENTAL EVALUATION

For the evaluation of the micro-viruses, we used three different chips: TTT, TFF, and TSS from the AppliedMicro's (APM) X-Gene 2 micro-server family. The TTT part is the nominal graded part. The TFF is the fast corner part, which has high leakage but at the same time can operate at higher frequency. The TSS part is also corner part which has low leakage and works at lower frequency. All chips can operate with maximum frequency at 2.4GHz. Using the I2C controller we decrease the voltage of the domains of the PMDs and the SoC with a step of 5mV, until the highest voltage point (safe $V_{min}$) before the occurrence of any error (corrected and uncorrected – reported by the hardware ECC mechanisms), SDC (Silent Data Corruption – output mismatch) or Crash. We repeat the experiments 10 times and we select the highest $V_{min}$ of this set of experiments.

We experimentally obtained also the safe $V_{min}$ values of the 10 SPEC CPU2006 benchmarks (bwaves, dealII, leslie3d, milc, soplex, cactusADM, gromacs, mcf, namd, zeusmp) on the three X-Gene 2 chips (TTT, TFF, TSS), running the entire time-consuming undervolting experiment 10 times for each benchmark. These experiments were performed during a period of 6 months on a single X-Gene 2 machine. We also ran our diagnostic micro-viruses, with the same setup for the 3 different chips, as for the SPEC CPU2006 benchmarks. This part of our study focuses on:

1. the quantitative analysis of the safe $V_{min}$ for three significantly different chips of the same architecture in order to expose the potential guardbands of each chip,
2. the measurement of the core-to-core and chip-to-chip variability, and
3. the demonstration of the voltage value provided by our diagnostic micro-viruses that stress the individual components, and finally reveal virtually the same voltage guardbands compared to benchmarks.

The voltage guardband for each program (benchmark or diagnostic micro-virus) is defined as the safe operation zone between the nominal voltage of the microprocessor and its safe $V_{min}$ (where no errors or other abnormal behavior occur).

## 4.1 Benchmarks vs Diagnostic Micro-Viruses

As we discussed earlier, to expose these voltage margins and variations among cores in the same chip and among the three different chips by using the 10 SPEC CPU2006 benchmarks, we spent 6 months of characterization. On the contrary, the same experimentation by using the diagnostic micro-viruses took only 4 days to expose the corresponding safe $V_{min}$ for each core. In Figure VII, Figure VIII, and Figure IX we present that the benchmarks and diagnostic micro-viruses have similar safe $V_{min}$ using the three chips of our study (the difference between them is at most 2%).

We also observed divergences of the $V_{min}$ values as shown in Figure VII, Figure VIII, and Figure IX. For a significant number of programs (benchmarks and diagnostic viruses), we can see variations among different cores and different chips. Figure VII, Figure VIII, and Figure IX represent the maximum safe $V_{min}$ for each core and chip among all the benchmarks (blue line) and all diagnostic micro-viruses (orange line). Considering that the nominal voltage in PMD voltage domain (where these experiments are executed) for the X-Gene 2 is 980mV, we can observe that the $V_{min}$ values of the diagnostic micro-viruses are very close to the corresponding safe $V_{min}$ provided by benchmarks.

We also notice in Figure VII, Figure VIII, and Figure IX that the core-to-core and chip-to-chip variation remains the same across the SPEC benchmarks and the proposed diagnostic micro-viruses; however, there is a relative variation among the three chips. Both SPEC CPU2006 benchmarks and diagnostic micro-viruses provide the same observations for core-to-core and chip-to-chip variation. For instance, in TTT and TFF chip, cores 4 and 5 are the most robust cores, and in TSS chip, core 3 is the most sensitive. This property holds in all programs but can be revealed by the micro-viruses in several orders of magnitude shorter characterization time.

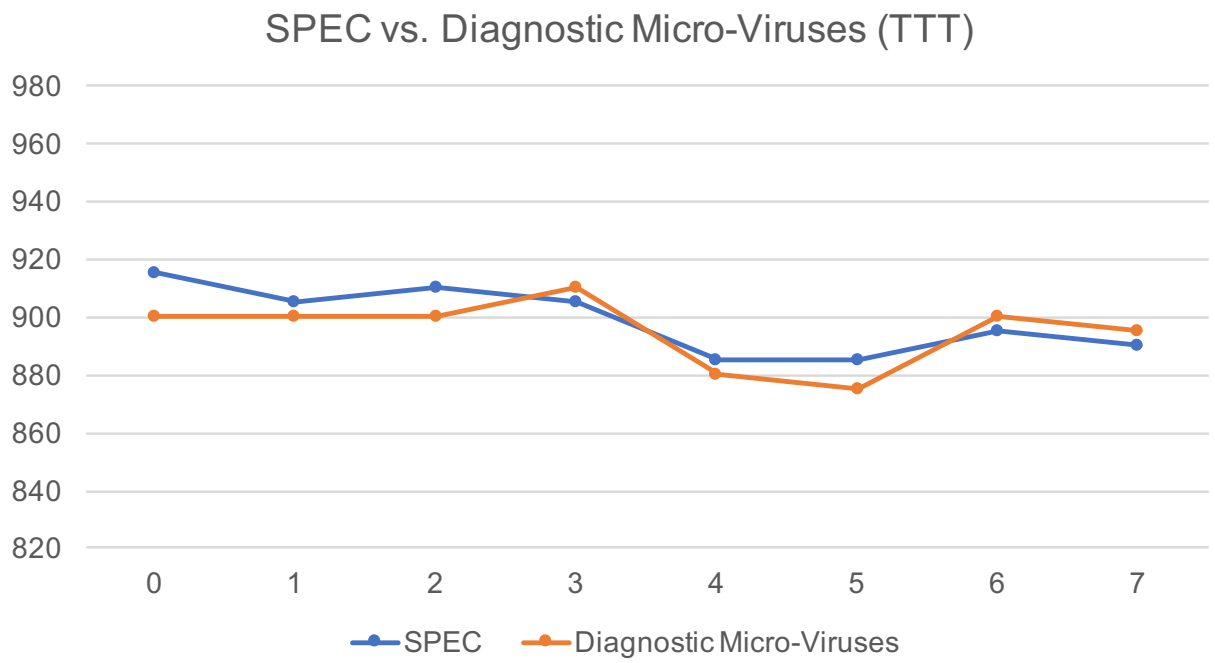


**Figure VII: Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed Diagnostic Micro-Viruses for TTT (nominal) chip**

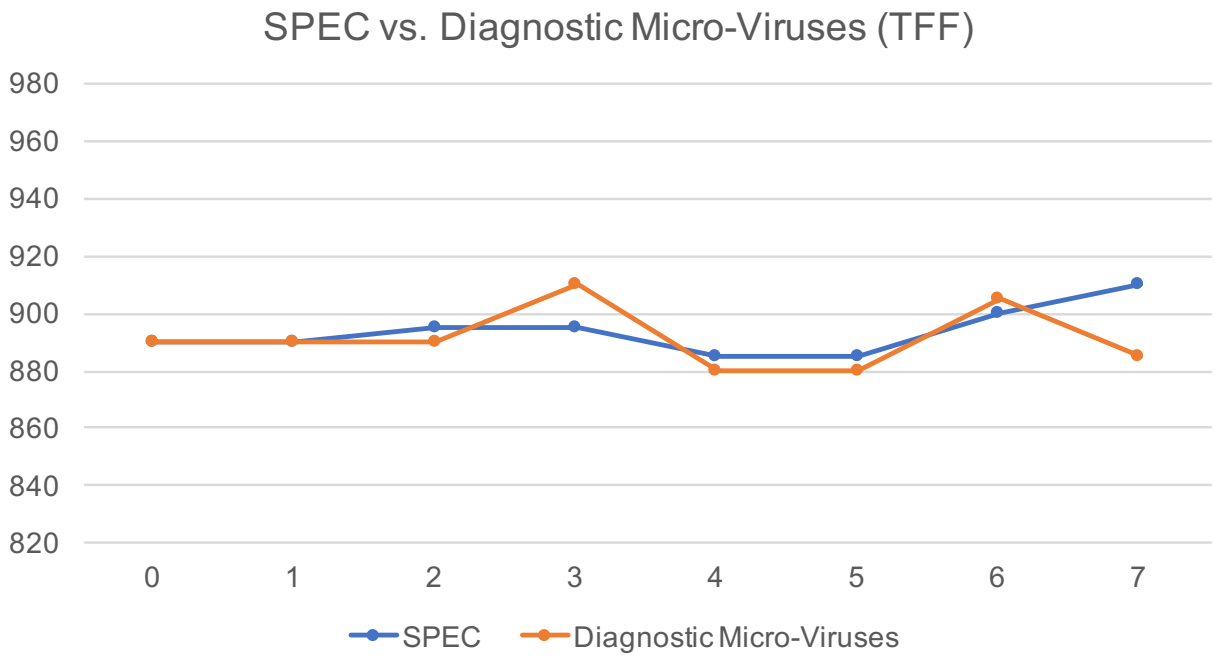


**Figure VIII: Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed Diagnostic Micro-Viruses for TFF (corner part) chip**

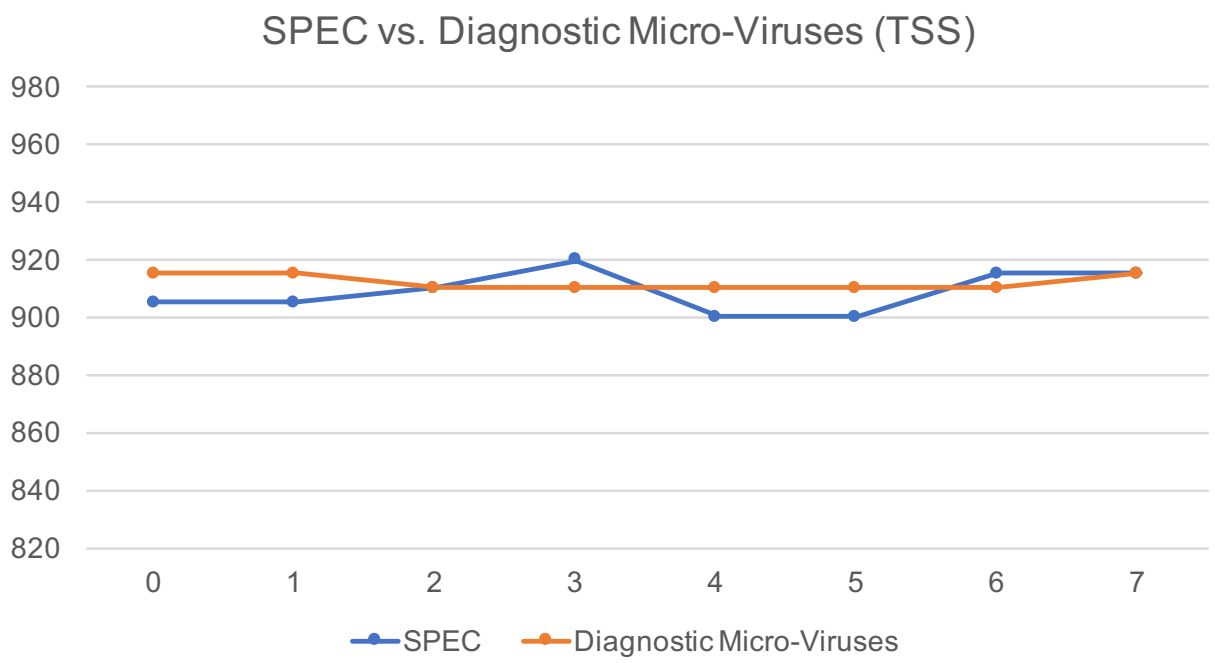


**Figure IX: Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed Diagnostic Micro-Viruses for TSS (corner part) chip**

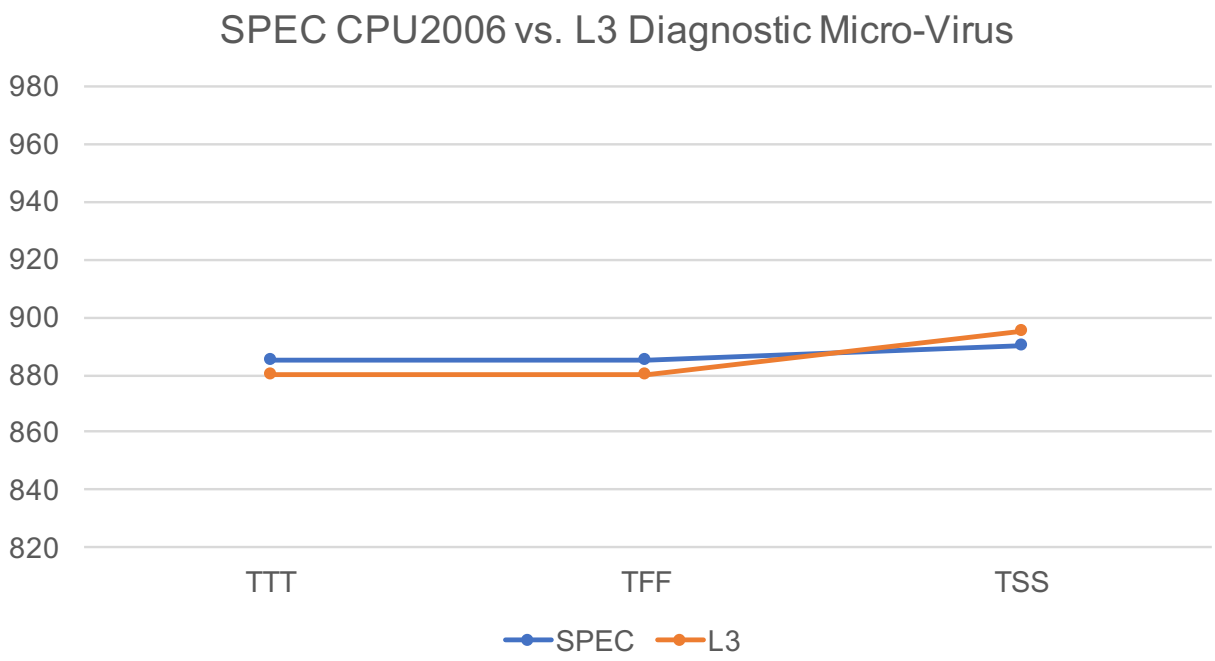


**Figure X: Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed L3 Diagnostic Micro-Viruses for all chips. Undervolting experiments occurred in SoC domain where the L3 cache takes place**

In Figure X we present the undervolting campaign in the SoC voltage domain (which is the focus of the L3 cache micro-virus). As we described in Section 2, in X-Gene 2 there are 2 different voltage domains: the PMD and the SoC domain. The SoC voltage

domain includes the L3 cache. Therefore, Figure X presents the comparison of the L3 diagnostic micro-virus with the 10 SPEC CPU 2006 benchmarks that were executed by reducing the voltage only in the SoC voltage domain. Note that the nominal voltage for the SoC domain is 950mV. In Figure X also, we can notice that there are again very small differences between SPEC and L3 micro-virus reported $V_{min}$.

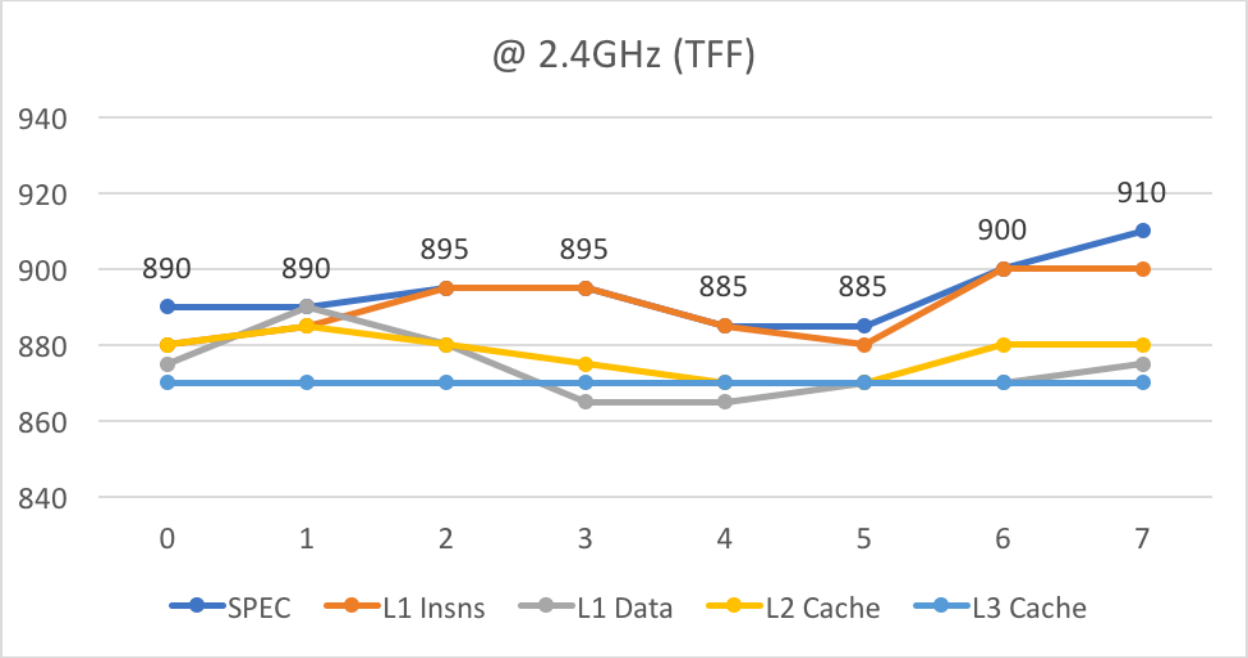


**Figure XI: Detailed Maximum Vmin among 10 SPEC CPU2006 benchmarks and the proposed Diagnostic Cache Micro-Viruses for TFF (corner part) chip**

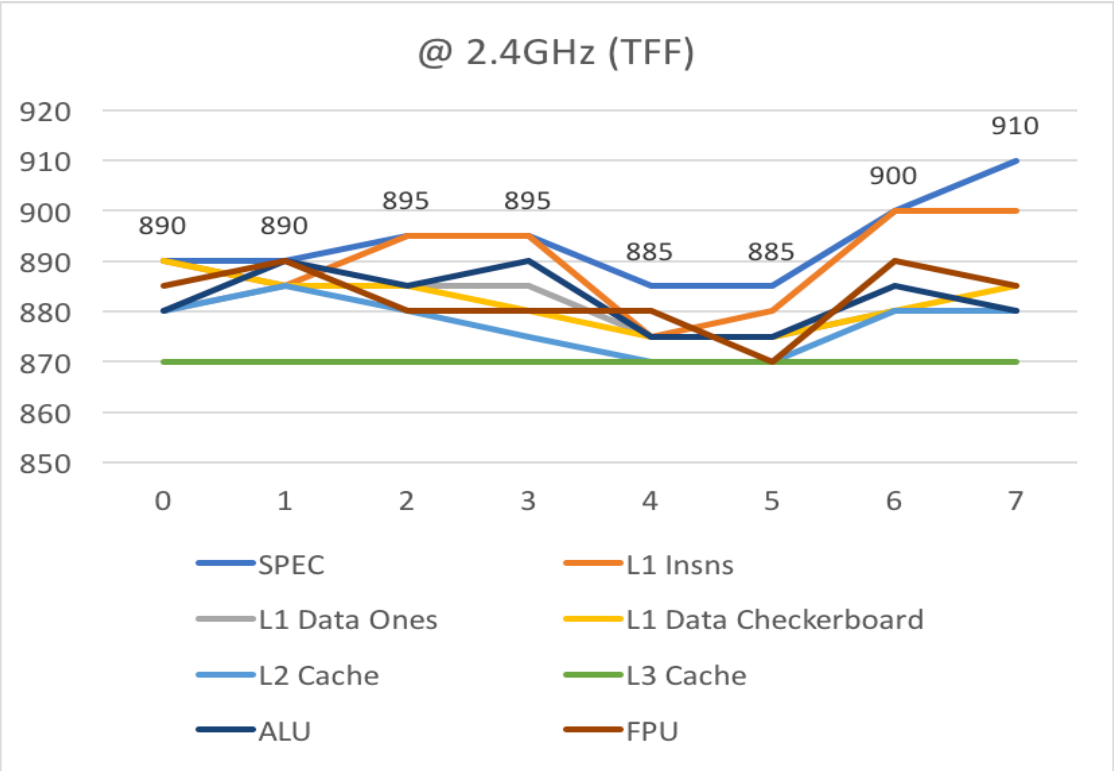


**Figure XII: Detailed Maximum Vmin among 10 SPEC CPU2006 benchmarks and all the proposed Diagnostic Micro-Viruses for TFF (corner part) chip**

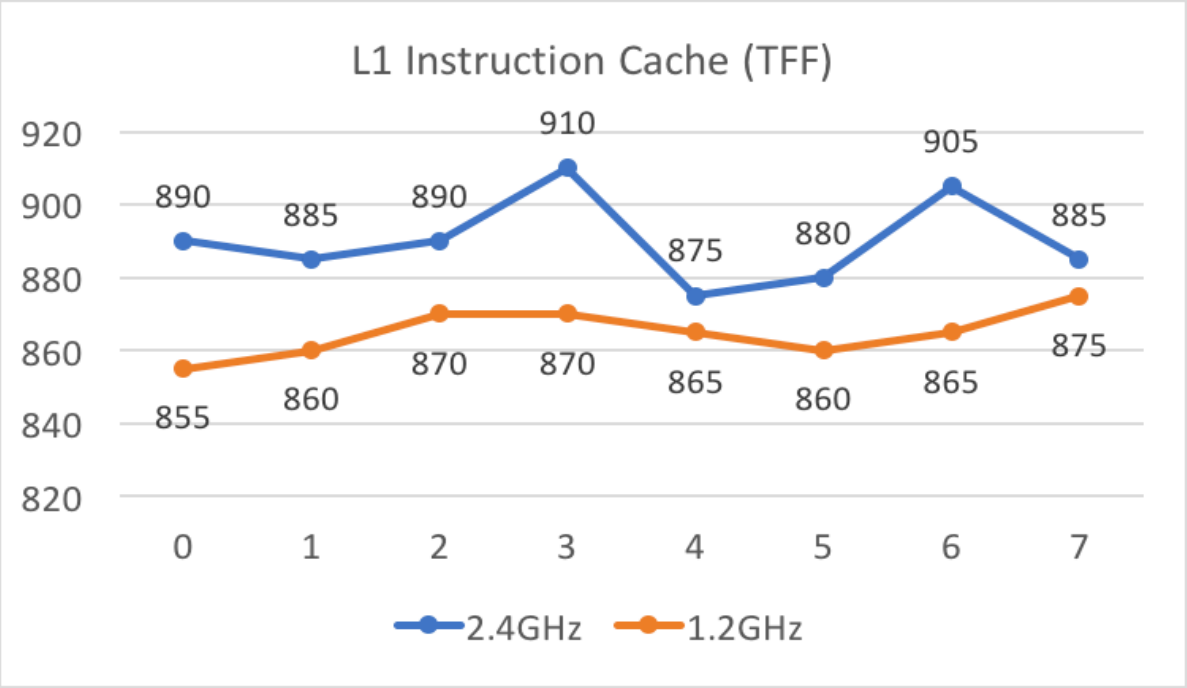


**Figure XIII: Maximum Vmin for the proposed L1 Instruction Cache Micro-Virus for TFF (corner part) chip and two different CPU frequencies**

Figure XI represents the maximum safe $V_{min}$ for each core for the TFF (corner part) chip among all the benchmarks (deep blue line) and all the diagnostic cache micro-viruses (rest of the lines). The most interesting observation is that the highest $V_{min}$ for each core among the diagnostic cache micro-viruses is obtained by the L1 instruction cache micro-virus (apart from core 1 in which case it is obtained by the L1 data cache micro-virus). Moreover, the L3 cache micro-virus has among all cores the lowest $V_{min}$ which can be attributed to the really low IPC of the micro-virus (during the testing phase all the accesses miss the L1 and L2 data cache and hit the L3 cache which has significantly higher access time). Finally, one can attribute the higher $V_{min}$ of the L1 instruction cache micro-virus to the fact this particular micro-virus performs a lot of ALU operations apart from stressing this particular cache.

Figure XII represents the maximum safe $V_{min}$ for each core for the TFF (corner part) chip among all the benchmarks (deep blue line) and all the diagnostic micro-viruses (including the ALU and FPU micro-viruses). The most interesting observation is that the ALU micro-virus has constantly higher $V_{min}$ across all the cores in comparison to the L2 and L3 micro-viruses (as we have already mentioned before this can be explained by the lower IPC of the micro-viruses for lower level of memories, while at the same time the ALU micro-virus has an IPC that is close to two). Furthermore, a really instresting observation is the high deviance of the FPU micro-virus across diferent cores (in comparison to the other micro-viruses).

Figure XIII represents the maximum safe $V_{min}$ for each core for the TFF (corner part) chip for the L1 instruction cache micro-virus for two different CPU frequencies (2.4GHz and 1.2GHz). Based on the blue line (that corresponds to 2.4GHz frequency), cores 4 and 5 of PMD 2 are the most robust ones, as we have already mentioned before, (they have the lower maximum safe $V_{min}$) and core 3 of PMD 1 is the most sensitive one (it has the highest maximum safe $V_{min}$).

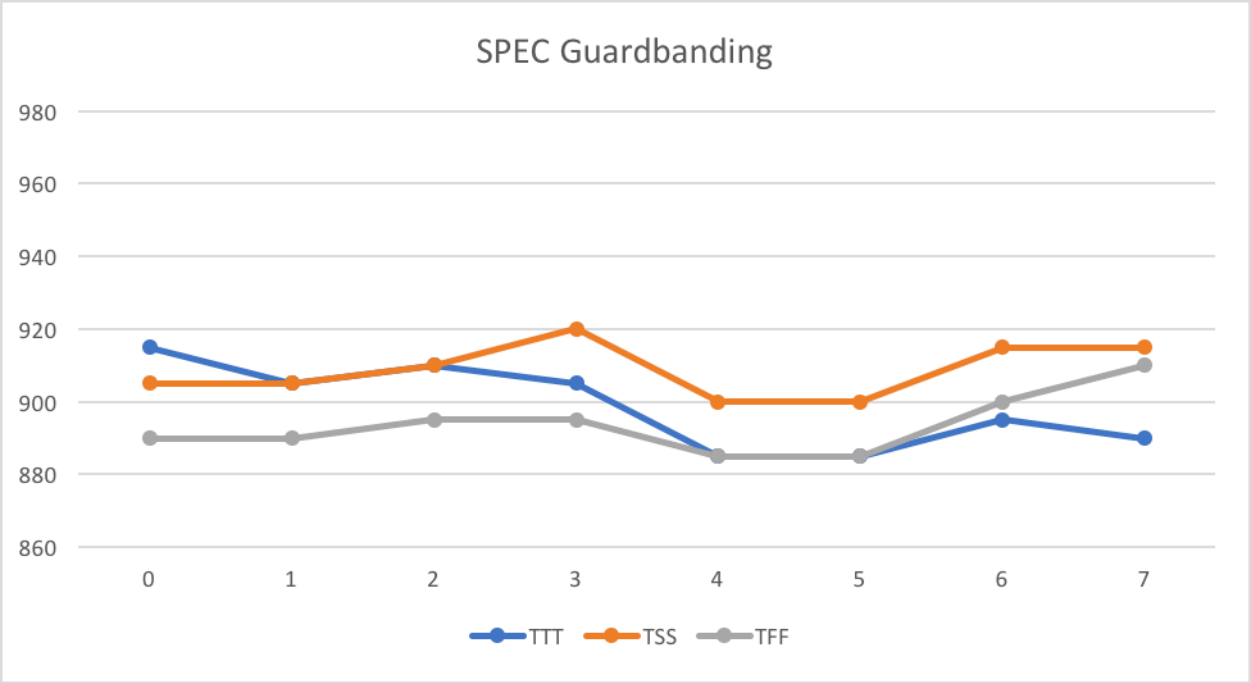


**Figure XIV: Guardband for all chips and all cores based on SPEC CPU 2006 benchmarks**

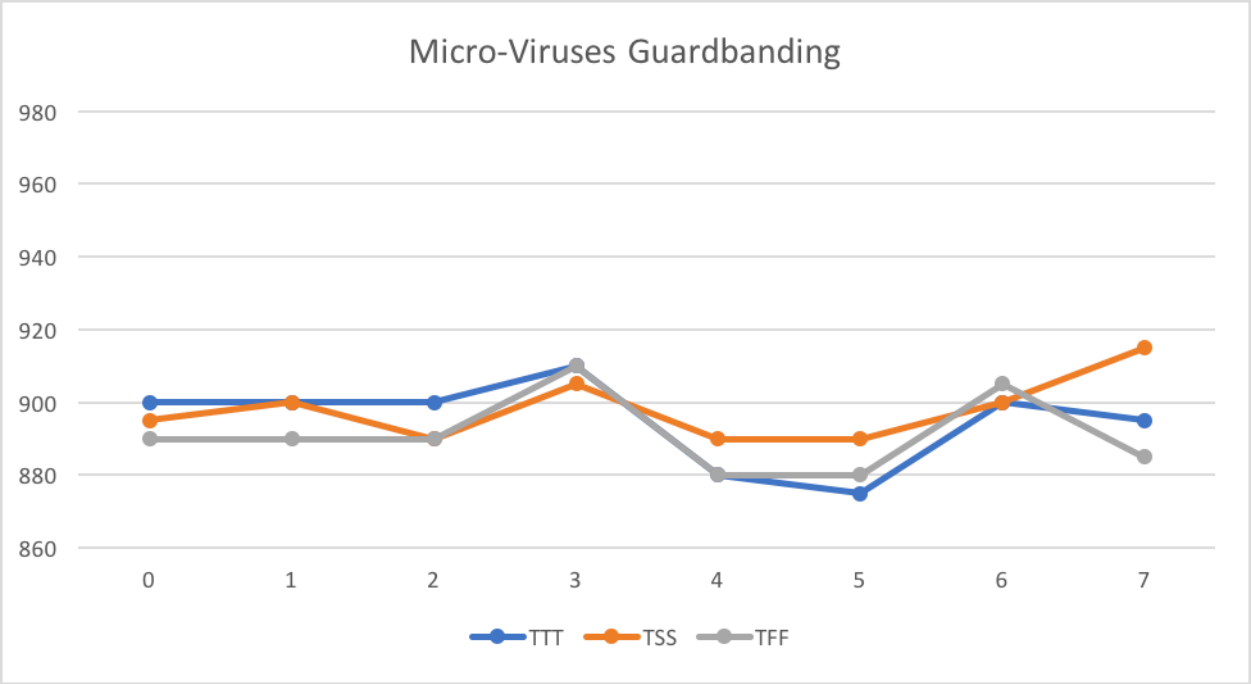


**Figure XV: Guardband for all chips and all cores based on the proposed micro-viruses**

Figure XIV and Figure XV represent the voltage guardbands for all chips and all cores based on the SPEC benchmarks and the proposed micro-viruses respectively. As we have already mentioned, the voltage guardband is defined as the safest voltage margin between the nominal voltage of the microprocessor and its safe $V_{min}$ (where no errors or other abnormal behavior occur).

## 4.2 Observations

We present a detailed study of the safe $V_{min}$ for all benchmarks and cores of the three chips in comparison to the safe $V_{min}$ of the diagnostic micro-viruses. By using the

proposed diagnostic micro-viruses, we can detect accurately (divergences have short range, at most 2%) the safe voltage margins for each chip and core, instead of running time-consuming benchmarks. For the specific ARMv8 design, we find and discuss the core-to-core and chip-to-chip variation, which are important to reduce the power consumption of the microprocessor.

**Core-to-Core Variation**: There are significant divergences among the cores due to process variation. Process variations can affect transistor dimensions (length, width, oxide thickness etc.) which have direct impact on the threshold voltage of a MOS device, and thus, on the guardband of each core. We demonstrate that with the proposed diagnostic micro-viruses we can reveal virtually the same guardbands and expose the variations among the cores of the same chip as by using time-consuming benchmarks.

**Chip-to-Chip Variation**: As Figure VII, Figure VIII, and Figure IX present, PMD 2 (cores 4 and 5) is the most robust PMD for all three chips (up to 3.6% compared to the most sensitive cores). We can notice that (on average among all cores of the same chip) the TFF chip has lower $V_{min}$ points than the TTT chip, in contrast to TSS (the chip with lower leakage), which has higher $V_{min}$ points than the other two chips, and thus, can deliver smaller power savings.

**Error Reporting**: By using the diagnostic micro-viruses we can also determine if and where an error or a silent data corruption (SDC) occurred. Through this component-focused stress process we have observed the following:

1. SDCs occur when the pipeline gets stressed (ALU and FPU tests), and
2. the cache bit-cells operate safely at higher voltages (the caches tests crash in lower voltages than the ALU and FPU tests).

Both observations lead us to conclude that there are more timing errors in X-Gene 2, which occur in higher voltages that any bit-cell error in the cache arrays. Note that in our experimental analysis we present the safe $V_{min}$, which is the safe voltage level where a program operates all times without any abnormal behavior (error detected or corrected by ECC or SDCs).

# 5. MICRO-VIRUS DEVELOPMENT & VALIDATION ON RASPBERRY PI

In this section, we present an additional aspect of the initial development of the proposed L1 data cache micro-virus, the L1 instruction cache micro-virus and the L2 cache micro-virus for the Raspberry Pi 3 Model B.

The Raspberry Pi 3 Model B consists of four 64-bit ARMv8-compliant cores. The basic microarchitectural details of the pipeline of each core are summarized in Table V.

**Table V: Basic Characteristics of Raspberry Pi 3 Model B.**

| Parameter | Configuration |
|---|---|
| **CPU** | 4 Cores, 1.2GHz |
| **ISA** | ARMv8 (AArch64, AArch32) |
| **Pipeline** | 64-bit In-Order |
| **Issue Queue** | 2-way superscalar |
| **ALU** | 1 single & 1 simple/complex integer arithmetic instructions |
| **FPU** | 1x64-bit |
| **Page Size** | 4KB |

Raspberry Pi 3 Model B is the first 64-bit version of the popular barebones computer, yet despite its processor upgrade (in comparison to the ARMv7 compliant CPU of its predecessor), there is not an official 64-bit OS available for it (the Raspberry Pi foundation does not yet provide a 64-bit version of Raspbian, the official OS for Raspberry Pi). That is because the Raspberry Pi Foundation has instead focused on making its Raspbian OS run on all generations of Pi.

After encountering compatibility problems with OpenSUSE (one of the unofficial 64-bit kernels that were available at the moment) we decided to build a 64-bit kernel for the Raspberry Pi 3 Model B based on the instructions found in an online developer blog [33]. The Raspberry Pi foundation maintains their own fork of the Linux Kernel which is especially tailored for their devices, while upstream gets merged regularly. There are two main methods for building a 64-bit kernel; we can build the kernel locally on a Raspberry Pi, which will take a long time or we can cross-compile the kernel, which is much quicker, but requires more setup.

However, for this particular case we cannot use the local building method as it would require a 64-bit Raspberry Pi, which we obviously do not have yet. So, we were forced to cross-compile it; Ubuntu is the recommended OS for this.

After installing a few build tools and the aarch64 cross-compiler, one can download the Linux Kernel sources and then build the kernel with the following commands:

```
> make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcmrpi3_defconfig
> make -j 4 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
```

The basic features of all the caches of the Raspberry Pi 3 Model B are summarized in Table VI. All the characteristics of the Raspberry Pi 3 Model B microprocessor architecture are very important for the construction of the diagnostic micro-viruses we present in the following section.

**Table VI: Raspberry Pi 3 Model B Caches Characteristics.**

| | L1 Instr | L1 Data | L2 | L3 |
|---|---|---|---|---|
| **Size** | 16 KB | 16 KB | 512 KB | - |
| **# of Ways** | 2 | 4 | 16 | - |
| **Block Size** | 64 B | 64 B | 64 B | - |
| **# of Blocks** | 256 | 256 | 8192 | - |
| **# of Sets** | 128 | 64 | 512 | - |
| **Write Policy** | - | Write-Back | Write-Back | - |
| **Write Miss Policy** | No-write allocate | Write allocate | Write allocate | - |
| **Organization** | VIPT[4] | PIPT | PIPT | - |
| **Prefetcher** | YES | YES | YES | - |
| **Scope** | Per Core | Per Core | Shared | - |
| **Protection** | Parity Protected | ECC Protected | ECC Protected | - |

[4] Virtually Indexed, Physically Tagged

### 5.1 L1 Data Cache

For the first level data cache of each core, we defined statically an array in memory with the same size as the L1 data cache (same approach as the respective X-Gene 2 micro-virus). Based on Table II and Table VI, the size of the L1 data cache of the Raspberry Pi 3 Model B is half the size of the L1 data cache of the X-Gene 2.

Moreover, as the L1 data cache is write allocate, before the first write of the desired pattern in all the words of the structure we did not have to read them first in order to bring all the blocks in the first level of data cache (this was not the case for the X-Gene 2 L1 data cache micro-virus due to the fact that the L1 data cache employed a no-write allocate write miss policy).

If our main goal was to develop portable micro-viruses that are able to run in different ARMv8-compliant CPUs with the minimum amount of changes, then for the L1 data cache micro-virus we should always read all the words of the structure in order to bring all the blocks in the first level of data cache, before performing any write operation.

Finally, due to the pseudo-LRU policy that is also employed in the L1 data cache of the Raspberry Pi 3 Model B, we had to read all the words of the cache two consecutive times (before the test begins), in order to ensure that all the blocks with the desired patterns are allocated in the first level data cache. As we have already mentioned before, we experimentally observed that a safe number of iterations is $\log_2$(# of ways) to guarantee that the L1 data cache is filled only with the data of the diagnostic micro-virus. With these steps, we achieve 100% read hit in the cache during the execution of the L1D micro-virus in undervolting conditions.

### 5.2 L1 Instruction Cache

The concept behind the Raspberry Pi 3 Model B L1 Instruction Cache micro-virus is exactly the same as the one in X-Gene 2, because both of the machines feature ARMv8-compliant CPUs. The only difference in this particular micro-virus lies in the fact that the size of the L1 instruction cache of the Raspberry Pi 3 Model B is the half the size of the L1 instruction cache of the X-Gene 2. As a result, we were able to fill the cache with 4096 instructions in comparison to the 8192 needed in the case of X-Gene (we also had to adapt the signatures that we are using for self-checking).

### 5.3 Unified L2 Cache

Once again, the aim of the L2 diagnostic micro-virus is to stress all the bits of the L2 cache. To achieve this, we take into account the microarchitectural characteristics of Raspberry Pi 3 Model B for the L2 cache, while developing the code. Based on Table VI the L2 cache is a 16-way associative PIPT cache with 512 sets; thus, the bits of the physical address that determine the block placement in the L2 cache are bits [14:6]. Moreover, the page size we rely on is 4KB and consequently the page offset is consisted of the 12 less significant bits of the physical address. According to this, the three most significant bits [14:12] of the set index are not a part of the page offset.

In order to guarantee the maximum block coverage (meaning to completely fill the L2 cache array), and thus to fully stress the cache array, the L2 diagnostic micro-virus should not depend on the MMU translations that may result in increased conflict misses. In the same fashion, as in the X-Gene 2, the way to achieve this is by allocating memory that is not only virtually contiguous (as with the standard C memory allocation

functions used in user space), but also physically contiguous by using *kmalloc*() function.

Furthermore, due to the fact that the L1 data cache has write-back policy and the L2 cache has write allocate on miss policy, all the stored data will not reside in the L2 cache right after the initial writes; we need to perform a read traversal after the writes in our structure in order to force write-backs for all the dirty blocks of the L1 data cache (after the writes the L1 data cache is filled with 256 dirty blocks that have not been written back to the L2 cache).

Finally, we have to take into consideration that the L2 diagnostic micro-virus should access the data only from the L2 cache during the test and not from the L1 data cache, to completely stress the former one. The solution to this problem in the respective X-Gene 2 micro-virus was a stride accessing of the array with a block stride of one block (meaning 8 words each time). Therefore, in the first iteration we were accessing the first word of each block, in the second iteration the second word of each block, and so on. Thus, it always missed the L1 Data cache.

However, this is not the case for the Raspberry Pi 3 Model B L2 micro-virus; according to the Cortex-A53 manual, the L1 data cache implements an automatic prefetcher that monitors cache misses in the core. When a pattern is detected, the automatic prefetcher starts linefills in the background. The prefetcher recognizes a sequence of data cache misses at a fixed stride pattern that lies in four cache lines, plus or minus.

Table VII presents the number of L1 Data Accesses and L1 Data Refills for both systems, for the L2 cache micro-virus with a stride accessing of the array with a block stride of one block during the self-checking phase. As one can see, the X-Gene 2 L2 micro-virus always misses the L1 data cache during the test phase (as we have already mentioned in Section 3.3), however this is not the case for the Raspberry Pi 3 Model B L2 micro-virus (if we port the micro-virus without any changes apart from changing the size of the allocated array).

**Table VII: L1 Data Accesses and Refills for X-Gene 2 and Raspberry Pi 3 Model B L2 micro-virus with a stride accessing of the array with a block stride of one block.**

| | **X-Gene 2** | **Raspberry Pi 3 Model B** |
|---|---|---|
| **L1 Data Accesses** | 32801 | 64985 |
| **L1 Data Refills** | 32776 | 2450 |

In order to deal with this problem, we performed a change in the stride accessing of the array; we increased the block stride in order to bypass the L1 data hardware prefetcher. Instead of accessing the first word of the second block in the second iteration, we access the first word of the eighth block. Then, instead of accessing the first word of the

third block in the third iteration, we access the first word of the sixteenth block, and so on. All in all, we increased the block stride by performing a rounding to the next power of 2 for the prefetcher stride (the L2 cache is 32 times larger than the L1 data cache so theoretically we could have picked even a 32-block stride).

In order to shed more light on the aforementioned striding access of the array, Table VIII presents the blocks accessed during the stride accessing of the array for block stride of one block and block stride of eight blocks (for simplicity reasons we assume that our array is consisted of sixty-four blocks in total and we only show the first sixteen of the sixty-four total iterations that are needed to access the first word of each block of the array).

**Table VIII: Block Accessed for block stride of one block and eight blocks**

| Iteration | Block Accessed<br>Block Stride = 1 block | Block Accessed<br>Block Stride = 8 blocks |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 1 | 8 |
| 3 | 2 | 16 |
| 4 | 3 | 24 |
| 5 | 4 | 32 |
| 6 | 5 | 40 |
| 7 | 6 | 48 |
| 8 | 7 | 56 |
| 9 | 8 | 1 |
| 10 | 9 | 9 |
| 11 | 10 | 17 |
| 12 | 11 | 25 |
| 13 | 12 | 33 |
| 14 | 13 | 41 |
| 15 | 14 | 49 |
| 16 | 16 | 57 |

Bellow we cite a generic skeleton of code that implements the stride accessing of an array. The two parameters for this code is WORDS_PER_BLOCKS (when block size equals sixty-four bytes WORDS_PER_BLOCKS equals eight) and BLOCK_STRIDE (when a prefetcher recognizes a sequence of data cache misses at a fixed stride pattern that lies in four cache lines, plus or minus, BLOCK_STRIDE equals eight).

```
1.        i = 0;
2.  stride:
3.        size = 0;
4.  prev:
5.        if (arr[(size+i)%(size_init+1)+WORDS_PER_BLOCK*((size+i)/(size_init+1))]!= 0x0)return -1;
6.        size += BLOCK_STRIDE * WORDS_PER_BLOCK;
7.        if (size==BLOCK_STRIDE*(size_init+1)) goto read;
8.        goto prev;
9.  read:
10.       i++;
11.       if(i == WORDS_PER_BLOCK) goto end;
12.       goto stride;
```

Table IX presents the number of L1 Data Accesses and L1 Data Refills of the Raspberry Pi 3 Model B L2 cache micro-virus with a stride accessing of the array with a block stride of eight blocks during the test phase. As one can see, the updated version of the micro-virus always misses the L1 data cache during the self-checking phase

**Table IX: L1 Data Accesses and Refills L2 micro-virus of Raspberry Pi 3 Model B with a stride accessing of the array with a block stride of eight blocks.**

| | **Raspberry Pi 3 Model B** |
|---|---|
| **L1 Data Accesses** | 64985 |
| **L1 Data Refills** | 64984 |

# 6. RELATED WORK

The studies that aim on energy efficiency using real hardware chips are very limited [9] [10] [11] [12] [13]. Whatmough et al. [14] [15] used an on-chip voltage monitoring circuit to characterize supply voltage droops in a dual-core ARM Cortex-A57 cluster operating at 1.2 GHz.

There are many recent approaches that focus on energy efficiency by studying the voltage noise limits of the chips. For example, Ketkar et al. in [16], and Kim et al. in [17] [18] propose methods to maximize voltage droops in single core and multicore chips. Furthermore, Gupta et al. in [19] and Reddi et al. in [2] focus on the prediction of critical parts of benchmarks where significant voltage noise effects are very likely to occur. Several studies either in the hardware or in the software level were presented to mitigate the effects of voltage noise [20] [21] [22] [23] [24] or to recover from them after their occurrence [25]. Moreover, many studies propose methods to design dedicated energy efficient cores, like Gopireddy et al. in [26].

Some other studies are mainly concentrated on the caches. For instance, Wilkerson et al. [27], Chishti et al. [28] and Duwe et al. [29] propose several microarchitectural approaches to ensure the correct operation of caches in ultra-low voltage conditions. Finally, Bacha et al. [11] [12] focus on the observation of the errors manifested on caches of a commercial Intel Itanium processor during the execution of benchmarks with off-nominal voltage value. None of all the aforementioned studies is focused on the development of diagnostic micro-viruses for fast characterization of voltage margins and variations in real commercial multicore CPUs, which is the scope of our study.

Finally, several software based techniques were used to validate caches, ALU and FPU units from manufacturing defects [30] [31], but none of these methods was used to boost energy efficiency by identifying the chip's safe voltage margins.

# 7. CONCLUSION AND FUTURE WORK

In this thesis, we proposed fast targeted programs (diagnostic micro-viruses) that aim to stress individually the fundamental hardware components (they are known to determine the limits of voltage reduction – $V_{min}$ values) of a multicore CPU architecture. We described the development of the diagnostic micro-viruses which target the three different cache memory levels and the main processing components, the integer and the floating-point arithmetic units. The combined execution of the micro-viruses takes very short time and thus the CPU cores can be extensively stressed to reveal their voltage limits when they operate below the nominal levels.

We demonstrated the effectiveness of the synthetic micro-viruses based program by comparing the $V_{min}$ values it reports to the corresponding $V_{min}$ values that an excessively long characterization campaign with SPEC CPU2006 benchmarks reports. The micro-viruses based characterization flow requires orders of magnitude shorter time while it delivers very close (in most cases identical): (a) $V_{min}$ values for the different CPU chips. (b) $V_{min}$ values for the different cores within a chip, (c) core-to-core and chip-to-chip voltage margins variability.

We evaluated the proposed diagnostic micro-viruses based characterization flow (and compare it to the SPEC-based flow) on three different chips (a nominal grade and two corner parts) of AppliedMicro's X-Gene 2 micro-server family (8-core, ARMv8-based CPUs manufactured in 28nm); the reported results validate the speed and accuracy of the proposed method. Our characterization revealed large voltage margins that can be translated into significant power savings and also large $V_{min}$ variation among the 8 cores of the chip, among 3 different chips (a nominal rated and two sigma chips).

As for future work, it is essential to study how the proposed diagnostic micro-viruses can contribute to increase the safe voltage margins (by entering to the zone with corrected and/or uncorrected ECC errors reported by the hardware). The reason is that the proposed micro-viruses are developed in such a way (due to their self-checking property) to expose in which memory address the error occurred. They can also report, when an SDC occurred, the correct and the faulty returned value. In such a way, we can further diagnose in finer-granularity the sources of weak paths and components of the microprocessor. Therefore, having these information, software or hardware techniques can be proposed to overcome these errors, and thus, to achieve better power efficiency.

Moreover, employing even more test patterns for the data caches, apart from the aforementioned ones, could possibly lead to the discovery of errors that we weren't able to detect with the ones we used.

Finally, a code generator could be implemented in order to automatically generate the diagnostic micro-viruses for other ARMv8-compliant CPUs according to their caches' characteristics; the generator could possibly try to discover these characteristics in order to generate the diagnostic micro-viruses. Among other things the generator could take decisions such as the following ones:

1) Based on the way size and the page size of the system, the generator could decide if a cache diagnostic micro-virus should be implemented in user space (with a call to *malloc)* or in kernel space (with one or multiple calls to *kmalloc* depending on the cache size, the page size and the number of buddy system freelists).
2) Based on the prefetcher stride and the block size, the generator could determine the correct stride traversal that is required in order to bypass higher level caches during the test phase.

3) Based on the replacement policy the generator could decide the number of traversals required before the self-checking phase (as we have already discussed in Section 3 a Pseudo-LRU replacement policy obliges us to traverse the array more than once in order to ensure that the block accesses hit the desired data cache).

# ABBREVIATIONS - ACRONYMS

| | |
|---|---|
| PIPT | Physically Indexed Physically Tagged |
| VIVT | Virtually Indexed Virtually Tagged |
| VIPT | Virtually Indexed Physically Tagged |
| PIVT | Physically Indexed Virtually Tagged |
| TLB | Translation Lookaside Buffer |
| MMU | Memory Management Unit |
| ECC | Error Correction Code |
| ISA | Instruction Set Architecture |
| ALU | Arithmetic Logic Unit |
| FPU | Floating Point Unit |
| PMD | Processor Module |
| MCU | Memory Control Unit |
| MCB | Memory Controller Bridge |
| | |

# ANNEX I

The code of the developed micro-viruses can be found in this annex. Every source file should be compiled without any optimization flag (by default gcc's optimization level is set to zero).

## L1 Data Cache Zero-One Pattern

```
1.  #include <stdio.h>
2.  #include <stdint.h>
3.  #include <stdlib.h>
4.  #include <fcntl.h>
5.  #include <unistd.h>
6.
7.  #define size_init       4095
8.
9.  register uint32_t  size     asm ("r28");
10. register uint64_t  k        asm ("r27");
11. register uint64_t *arr      asm ("r26");
12. register uint32_t  j        asm ("r25");
13.
14. uint64_t  __attribute__((section (".myArrSection"))) array[size_init+1] __attribute__ ((aligned (64)));
15.
16. int main(void) {
17.
18.     arr = array;
19.     j = 0;
20.
21. begin:
22.     size = size_init;
23. prev:
24.     k = arr[size];
25.     arr[size] = (uint64_t)0x0;
26.     if (!--size) goto read;
27.     goto prev;
28. read:
29.     k = arr[size];
30.     arr[size] = (uint64_t)0x0;
31.
32. prev1:
33.     k = arr[size];
34.     arr[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
35.     if (++size==size_init) goto read1;
36.     goto prev1;
37. read1:
38.     k = arr[size];
39.     arr[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
40.
41. prev2:
42.     k = arr[size];
43.     if(j==0){
44.         arr[size] = (uint64_t)0x0;
45.     }
46.     else{
47.         arr[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
48.     }
49.     if (!--size) goto read2;
50.     goto prev2;
51. read2:
52.     k = arr[size];
53.     if(j==0){
54.         arr[size] = (uint64_t)0x0;
55.     }
56.     else{
57.         arr[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
```

```
58.     }
59.
60.  again:
61.     if(j==0){
62.         if ( arr[size] != (uint64_t)0x0 ) return -1;
63.     }
64.     else{
65.         if ( arr[size] != (uint64_t)0xFFFFFFFFFFFFFFFF ) return -1;
66.     }
67.     if (++size==size_init) goto next;
68.     goto again;
69.  next:
70.     if(j==0){
71.         if ( arr[size] != (uint64_t)0x0 ) return -1;
72.     }
73.     else{
74.         if ( arr[size] != (uint64_t)0xFFFFFFFFFFFFFFFF ) return -1;
75.     }
76.
77.     if(j==0){
78.         j = 1;
79.     }
80.     else{
81.         j = 0;
82.     }
83.
84.     goto begin;
85.
86.     return 0;
87.  }
```

## L1 Data Cache Checkerboard Pattern

```
1.   #include <stdio.h>
2.   #include <stdint.h>
3.   #include <stdlib.h>
4.   #include <fcntl.h>
5.   #include <unistd.h>
6.
7.   #define size_init       4095
8.
9.   register uint32_t  size         asm ("r28");
10.  register uint64_t  k            asm ("r27");
11.  register uint64_t *arr          asm ("r26");
12.  register uint32_t  j            asm ("r25");
13.  register uint64_t  block_size   asm ("r21");
14.
15.  uint64_t  __attribute__((section (".myArrSection"))) array[size_init+1] __attribute__ ((aligned (64)));
16.
17.  int main(void) {
18.
19.      arr = array;
20.      j = 0;
21.      block_size = 8; // block size in words
22.
23.  begin:
24.      size = size_init;
25.  prev:
26.      k = arr[size];
27.      arr[size] = (uint64_t)0x0;
28.      if (!--size) goto read;
29.      goto prev;
30.  read:
31.      k = arr[size];
32.      arr[size] = (uint64_t)0x0;
33.
```

```
34. prev1:
35.     k = arr[size];
36.     arr[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
37.     if (++size==size_init) goto read1;
38.     goto prev1;
39. read1:
40.     k = arr[size];
41.     arr[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
42.
43. prev2:
44.     k = arr[size];
45.     if(j==0){
46.         if((size/block_size)%2==0){
47.             arr[size] = (uint64_t)0x5555555555555555;
48.         }
49.         else{
50.             arr[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
51.         }
52.     }
53.     else{
54.         if((size/block_size)%2==0){
55.             arr[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
56.         }
57.         else{
58.             arr[size] = (uint64_t)0x5555555555555555;
59.         }
60.
61.     }
62.     if (!--size) goto read2;
63.     goto prev2;
64. read2:
65.     k = arr[size];
66.     if(j==0){
67.         if((size/block_size)%2==0){
68.             arr[size] = (uint64_t)0x5555555555555555;
69.         }
70.         else{
71.             arr[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
72.         }
73.     }
74.     else{
75.         if((size/block_size)%2==0){
76.             arr[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
77.         }
78.         else{
79.             arr[size] = (uint64_t)0x5555555555555555;
80.         }
81.
82.     }
83.
84. again:
85.     if(j==0){
86.         if((size/block_size)%2==0){
87.             if ( arr[size] != (uint64_t)0x5555555555555555 ) return -1;
88.         }
89.         else{
90.             if ( arr[size] != (uint64_t)0xaaaaaaaaaaaaaaaa ) return -1;
91.         }
92.     }
93.     else{
94.         if((size/block_size)%2==0){
95.             if ( arr[size] != (uint64_t)0xaaaaaaaaaaaaaaaa ) return -1;
96.         }
97.         else{
98.             if ( arr[size] != (uint64_t)0x5555555555555555 ) return -1;
99.         }
100.    }
101.    if (++size==size_init) goto next;
102.    goto again;
103.next:
```

```
104.    if(j==0){
105.        if((size/block_size)%2==0){
106.            if ( arr[size] != (uint64_t)0x5555555555555555 ) return -1;
107.        }
108.        else{
109.            if ( arr[size] != (uint64_t)0xaaaaaaaaaaaaaaaa ) return -1;
110.        }
111.    }
112.    else{
113.        if((size/block_size)%2==0){
114.            if ( arr[size] != (uint64_t)0xaaaaaaaaaaaaaaaa ) return -1;
115.        }
116.        else{
117.            if ( arr[size] != (uint64_t)0x5555555555555555 ) return -1;
118.        }
119.    }
120.
121.
122.    if(j==0){
123.        j = 1;
124.    }
125.    else{
126.        j = 0;
127.    }
128.
129.    goto begin;
130.
131.    return 0;
132.}
```

## L1 Instruction Cache

```
1.  #include <stdio.h>
2.  #include <stdint.h>
3.
4.  #define INSTRA04()   asm volatile("add x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;")
5.  #define INSTRA07()   asm volatile("add x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;add
    x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;")
6.  #define INSTRA10()   asm volatile("add x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;add
    x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;add x28,x28,0x1;")
7.  #define INSTRA70()   INSTRA10();INSTRA10();INSTRA10();INSTRA10();INSTRA10();INSTRA10();INSTRA10();
8.  #define INSTRA100()
     INSTRA10();INSTRA10();INSTRA10();INSTRA10();INSTRA10();INSTRA10();INSTRA10();INSTRA10();INSTRA10();IN
    STRA10();
9.  #define INSTRA1000()
    INSTRA100();INSTRA100();INSTRA100();INSTRA100();INSTRA100();INSTRA100();INSTRA100();INSTRA100();INST
    RA100();INSTRA100();
10.
11. #define INSTRB05()   asm volatile("sub x3,x3,0xffe;sub x3,x3,0xffe;sub x3,x3,#0xffe;sub x3,x3,0xffe;sub x3,x3,0xffe;")
12. #define INSTRB10()   asm volatile("sub x3,x3,0xffe;sub x3,x3,0xffe;sub x3,x3,#0xffe;sub x3,x3,0xffe;sub x3,x3,0xffe;sub
    x3,x3,0xffe;sub x3,x3,#0xffe;sub x3,x3,0xffe;sub x3,x3,0xffe;sub x3,x3,0xffe;")
13. #define INSTRB70()   INSTRB10();INSTRB10();INSTRB10();INSTRB10();INSTRB10();INSTRB10();INSTRB10();
14. #define INSTRB100()
     INSTRB10();INSTRB10();INSTRB10();INSTRB10();INSTRB10();INSTRB10();INSTRB10();INSTRB10();INSTRB10();IN
    STRB10();
15. #define INSTRB1000()
    INSTRB100();INSTRB100();INSTRB100();INSTRB100();INSTRB100();INSTRB100();INSTRB100();INSTRB100();INST
    RB100();INSTRB100();
16.
17. #define INSTRC04()   asm volatile("madd x28,x28,x27,x27;madd x28,x28,x27,x27;madd x28,x28,x27,x27;madd
    x28,x28,x27,x27;")
18. #define INSTRC06()   asm volatile("madd x28,x28,x27,x27;madd x28,x28,x27,x27;madd x28,x28,x27,x27;madd
    x28,x28,x27,x27;madd x28,x28,x27,x27;madd x28,x28,x27,x27;")
19. #define INSTRC10()   INSTRC04();INSTRC06();
20. #define INSTRC70()   INSTRC10();INSTRC10();INSTRC10();INSTRC10();INSTRC10();INSTRC10();INSTRC10();
```

```
21. #define INSTRC100()
     INSTRC10();INSTRC10();INSTRC10();INSTRC10();INSTRC10();INSTRC10();INSTRC10();INSTRC10();INSTRC10();I
    NSTRC10();
22. #define INSTRC1000()
    INSTRC100();INSTRC100();INSTRC100();INSTRC100();INSTRC100();INSTRC100();INSTRC100();INSTRC100();INST
    RC100();INSTRC100();
23.
24. #define INSTRD04()   asm volatile("add x28,x28,x27,asr 2;add x28,x28,x27,asr 2;add x28,x28,x27,asr 2;add
    x28,x28,x27,asr 2;")
25. #define INSTRD06()   asm volatile("add x28,x28,x27,asr 2;add x28,x28,x27,asr 2;add x28,x28,x27,asr 2;add
    x28,x28,x27,asr 2;add x28,x28,x27,asr 2;add x28,x28,x27,asr 2;")
26. #define INSTRD10()   INSTRD04();INSTRD06();
27. #define INSTRD70()   INSTRD10();INSTRD10();INSTRD10();INSTRD10();INSTRD10();INSTRD10();INSTRD10();
28. #define INSTRD100()
     INSTRD10();INSTRD10();INSTRD10();INSTRD10();INSTRD10();INSTRD10();INSTRD10();INSTRD10();INSTRD10();I
    NSTRD10();
29. #define INSTRD1000()
    INSTRD100();INSTRD100();INSTRD100();INSTRD100();INSTRD100();INSTRD100();INSTRD100();INSTRD100();INST
    RD100();INSTRD100();
30.
31. #define INSTRE04()   asm volatile("add w28,w28,w27,lsr 2;add w28,w28,w27,lsr 2;add w28,w28,w27,lsr 2;add
    w28,w28,w27,lsr 2;")
32. #define INSTRE06()   asm volatile("add w28,w28,w27,lsr 2;add w28,w28,w27,lsr 2;add w28,w28,w27,lsr 2;add
    w28,w28,w27,lsr 2;add w28,w28,w27,lsr 2;add w28,w28,w27,lsr 2;")
33. #define INSTRE10()   INSTRE04();INSTRE06();
34. #define INSTRE70()   INSTRE10();INSTRE10();INSTRE10();INSTRE10();INSTRE10();INSTRE10();INSTRE10();
35. #define INSTRE100()
     INSTRE10();INSTRE10();INSTRE10();INSTRE10();INSTRE10();INSTRE10();INSTRE10();INSTRE10();INSTRE10();IN
    STRE10();
36. #define INSTRE1000()
    INSTRE100();INSTRE100();INSTRE100();INSTRE100();INSTRE100();INSTRE100();INSTRE100();INSTRE100();INST
    RE100();INSTRE100();
37.
38. #define INSTRF04()   asm volatile("bics x28,x28,x27;bics x28,x28,x27;bics x28,x28,x27;bics x28,x28,x27;")
39. #define INSTRF06()   asm volatile("bics x28,x28,x27;bics x28,x28,x27;bics x28,x28,x27;bics x28,x28,x27;bics
    x28,x28,x27;bics x28,x28,x27;")
40. #define INSTRF10()   INSTRF04();INSTRF06();
41. #define INSTRF70()   INSTRF10();INSTRF10();INSTRF10();INSTRF10();INSTRF10();INSTRF10();INSTRF10();
42. #define INSTRF100()
     INSTRF10();INSTRF10();INSTRF10();INSTRF10();INSTRF10();INSTRF10();INSTRF10();INSTRF10();INSTRF10();INS
    TRF10();
43. #define INSTRF1000()
    INSTRF100();INSTRF100();INSTRF100();INSTRF100();INSTRF100();INSTRF100();INSTRF100();INSTRF100();INSTR
    F100();INSTRF100();
44.
45. #define INSTRG04()   asm volatile("nop;nop;nop;nop")
46. #define INSTRG06()   asm volatile("nop;nop;nop;nop;nop;nop")
47. #define INSTRG10()   INSTRG04();INSTRG06();
48. #define INSTRG70()   INSTRG10();INSTRG10();INSTRG10();INSTRG10();INSTRG10();INSTRG10();INSTRG10();
49. #define INSTRG100()
     INSTRG10();INSTRG10();INSTRG10();INSTRG10();INSTRG10();INSTRG10();INSTRG10();INSTRG10();INSTRG10();
    INSTRG10();
50. #define INSTRG1000()
    INSTRG100();INSTRG100();INSTRG100();INSTRG100();INSTRG100();INSTRG100();INSTRG100();INSTRG100();INS
    TRG100();INSTRG100();
51.
52.
53. register uint64_t  i    asm ("r28");
54. register uint64_t  j    asm ("r27");
55. register uint64_t  k    asm ("r26");
56. register uint64_t  l    asm ("r25");
57. register uint64_t  m    asm ("r24");
58. register uint64_t  n    asm ("r22");
59.
60. int main(int argc,char** argv) {
61.
62. begin:
63.     // Main Prologue is 3 Instructions...
64.     n = 0;
65.     j = 0xfff;
```

```
66.     k = 0xfff;
67.     l = 0xfff;
68.     m = 0xfff;
69.     asm volatile("nop;nop;nop;nop;nop;nop;nop;nop");
70.  Loop1:
71.     i = 0;
72.     INSTRA1000();
73.     INSTRA1000();
74.     INSTRA1000();
75.     INSTRA1000();
76.     INSTRA1000();
77.     INSTRA1000();
78.     INSTRA1000();
79.     INSTRA1000();
80.     INSTRA100();
81.     INSTRA70();
82.     INSTRA07();
83.     if(i!=8177) {
84.         printf("Loop1 Silent Data Corruption i = %lld\n",i);
85.         return -1;
86.     }
87.     n++;
88.     if(n<20) goto Loop1;
89.     n = 0;
90.
91.  Loop2:
92.     asm volatile("add x3,xzr,xzr");
93.     INSTRB1000();
94.     INSTRB1000();
95.     INSTRB1000();
96.     INSTRB1000();
97.     INSTRB1000();
98.     INSTRB1000();
99.     INSTRB1000();
100.    INSTRB1000();
101.    INSTRB100();
102.    INSTRB70();
103.    INSTRB05();
104.    asm volatile("add x28,x3,xzr");
105.    if(i!=(-33468450)){
106.        printf("Loop2 Silent Data Corruption i = %d\n",i);
107.        return -1;
108.    }
109.    n++;
110.    if(n<20) goto Loop2;
111.    n = 0;
112.
113.Loop3:
114.    i = 0;
115.    j = 1;
116.    INSTRC1000();
117.    INSTRC1000();
118.    INSTRC1000();
119.    INSTRC1000();
120.    INSTRC1000();
121.    INSTRC1000();
122.    INSTRC1000();
123.    INSTRC1000();
124.    INSTRC100();
125.    INSTRC70();
126.    INSTRC06();
127.    if(i!=8176) {
128.        printf("Loop3 Silent Data Corruption i = %d\n",i);
129.        return -1;
130.    }
131.    n++;
132.    if(n<20) goto Loop3;
133.    n = 0;
134.
135.Loop4:
```

```
136.    i = 0;
137.    j = 4;
138.    INSTRD1000();
139.    INSTRD1000();
140.    INSTRD1000();
141.    INSTRD1000();
142.    INSTRD1000();
143.    INSTRD1000();
144.    INSTRD1000();
145.    INSTRD1000();
146.    INSTRD100();
147.    INSTRD70();
148.    INSTRD06();
149.    if(i!=8176) {
150.        printf("Loop4 Silent Data Corruption i = %d\n",i);
151.        return -1;
152.    }
153.    n++;
154.    if(n<20) goto Loop4;
155.    n = 0;
156.
157.Loop5:
158.    i = 0;
159.    j = 4;
160.    INSTRE1000();
161.    INSTRE1000();
162.    INSTRE1000();
163.    INSTRE1000();
164.    INSTRE1000();
165.    INSTRE1000();
166.    INSTRE1000();
167.    INSTRE1000();
168.    INSTRE100();
169.    INSTRE70();
170.    INSTRE06();
171.    if(i!=8176) {
172.        printf("Loop5 Silent Data Corruption i = %d\n",i);
173.        return -1;
174.    }
175.    n++;
176.    if(n<20) goto Loop5;
177.    n = 0;
178.
179.Loop6:
180.    i = 0xabcd;
181.    j = 0xffff;
182.    INSTRF1000();
183.    INSTRF1000();
184.    INSTRF1000();
185.    INSTRF1000();
186.    INSTRF1000();
187.    INSTRF1000();
188.    INSTRF1000();
189.    INSTRF1000();
190.    INSTRF100();
191.    INSTRF70();
192.    INSTRF04();
193.    if(i!=0x0) {
194.        printf("Loop6 Silent Data Corruption i = %x\n",i);
195.        return -1;
196.    }
197.    n++;
198.    if(n<20) goto Loop6;
199.    n = 0;
200.
201.Loop7:
202.    INSTRG1000();
203.    INSTRG1000();
204.    INSTRG1000();
205.    INSTRG1000();
```

```
206.    INSTRG1000();
207.    INSTRG1000();
208.    INSTRG1000();
209.    INSTRG1000();
210.    INSTRG100();
211.    INSTRG70();
212.    INSTRG10();
213.    INSTRG04();
214.    n++;
215.    if(n<20) goto Loop7;
216.    n = 0;
217.
218.    goto begin;
219.    return 0;
220.}
```

## L2 Data Cache Zero-One Pattern

```
1.   #include <linux/module.h>
2.   #include <linux/kernel.h>
3.   #include <linux/vmalloc.h>
4.   #include <linux/slab.h>
5.   #include "armpmu_lib.h"
6.
7.   #define size_init   32767
8.
9.   register uint32_t  size asm ("r28");
10.  register uint64_t  k    asm ("r27");
11.  register uint64_t *arr  asm ("r26");
12.  register uint32_t  i    asm ("r25");
13.  register uint32_t  j    asm ("r24");
14.  register uint64_t it    asm ("r20");
15.
16.  int init_module() {
17.
18.      uint64_t *ptr;
19.      arr = kmalloc((size_init+1+64)*sizeof(uint64_t), GFP_KERNEL);
20.      if(arr) {
21.          printk(KERN_INFO "InfTest --> Allocated address:%p\n",arr);
22.      }
23.      else {
24.          printk(KERN_INFO "InfTest --> Allocation failed\n");
25.          return -1;
26.      }
27.
28.      ptr = arr;
29.      while(((int)arr & ~(int)(0x3F))!=(int)arr) {
30.          arr++;
31.      }
32.
33.      // Infinent Loop Start
34.      j = 0;
35.      it = 425000;
36.  inf:
37.      // First Read
38.      size = 0;
39.  prev:
40.      k = arr[size];
41.      if (++size==size_init) goto read;
42.      goto prev;
43.  read:
44.      k = arr[size];
45.
46.      // Second Read
47.      size = 0;
```

```
48.  prev1:
49.      k = arr[size];
50.      if (++size==size_init) goto read1;
51.      goto prev1;
52.  read1:
53.      k = arr[size];
54.
55.      // Third Read
56.      size = 0;
57.  prev2:
58.      k = arr[size];
59.      if (++size==size_init) goto read2;
60.      goto prev2;
61.  read2:
62.      k = arr[size];
63.
64.      // Fourth Read
65.      size = 0;
66.  prev3:
67.      k = arr[size];
68.      if (++size==size_init) goto read3;
69.      goto prev3;
70.  read3:
71.      k = arr[size];
72.
73.      // Fifth Read
74.      size = 0;
75.  prev4:
76.      k = arr[size];
77.      if (++size==size_init) goto read4;
78.      goto prev4;
79.  read4:
80.      k = arr[size];
81.
82.      // Write
83.      size = 0;
84.  prev5:
85.      k = arr[size];
86.      if(j==0){
87.          arr[size] = (uint64_t)0x0;
88.      }
89.      else {
90.          arr[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
91.      }
92.      if (++size==size_init) goto read5;
93.      goto prev5;
94.  read5:
95.      k = arr[size];
96.      if(j==0){
97.          arr[size] = (uint64_t)0x0;
98.      }
99.      else{
100.         arr[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
101.     }
102.
103.     // Stride = 1 Block
104.     i = 0;
105.stride:
106.     size = 0;
107.prev6:
108.     if(j==0){
109.         if ( arr[size+i] != (uint64_t)0x0 ){
110.             printk(KERN_INFO "L2 --> SDC Iteration : %lld Expected : 0x0 Actual: 0x%llx Index: %d \n",(425000-
    it),arr[size+i],(size+i));
111.         }
112.     }
113.     else{
114.         if ( arr[size+i] != (uint64_t)0xFFFFFFFFFFFFFFFF ){
115.             printk(KERN_INFO "L2 --> SDC Iteration : %lld Expected : 0xFFFFFFFFFFFFFFFF Actual: 0x%llx Index:
    %d \n",(425000-it),arr[size+i],(size+i));
```

```
116.        }
117.    }
118.    size += 8; // 8 words per block * 1 block stride
119.    if (size==(size_init+1)) goto read6;
120.    goto prev6;
121.read6:
122.    i++;
123.    if(i == 8) goto end;
124.    goto stride;
125.end:
126.    if(j==0){
127.        j = 1;
128.    }
129.    else {
130.        j = 0;
131.    }
132.
133.    it--;
134.    if(it!=0) goto inf;
135.
136.    if(ptr) {
137.        kfree(ptr);
138.        printk(KERN_INFO "InfTest --> After kfree call\n");
139.    }
140.    else {
141.        printk(KERN_INFO "InfTest --> Nothing to deallocate\n");
142.    }
143.    return 0;
144.}
145.
146.void cleanup_module(void) {
147.
148.    printk(KERN_INFO "InfTest --> Unloading module\n");
149.}
150.
151.MODULE_AUTHOR("Giannos");
152.MODULE_DESCRIPTION("Selftest");
153.MODULE_LICENSE("GPL");
```

## L2 Data Cache Checkerboard Pattern

```
1.    #include <linux/module.h>
2.    #include <linux/kernel.h>
3.    #include <linux/vmalloc.h>
4.    #include <linux/slab.h>
5.    #include "armpmu_lib.h"
6.
7.    #define size_init   32767
8.
9.    register uint32_t  size asm ("r28");
10.   register uint64_t  k    asm ("r27");
11.   register uint64_t *arr  asm ("r26");
12.   register uint32_t  i    asm ("r25");
13.   register uint32_t  j    asm ("r24");
14.   register uint64_t it    asm ("r20");
15.   register uint64_t block_size asm("r19");
16.
17.   int init_module() {
18.
19.       uint64_t *ptr;
```

```
20.     arr = kmalloc((size_init+1+8)*sizeof(uint64_t), GFP_KERNEL);
21.     if(arr) {
22.         printk(KERN_INFO "InfTest --> Allocated address:%p\n",arr);
23.     }
24.     else {
25.         printk(KERN_INFO "InfTest --> Allocation failed\n");
26.         return -1;
27.     }
28.
29.     ptr = arr;
30.     while(((int64_t)arr & ~(int64_t)(0x3F))!=(int64_t)arr) {
31.         arr++;
32.     }
33.
34.     // Infinent Loop Start
35.     j = 0;
36.     it = 176000;
37.     block_size = 8; // block size in words
38. inf:
39.     // First Read
40.     size = 0;
41. prev:
42.     k = arr[size];
43.     if (++size==size_init) goto read;
44.     goto prev;
45. read:
46.     k = arr[size];
47.
48.     // Second Read
49.     size = 0;
50. prev1:
51.     k = arr[size];
52.     if (++size==size_init) goto read1;
53.     goto prev1;
54. read1:
55.     k = arr[size];
56.
57.     // Third Read
58.     size = 0;
59. prev2:
60.     k = arr[size];
61.     if (++size==size_init) goto read2;
62.     goto prev2;
63. read2:
64.     k = arr[size];
65.
66.     // Fourth Read
67.     size = 0;
68. prev3:
69.     k = arr[size];
70.     if (++size==size_init) goto read3;
71.     goto prev3;
72. read3:
73.     k = arr[size];
74.
75.     // Fifth Read
76.     size = 0;
77. prev4:
78.     k = arr[size];
79.     if (++size==size_init) goto read4;
80.     goto prev4;
81. read4:
82.     k = arr[size];
83.
84.     // Write
85.     size = 0;
86. prev5:
87.     k = arr[size];
88.     if(j==0){
89.         if((size/block_size)%2==0){
```

```
90.             arr[size] = (uint64_t)0x5555555555555555;
91.         }
92.         else{
93.             arr[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
94.         }
95.     }
96.     else{
97.         if((size/block_size)%2==0){
98.             arr[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
99.         }
100.        else{
101.            arr[size] = (uint64_t)0x5555555555555555;
102.        }
103.    }
104.    if (++size==size_init) goto read5;
105.    goto prev5;
106.read5:
107.    k = arr[size];
108.    if(j==0){
109.        if((size/block_size)%2==0){
110.            arr[size] = (uint64_t)0x5555555555555555;
111.        }
112.        else{
113.            arr[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
114.        }
115.    }
116.    else{
117.        if((size/block_size)%2==0){
118.            arr[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
119.        }
120.        else{
121.            arr[size] = (uint64_t)0x5555555555555555;
122.        }
123.    }
124.
125.    // Stride = 1 Block
126.    i = 0;
127.stride:
128.    size = 0;
129.prev6:
130.    if(j==0){
131.        if(((size+i)/block_size)%2==0){
132.            if ( arr[size+i] != (uint64_t)0x5555555555555555 ) {
133.                printk(KERN_INFO "L2 --> SDC Iteration : %lld Expected : 0x5555555555555555 Actual: 0x%llx Index:
    %d \n",(176000-it),arr[size+i],(size+i));
134.            }
135.        }
136.        else{
137.            if ( arr[size+i] != (uint64_t)0xaaaaaaaaaaaaaaaa ) {
138.                printk(KERN_INFO "L2 --> SDC Iteration : %lld Expected : 0xaaaaaaaaaaaaaaaa Actual: 0x%llx Index:
    %d \n",(176000-it),arr[size+i],(size+i));
139.            }
140.        }
141.    }
142.    else{
143.        if(((size+i)/block_size)%2==0){
144.            if ( arr[size+i] != (uint64_t)0xaaaaaaaaaaaaaaaa ) {
145.                printk(KERN_INFO "L2 --> SDC Iteration : %lld Expected : 0xaaaaaaaaaaaaaaaa Actual: 0x%llx Index:
    %d \n",(176000-it),arr[size+i],(size+i));
146.            }
147.        }
148.        else{
149.            if ( arr[size+i] != (uint64_t)0x5555555555555555 ) {
150.                printk(KERN_INFO "L2 --> SDC Iteration : %lld Expected : 0x5555555555555555 Actual: 0x%llx Index:
    %d \n",(176000-it),arr[size+i],(size+i));
151.            }
152.        }
153.    }
154.    size += 8; // 8 words per block * 1 block stride
155.    if (size==(size_init+1)) goto read6;
```

```
156.    goto prev6;
157.read6:
158.    i++;
159.    if(i == 8) goto end;
160.    goto stride;
161.end:
162.    if(j==0){
163.        j = 1;
164.    }
165.    else {
166.        j = 0;
167.    }
168.
169.    it--;
170.    if(it!=0) goto inf;
171.
172.    if(ptr) {
173.        kfree(ptr);
174.        printk(KERN_INFO "InfTest --> After kfree call\n");
175.    }
176.    else {
177.        printk(KERN_INFO "InfTest --> Nothing to deallocate\n");
178.    }
179.    return 0;
180.}
181.
182.void cleanup_module(void) {
183.
184.    printk(KERN_INFO "InfTest --> Unloading module\n");
185.}
186.
187.MODULE_AUTHOR("Giannos");
188.MODULE_DESCRIPTION("Selftest");
189.MODULE_LICENSE("GPL");
```

## L3 Data Cache Zero-One Pattern

```
1.    #include <linux/module.h>
2.    #include <linux/kernel.h>
3.    #include <linux/vmalloc.h>
4.    #include <linux/slab.h>
5.    #include <asm/io.h>
6.
7.    #define size_init   1048575
8.    #define sub_size     524287
9.
10.   register uint32_t  size  asm ("r28");
11.   register uint64_t  k     asm ("r27");
12.   register uint32_t  i     asm ("r26");
13.   register uint32_t  j     asm ("r25");
14.   register uint64_t it     asm ("r24");
15.   register uint64_t *arr1  asm ("r21");
16.   register uint64_t *arr2  asm ("r20");
17.
18.   int init_module() {
19.
20.       printk(KERN_INFO "L3 --> Allocating Memory : %d kB\n",(size_init+1)*8/1024);
21.       arr1 = kmalloc((size_init+1)/2*sizeof(uint64_t),GFP_KERNEL);
22.       arr2 = kmalloc((size_init+1)/2*sizeof(uint64_t),GFP_KERNEL);
23.
```

```
24.     if(arr1==NULL){
25.         printk(KERN_INFO "L3 --> Allocation failed\n");
26.         return -1;
27.     }
28.     else{
29.         printk(KERN_INFO "L3 --> Array1 Virtual Address:%p\n",arr1);
30.         printk(KERN_INFO "L3 --> Array1 Physical Address:%llx\n",virt_to_phys(arr1));
31.     }
32.
33.     if(arr2==NULL){
34.         printk(KERN_INFO "L3 --> Allocation failed\n");
35.         return -1;
36.     }
37.     else{
38.         printk(KERN_INFO "L3 --> Array2 Virtual Address:%p\n",arr2);
39.         printk(KERN_INFO "L3 --> Array2 Physical Address:%llx\n",virt_to_phys(arr2));
40.     }
41.
42.     j=0;
43.     it = 9600;
44. begin:
45.     // Write without read
46.     size = 0;
47. prev5:
48.     if(j==0){
49.         arr1[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
50.         arr2[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
51.     }
52.     else{
53.         arr1[size] = (uint64_t)0x0;
54.         arr2[size] = (uint64_t)0x0;
55.     }
56.     if (++size==sub_size) goto read5;
57.     goto prev5;
58. read5:
59.     if(j==0){
60.         arr1[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
61.         arr2[size] = (uint64_t)0xFFFFFFFFFFFFFFFF;
62.     }
63.     else{
64.         arr1[size] = (uint64_t)0x0;
65.         arr2[size] = (uint64_t)0x0;
66.     }
67.
68.     // First Read
69.     size = 0;
70. prev:
71.     k = arr1[size];
72.     k = arr2[size];
73.     if (++size==sub_size) goto read;
74.     goto prev;
75. read:
76.     k = arr1[size];
77.     k = arr2[size];
78.
79.     // Second Read
80.     size = 0;
81. prev1:
82.     k = arr1[size];
83.     k = arr2[size];
84.     if (++size==sub_size) goto read1;
85.     goto prev1;
86. read1:
87.     k = arr1[size];
88.     k = arr2[size];
89.
90.     // Third Read
91.     size = 0;
92. prev2:
93.     k = arr1[size];
```

```
94.     k = arr2[size];
95.     if (++size==sub_size) goto read2;
96.     goto prev2;
97.  read2:
98.     k = arr1[size];
99.     k = arr2[size];
100.
101.    // Fourth Read
102.    size = 0;
103.prev3:
104.    k = arr1[size];
105.    k = arr2[size];
106.    if (++size==sub_size) goto read3;
107.    goto prev3;
108.read3:
109.    k = arr1[size];
110.    k = arr2[size];
111.
112.    // Fifth Read
113.    size = 0;
114.prev4:
115.    k = arr1[size];
116.    k = arr2[size];
117.    if (++size==sub_size) goto read4;
118.    goto prev4;
119.read4:
120.    k = arr1[size];
121.    k = arr2[size];
122.
123.    // Stride = 8 Blocks
124.    i = 0;
125.stride:
126.    size = 0;
127.prev6:
128.    if(j==0){
129.        if ( arr1[size+i] != (uint64_t)0xFFFFFFFFFFFFFFFF ){
130.            printk(KERN_INFO "L3 --> SDC Arr1 Iteration : %lld Expected : 0xFFFFFFFFFFFFFFFF Actual: 0x%llx Index:
    %d \n",(9600-it),arr1[size+i],(size+i));
131.        }
132.
133.        if ( arr2[size+i] != (uint64_t)0xFFFFFFFFFFFFFFFF ){
134.            printk(KERN_INFO "L3 --> SDC Arr2 Iteration : %lld Expected : 0xFFFFFFFFFFFFFFFF Actual: 0x%llx Index:
    %d \n",(9600-it),arr2[size+i],(size+i));
135.        }
136.    }
137.    else{
138.        if ( arr1[size+i] != (uint64_t)0x0 ){
139.            printk(KERN_INFO "L3 --> SDC Arr1 Iteration : %lld Expected : 0x0 Actual: 0x%llx Index: %d \n",(9600-
    it),arr1[size+i],(size+i));
140.        }
141.
142.        if ( arr2[size+i] != (uint64_t)0x0 ){
143.            printk(KERN_INFO "L3 --> SDC Arr2 Iteration : %lld Expected : 0x0 Actual: 0x%llx Index: %d \n",(9600-
    it),arr2[size+i],(size+i));
144.        }
145.    }
146.    size += 8; // 8 words per block * 1 block stride
147.    if (size==(sub_size+1)) goto read6;
148.    goto prev6;
149.read6:
150.    i++;
151.    if(i == 8) goto end;
152.    goto stride;
153.end:
154.
155.    if(j==0){
156.        j = 1;
157.    }
158.    else {
159.        j = 0;
```

```
160.    }
161.
162.    it--;
163.    if(it!=0) goto begin;
164.
165.    if(arr1) {
166.        kfree(arr1);
167.    }
168.    if(arr2) {
169.        kfree(arr2);
170.    }
171.
172.    return 0;
173.}
174.
175.
176.
177.void cleanup_module(void) {
178.
179.    printk(KERN_INFO "L3 --> Unloading module\n");
180.}
181.
182.MODULE_AUTHOR("Giannos");
183.MODULE_DESCRIPTION("Selftest");
184.MODULE_LICENSE("GPL");
```

## L3 Data Cache Checkerboard Pattern

```
1.  #include <linux/kernel.h>
2.  #include <linux/vmalloc.h>
3.  #include <linux/slab.h>
4.  #include <asm/io.h>
5.
6.  #define size_init   1048575
7.  #define sub_size     524287
8.
9.  register uint32_t  size  asm ("r28");
10. register uint64_t  k     asm ("r27");
11. register uint32_t  i     asm ("r26");
12. register uint32_t  j     asm ("r25");
13. register uint64_t it     asm ("r24");
14. register uint64_t *arr1  asm ("r21");
15. register uint64_t *arr2  asm ("r20");
16. register uint64_t block_size asm("r19");
17.
18. int init_module() {
19.
20.     printk(KERN_INFO "L3 --> Allocating Memory : %d kB\n",(size_init+1)*8/1024);
21.     arr1 = kmalloc((size_init+1)/2*sizeof(uint64_t),GFP_KERNEL);
22.     arr2 = kmalloc((size_init+1)/2*sizeof(uint64_t),GFP_KERNEL);
23.
24.     if(arr1==NULL){
25.         printk(KERN_INFO "L3 --> Allocation failed\n");
26.         return -1;
27.     }
28.     else{
29.         printk(KERN_INFO "L3 --> Array1 Virtual Address:%p\n",arr1);
30.         printk(KERN_INFO "L3 --> Array1 Physical Address:%llx\n",virt_to_phys(arr1));
31.     }
32.
33.     if(arr2==NULL){
34.         printk(KERN_INFO "L3 --> Allocation failed\n");
35.         return -1;
36.     }
37.     else{
38.         printk(KERN_INFO "L3 --> Array2 Virtual Address:%p\n",arr2);
39.         printk(KERN_INFO "L3 --> Array2 Physical Address:%llx\n",virt_to_phys(arr2));
40.     }
41.
42.     j=0;
43.     it = 8000;
44.     block_size = 8; // block size in words
45. begin:
46.     // Write without read
47.     size = 0;
48. prev5:
49.     if(j==0){
50.         if((size/block_size)%2==0){
51.             arr1[size] = (uint64_t)0x5555555555555555;
52.             arr2[size] = (uint64_t)0x5555555555555555;
53.         }
54.         else{
55.             arr1[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
56.             arr2[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
57.         }
58.     }
59.     else{
60.         if((size/block_size)%2==0){
61.             arr1[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
62.             arr2[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
63.         }
64.         else{
65.             arr1[size] = (uint64_t)0x5555555555555555;
```

```
66.             arr2[size] = (uint64_t)0x5555555555555555;
67.         }
68.     }
69.     if (++size==sub_size) goto read5;
70.     goto prev5;
71. read5:
72.     if(j==0){
73.         if((size/block_size)%2==0){
74.             arr1[size] = (uint64_t)0x5555555555555555;
75.             arr2[size] = (uint64_t)0x5555555555555555;
76.         }
77.         else{
78.             arr1[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
79.             arr2[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
80.         }
81.     }
82.     else{
83.         if((size/block_size)%2==0){
84.             arr1[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
85.             arr2[size] = (uint64_t)0xaaaaaaaaaaaaaaaa;
86.         }
87.         else{
88.             arr1[size] = (uint64_t)0x5555555555555555;
89.             arr2[size] = (uint64_t)0x5555555555555555;
90.         }
91.     }
92.
93.     // First Read
94.     size = 0;
95. prev:
96.     k = arr1[size];
97.     k = arr2[size];
98.     if (++size==sub_size) goto read;
99.     goto prev;
100.read:
101.    k = arr1[size];
102.    k = arr2[size];
103.
104.    // Second Read
105.    size = 0;
106.prev1:
107.    k = arr1[size];
108.    k = arr2[size];
109.    if (++size==sub_size) goto read1;
110.    goto prev1;
111.read1:
112.    k = arr1[size];
113.    k = arr2[size];
114.
115.    // Third Read
116.    size = 0;
117.prev2:
118.    k = arr1[size];
119.    k = arr2[size];
120.    if (++size==sub_size) goto read2;
121.    goto prev2;
122.read2:
123.    k = arr1[size];
124.    k = arr2[size];
125.
126.    // Fourth Read
127.    size = 0;
128.prev3:
129.    k = arr1[size];
130.    k = arr2[size];
131.    if (++size==sub_size) goto read3;
132.    goto prev3;
133.read3:
134.    k = arr1[size];
135.    k = arr2[size];
```

```
136.
137.    // Fifth Read
138.    size = 0;
139.prev4:
140.    k = arr1[size];
141.    k = arr2[size];
142.    if (++size==sub_size) goto read4;
143.    goto prev4;
144.read4:
145.    k = arr1[size];
146.    k = arr2[size];
147.
148.    // Stride = 8 Blocks
149.    i = 0;
150.stride:
151.    size = 0;
152.prev6:
153.    if(j==0){
154.        if(((size+i)/block_size)%2==0){
155.            if ( arr1[size+i] != (uint64_t)0x5555555555555555 ) {
156.                printk(KERN_INFO "L3 --> SDC Arr1 Iteration : %lld Expected : 0x5555555555555555 Actual: 0x%llx Index:
    %d \n",(7980-it),arr1[size+i],(size+i));
157.            }
158.            if ( arr2[size+i] != (uint64_t)0x5555555555555555 ) {
159.                printk(KERN_INFO "L3 --> SDC Arr2 Iteration : %lld Expected : 0x5555555555555555 Actual: 0x%llx Index:
    %d \n",(7980-it),arr2[size+i],(size+i));
160.            }
161.        }
162.        else{
163.            if ( arr1[size+i] != (uint64_t)0xaaaaaaaaaaaaaaaa ) {
164.                printk(KERN_INFO "L3 --> SDC Arr1 Iteration : %lld Expected : 0xaaaaaaaaaaaaaaaa Actual: 0x%llx Index:
    %d \n",(7980-it),arr1[size+i],(size+i));
165.            }
166.            if ( arr2[size+i] != (uint64_t)0xaaaaaaaaaaaaaaaa ) {
167.                printk(KERN_INFO "L3 --> SDC Arr2 Iteration : %lld Expected : 0xaaaaaaaaaaaaaaaa Actual: 0x%llx Index:
    %d \n",(7980-it),arr2[size+i],(size+i));
168.            }
169.        }
170.    }
171.    else{
172.        if(((size+i)/block_size)%2==0){
173.            if ( arr1[size+i] != (uint64_t)0xaaaaaaaaaaaaaaaa ) {
174.                printk(KERN_INFO "L3 --> SDC Arr1 Iteration : %lld Expected : 0xaaaaaaaaaaaaaaaa Actual: 0x%llx Index:
    %d \n",(7980-it),arr1[size+i],(size+i));
175.            }
176.            if ( arr2[size+i] != (uint64_t)0xaaaaaaaaaaaaaaaa ) {
177.                printk(KERN_INFO "L3 --> SDC Arr1 Iteration : %lld Expected : 0xaaaaaaaaaaaaaaaa Actual: 0x%llx Index:
    %d \n",(7980-it),arr2[size+i],(size+i));
178.            }
179.        }
180.        else{
181.            if ( arr1[size+i] != (uint64_t)0x5555555555555555 ) {
182.                printk(KERN_INFO "L3 --> SDC Arr1 Iteration : %lld Expected : 0x5555555555555555 Actual: 0x%llx Index:
    %d \n",(7980-it),arr1[size+i],(size+i));
183.            }
184.            if ( arr2[size+i] != (uint64_t)0x5555555555555555 ) {
185.                printk(KERN_INFO "L3 --> SDC Arr1 Iteration : %lld Expected : 0x5555555555555555 Actual: 0x%llx Index:
    %d \n",(7980-it),arr2[size+i],(size+i));
186.            }
187.        }
188.    }
189.    size += 8; // 8 words per block * 1 block stride
190.    if (size==(sub_size+1)) goto read6;
191.    goto prev6;
192.read6:
193.    i++;
194.    if(i == 8) goto end;
195.    goto stride;
196.end:
197.
```

```
198.    if(j==0){
199.        j = 1;
200.    }
201.    else {
202.        j = 0;
203.    }
204.
205.    it--;
206.    if(it!=0) goto begin;
207.
208.    if(arr1) {
209.        kfree(arr1);
210.    }
211.    if(arr2) {
212.        kfree(arr2);
213.    }
214.    return 0;
215.}
216.
217.
218.
219.void cleanup_module(void) {
220.
221.    printk(KERN_INFO "L3 --> Unloading module\n");
222.}
223.
224.MODULE_AUTHOR("Giannos");
225.MODULE_DESCRIPTION("Selftest");
226.MODULE_LICENSE("GPL");
```

## ALU

```
1.   #include <stdio.h>
2.   #include <stdint.h>
3.   #include <time.h>
4.
5.   register uint64_t  i    asm ("r28");
6.   register uint64_t  j    asm ("r27");
7.   register uint64_t  k    asm ("r26");
8.   register uint64_t  l    asm ("r25");
9.   register uint64_t  m    asm ("r24");
10.  register uint64_t  n    asm ("r22");
11.  register uint64_t it    asm ("r21");
12.
13.  __attribute__(( aligned(64) )) int main(int argc,char **argv){
14.
15.      time_t timer;
16.      char buffer[26];
17.      struct tm* tm_info;
18.
19.      FILE *fp = fopen("log-alu.txt", "a");
20.      if (fp == NULL) {
21.          printf("Error opening file!\n");
22.          return -1;
23.      }
24.
25.      it = 35200000;
26.
27.  inf:
28.      i = 1000;   //x28
29.      j = 105;    //x27
30.      k = 90;     //x26
31.      l = 40;     //x25
32.      m = 0x5c0482a;  //x24
33.      n = 0x3f2f2e0;  //x22
34.
```

```
35.     asm volatile("nop;nop;nop");
36. start:
37.     asm volatile("mul x0,x27,x25");
38.     asm volatile("sub x1,x27,x26");
39.     asm volatile("add x2,x27,x26");
40.     asm volatile("eon x3,x24,x22");
41.
42.     asm volatile("mul x5,x1,x2");
43.     asm volatile("lsl x24,x24,12");
44.     asm volatile("orr x4,x24,x22");
45.     asm volatile("sub x28,x28,1"); // i--;
46.
47.     asm volatile("sub x6,x0,x5");
48.     asm volatile("lsl x3,x3,8");
49.     asm volatile("and x22,x5,x4");
50.     asm volatile("add x26,x26,20");
51.
52.     asm volatile("add x27,x27,30");
53.     asm volatile("eor x24,x6,x3");
54.     asm volatile("add x25,x25,10");
55.
56.     asm volatile goto ("cbnz %0, %l[start];"
57.             :
58.             : "r" (i)
59.             :
60.             : start);
61.
62.     if ((j!=30105)||(k!=20090)||(l!=10040)||(m!=(0x33e482255b70f0a5))||(n!=(0x1dc05000))){
63.         time(&timer);
64.         tm_info = localtime(&timer);
65.         strftime(buffer, 26, "%d-%m-%Y %H:%M:%S", tm_info);
66.         fprintf(fp,"%s Error in Iteration = %d Expected : j=30105 k=20090 l=10040 m=0x33e482255b70f0a5
    n=0x1dc05000"
67.                 "Actual : j : %llu k : %llu l : %llu m : 0x%llx n : 0x%llx\n",buffer,(35200000-it),j,k,l,m,n);
68.         fflush(fp);
69.     }
70.
71.     it--;
72.     if(it!=0) goto inf;
73.     fclose(fp);
74.     return 0;
75. }
```

## FPU

```
1.   #include <stdio.h>
2.   #include <stdint.h>
3.   #include <time.h>
4.
5.   register uint64_t  i    asm ("r28");
6.
7.   __attribute__(( aligned(64) )) int main(int argc,char** argv) {
8.
9.       time_t timer;
10.      char buffer[26];
11.      struct tm* tm_info;
12.
13.      register double    j    asm ("d8");
14.      register double    k    asm ("d9");
15.      register double    l    asm ("d10");
16.      register double    t1   asm ("d11");
17.      register double    t2   asm ("d12");
18.      register double    t3   asm ("d13");
19.      register double    t4   asm ("d14");
20.
```

```
21.     register double    x1   asm ("d15");
22.     register double    x2   asm ("d16");
23.     register double    x3   asm ("d17");
24.
25.     FILE *fp = fopen("log-fpu.txt", "a");
26.     if (fp == NULL) {
27.         printf("Error opening file!\n");
28.         return -1;
29.     }
30.     fprintf(fp,"Foo\n");
31.
32.     i = 1100000000;
33.     j = 16.250;         // d8
34.     k = 56364902889004.243; // d9
35.     l = 12.133;         // d10
36.     t1 = 0.0;       // d11
37.     t2 = 0.0;       // d12
38.     t3 = 0.0;       // d13
39.     t4 = 1.25;      // d14
40.
41.     x1 = 0.0;       // d15
42.     x2 = 0.0;       // d16
43.     x3 = 0.0;       // d17
44.
45.     asm volatile("nop;nop;nop;nop;nop;nop;nop;nop");
46. start:
47.     asm volatile("fmadd  d11,d9,d14,d8");
48.     asm volatile("fmadd  d12,d10,d14,d8");
49.     asm volatile("fdiv   d13,d11,d12");
50.     asm volatile("fsqrt  d13,d13");
51.
52.     asm volatile("fmadd  d15,d9,d14,d8");
53.     asm volatile("fmadd  d16,d10,d14,d8");
54.     asm volatile("fdiv   d17,d15,d16");
55.     asm volatile("fsqrt  d17,d17");
56.
57.     asm volatile("fcmp d13,d17");
58.     asm volatile goto ("b.ne %l[error];"
59.             :
60.             :
61.             :
62.             : error);
63.
64. cont:
65.     asm volatile("fadd  d8,d8,d13");
66.     asm volatile("fadd  d9,d9,d13");
67.     asm volatile("fadd  d10,d10,d13");
68.     asm volatile("fadd  d14,d14,d13");
69.
70.     asm volatile("sub x28,x28,1");
71.     asm volatile goto ("cbnz %0, %l[start];"
72.             :
73.             : "r" (i)
74.             :
75.             : start);
76.
77.
78.     return 0;
79.
80. error:
81.     time(&timer);
82.     tm_info = localtime(&timer);
83.     strftime(buffer, 26, "%d-%m-%Y %H:%M:%S", tm_info);
84.     fprintf(fp,"%s SDC --> %f != %f\n",buffer,t3,x3);
85.     fflush(fp);
86.     goto cont;
87.
88. }
```

## Pipeline

```
1.  #include <stdio.h>
2.  #include <stdint.h>
3.  #include <time.h>
4.
5.  #define size_init 255
6.
7.  register uint64_t  i    asm ("r28");
8.  register uint64_t  *arr asm ("r27");
9.  register uint64_t  k    asm ("r26");
10. register uint64_t  size asm ("r25");
11. register uint64_t  r24  asm ("r24");
12. register uint64_t  r22  asm ("r22");
13. register uint64_t  r21  asm ("r21");
14. register uint64_t  r20  asm ("r20");
15.
16. register double   *arr2 asm ("r19");
17.
18. __attribute__(( aligned(64) )) int main(int argc,char** argv) {
19.
20.     time_t timer;
21.     char buffer[26];
22.     struct tm* tm_info;
23.
24.     uint64_t array[size_init+1];
25.     double array2[size_init+1];
26.
27.     register double    d8    asm ("d8");
28.     register double    d9    asm ("d9");
29.     register double    d10   asm ("d10");
30.     register double    d11   asm ("d11");
31.     register double    d12   asm ("d12");
32.     register double    d13   asm ("d13");
33.     register double    d14   asm ("d14");
34.     register double    d15   asm ("d15");
35.     register double    d16   asm ("d16");
36.     register double    d17   asm ("d17");
37.     register double    d18   asm ("d18");
38.
39.     FILE *fp = fopen("log-pipeline.txt", "a");
40.     if (fp == NULL) {
41.         printf("Error opening file!\n");
42.         return -1;
43.     }
44.     fprintf(fp,"Pipeline\n");
45.
46.     arr = array;
47.     arr2 = array2;
48.
49.     i = 3000000; //1100000000;
50.     d8 = 16.250;
51.     d9 = 56364902889004.243;
52.     d10 = 12.133;
53.     d11 = 0.0;
54.     d12 = 0.0;
55.     d13 = 0.0;
56.     d14 = 1.25;
57.     d15 = 0.0;
58.     d16 = 0.2;
59.     d17 = 0.0;
60.
61.     r24 = 29;
62.     r22 = 27;
63.     r21 = 22;
```

```
64.     r20 = 16;
65.
66.     size = 0;
67.  prev:
68.     k = arr[size];
69.     arr[size] = 0x0;
70.     d11 = arr2[size];
71.     arr2[size]= 0.01;
72.     if (++size==size_init) goto next;
73.     goto prev;
74.  next:
75.
76.  prev1:
77.     k = arr[size];
78.     arr[size] = 0x0;
79.     d11 = arr2[size];
80.     arr2[size]= 0.01;
81.     if (!--size) goto next1;
82.     goto prev1;
83.  next1:
84.
85.  prev2:
86.     k = arr[size];
87.     arr[size] = 0x0;
88.     d11 = arr2[size];
89.     arr2[size]= 0.01;
90.     if (++size==size_init) goto next2;
91.     goto prev2;
92.  next2:
93.
94.  start:
95.     asm volatile("fmadd d11,d9,d14,d8");
96.
97.     size = 0;
98.  prev3:
99.     d18 = arr2[size];
100.    arr2[size] = d9 - (double) (size + i);
101.    k = arr[size];
102.    arr[size] = (size * i) ^ (size - i);
103.    if (++size==25) goto next3;
104.    goto prev3;
105.next3:
106.
107.    asm volatile("fmadd d12,d10,d14,d8");
108.
109.    size = 25;
110.prev4:
111.    d18 = arr2[size];
112.    arr2[size] = d9 - (double) (size + i);
113.    k = arr[size];
114.    arr[size] = (size * i) ^ (size - i);
115.    if (++size==49) goto next4;
116.    goto prev4;
117.next4:
118.
119.    asm volatile("fdiv d13,d11,d12");
120.
121.    size = 49;
122.prev5:
123.    d18 = arr2[size];
124.    arr2[size] = d9 - (double) (size + i);
125.    k = arr[size];
126.    arr[size] = (size * i) ^ (size - i);
127.    if (++size==73) goto next5;
128.    goto prev5;
129.next5:
130.
131.    size = 73;
132.prev6:
133.    d18 = arr2[size];
```

```
134.    arr2[size] = d9 - (double) (size + i);
135.    k = arr[size];
136.    arr[size] = (size * i) ^ (size - i);
137.    if (++size==97) goto next6;
138.    goto prev6;
139.next6:
140.
141.    asm volatile("fsqrt  d13,d13");
142.
143.    size = 73;
144.prev7:
145.    d18 = arr2[size];
146.    arr2[size] = d9 - (double) (size + i);
147.    k = arr[size];
148.    arr[size] = (size * i) ^ (size - i);
149.    if (++size==121) goto next7;
150.    goto prev7;
151.next7:
152.
153.    asm volatile("fmadd  d15,d9,d14,d8");
154.
155.    size = 121;
156.prev8:
157.    d18 = arr2[size];
158.    arr2[size] = d9 - (double) (size + i);
159.    k = arr[size];
160.    arr[size] = (size * i) ^ (size - i);
161.    if (++size==145) goto next8;
162.    goto prev8;
163.next8:
164.
165.    asm volatile("fmadd  d16,d10,d14,d8");
166.
167.    size = 145;
168.prev9:
169.    d18 = arr2[size];
170.    arr2[size] = d9 - (double) (size + i);
171.    k = arr[size];
172.    arr[size] = (size * i) ^ (size - i);
173.    if (++size==169) goto next9;
174.    goto prev9;
175.next9:
176.
177.    asm volatile("fdiv   d17,d15,d16");
178.
179.    size = 169;
180.prev10:
181.    d18 = arr2[size];
182.    arr2[size] = d9 - (double) (size + i);
183.    k = arr[size];
184.    arr[size] = (size * i) ^ (size - i);
185.    if (++size==193) goto next10;
186.    goto prev10;
187.next10:
188.
189.    asm volatile("fsqrt  d17,d17");
190.
191.    size = 193;
192.prev11:
193.    d18 = arr2[size];
194.    arr2[size] = d9 - (double) (size + i);
195.    k = arr[size];
196.    arr[size] = (size * i) ^ (size - i);
197.    if (++size==217) goto next11;
198.    goto prev11;
199.next11:
200.
201.    asm volatile("fcmp d13,d17");
202.
203.    size = 217;
```

```
prev12:
    d18 = arr2[size];
    arr2[size] = d9 - (double) (size + i);
    k = arr[size];
    arr[size] = (size * i) ^ (size - i);
    if (++size==241) goto next12;
    goto prev12;
next12:

    asm volatile goto ("b.ne %l[error1];"
            :
            :
            :
            : error1);

cont1:

    size = 0;
prev13:
    if ( arr[size] != ((size * i) ^ (size - i))) goto error2;
cont2:
    if ( arr2[size] != (d9 - (double) (size + i))) goto error3;
cont3:
    if (++size==241) goto next13;
    goto prev13;
next13:

    asm volatile("fadd  d8,d8,d13");
    asm volatile("fadd  d9,d9,d13");
    asm volatile("fadd  d10,d10,d13");
    asm volatile("fadd  d14,d14,d13");

    asm volatile("sub x28,x28,1");
    asm volatile goto ("cbnz %0, %l[start];"
            :
            : "r" (i)
            :
            : start);

    return 0;

error1:
    time(&timer);
    tm_info = localtime(&timer);
    strftime(buffer, 26, "%d-%m-%Y %H:%M:%S", tm_info);
    fprintf(fp,"%s SDC --> %f != %f\n",buffer,d13,d17);
    fflush(fp);
    goto cont1;
error2:
    time(&timer);
    tm_info = localtime(&timer);
    strftime(buffer, 26, "%d-%m-%Y %H:%M:%S", tm_info);
    fprintf(fp,"%s SDC --> %llx != %llx\n",buffer,arr[size],((size * i) ^ (size - i)));
    fflush(fp);
    goto cont2;
error3:
    time(&timer);
    tm_info = localtime(&timer);
    strftime(buffer, 26, "%d-%m-%Y %H:%M:%S", tm_info);
    fprintf(fp,"%s SDC --> %f != %f\n",buffer,arr2[size],(d9 - (double) (size + i)));
    fflush(fp);
    goto cont3;
}
```

# ANNEX II

This annex is part of a subsection of the ARMv8 Manual [32] that describes some of the architectural and microarchitectural events we counted with help of the performance counters while developing the diagnostic micro-viruses along with their associated event numbers and mnemonics.

I. 0x001, L1I_CACHE_REFILL, Attributable Level 1 instruction cache refill

The counter counts Attributable instruction memory accesses that cause a refill of at least the Level 1 instruction or unified cache. This includes each instruction memory access that causes a refill from outside the cache. It excludes accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss. A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache. The counter does not count cache maintenance instructions.

II. 0x003, L1D_CACHE_REFILL, Attributable Level 1 data cache refill

The counter counts each Attributable memory-read operation or Attributable memory-write operation that causes a refill of at least the Level 1 data or unified cache from outside the Level 1 cache. Each access to a cache line that causes a new linefill is counted, including those from instructions that generate multiple accesses, such as load or store multiples, and PUSH and POP instructions. In particular, the counter counts accesses to the Level 1 cache that cause a refill that is satisfied by another Level 1 data or unified cache, or a Level 2 cache, or memory. A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.
The counter does not count:

- Accesses that do not cause a new Level 1 cache refill but are satisfied from refilling data of a previous miss.
- Accesses to a cache line that generate a memory access but not a new linefill, such as write-through writes that hit in the cache.
- Cache maintenance instructions.
- A write that writes an entire line to the cache and does not fetch any data from outside the Level 1 cache, for example:
  - — A write of a full cache line from a coalescing buffer.
  - — A DC ZVA operation.
- A write that misses in the cache, and writes through the cache without allocating a line.

III. 0x004, L1D_CACHE, Attributable Level 1 data cache access

The counter counts each Attributable memory-read operation or Attributable memory-write operation that causes a cache access to at least the Level 1 data or unified cache. Each access to a cache line is counted including the multiple accesses of instructions, such as LDM or STM. Each access to other Level 1 data or unified memory structures, for example refill buffers, write buffers, and

write-back buffers, is also counted. The counter does not count cache maintenance instructions.

IV. 0x011, CPU_CYCLES, Cycle

The counter increments on every cycle. All counters are subject to changes in clock frequency, including when a WFI or WFE instruction stops the clock. This means that it is CONSTRAINED UNPREDICTABLE whether or not CPU_CYCLES continues to increment when the clocks are stopped by WFI and WFE instructions.

V. 0x014, L1I_CACHE, Attributable Level 1 instruction cache access

The counter counts Attributable instruction memory accesses that access at least the Level 1 instruction or unified cache. Each access to other Level 1 instruction memory structures, such as refill buffers, is also counted.

VI. 0x016, L2D_CACHE, Attributable Level 2 data cache access

The counter counts Attributable memory-read or Attributable memory-write operations, that the PE made, that access at least the Level 2 data or unified cache. Each access to a cache line is counted including refills of and write-backs from the Level 1 data, instruction, or unified caches. Each access to other Level 2 data or unified memory structures, such as refill buffers, write buffers, and write-back buffers, is also counted.
The counter does not count:

- Operations made by other PEs that share this cache.
- Cache maintenance instructions.

VII. 0x017, L2D_CACHE_REFILL, Attributable Level 2 data cache refill

The counter counts Attributable memory-read or Attributable memory-write operations, that the PE made, that access at least the Level 2 data or unified cache and cause a refill of a Level 1 data, instruction, or unified cache or of the Level 2 data or unified cache. Each read from or write to the cache that causes a refill from outside the Level 1 and Level 2 caches is counted.
A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.
For example, the counter counts:

- Accesses to the Level 2 cache that cause a refill that is satisfied by another Level 2 cache, a Level 3 cache, or memory.
- Refills of and write-backs from any Level 1 data, instruction or unified cache that cause a refill from outside the Level 1 and Level 2 caches.
- Accesses to the Level 2 cache that cause a refill of a Level 1 cache from outside of the Level 1 and Level 2 caches, even if there is no refill of the Level 2 cache.

The counter does not count, as events on this PE:

- Accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.

- Accesses to the Level 2 cache that generate a memory access but not a new linefill, such as write-through writes that hit in the Level 2 cache.
- Accesses to the Level 2 cache that are part of a Level 1 cache refill or write-back that hit in the Level 2 cache so do not cause a refill from outside of the Level 1 and Level 2 caches.
- Operations made by other PEs that share this cache.
- Cache maintenance instructions.
- A write that writes an entire line to the cache and does not fetch any data from outside the Level 1 and Level 2 caches, for example:
  - — A write-back from a Level 1 cache to a Level 2 cache.
  - — A write from a coalescing buffer of a full cache line.
  - — A DC ZVA operation.
- A write that misses in the cache, and writes through the cache without allocating a line.

VIII. 0x018, L2D_CACHE_WB, Attributable Level 2 data cache write-back

The counter counts every write-back of data from the Level 2 data or unified cache that occurs as a result of an operation by this PE. It counts each write-back that causes data to be written from the Level 2 cache to outside the Level 1 and Level 2 caches. For example, the counter counts:

- A write-back that causes data to be written to a Level 3 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 2 cache.

Each write-back is counted once, even if it requires multiple accesses to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of data from the Level 2 cache to outside the Level 1 and Level 2 cache made as a result of a coherency request.
- Write-backs made as a result of Cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to a Level 3 cache or memory.
- Writes from the PE or Level 1 data or unified cache that write through the Level 2 cache to outside the Level 1 and Level 2 caches.
- Transfers of data from the Level 2 cache to a Level 1 cache, to satisfy a Level 1 cache refill.

An Unattributable write-back event occurs when a requestor outside the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines streaming writes to memory and does not allocate lines to the cache, or by a DC ZVA operation.

# REFERENCES


[1] E. Le Sueur, and G. Heiser, “Dynamic voltage and frequency scaling: the laws of diminishing returns”, in *International Conference on Power Aware Computing and Systems (HotPOWER)*, 2010.

[2] V. J. Reddi, M. S. Gupta, G. H. Holloway, G.-Y. Wei, M. D. Smith, and D. M. Brooks, “Voltage emergency prediction: Using signatures to reduce operating margins”, in *International Conference on High-Performance Computer Architecture (HPCA)*, 2009, pages 18–29.

[3] S. Herbert, and D. Marculescu, “Variation-aware dynamic voltage/frequency scaling”, in *International Conference on High-Performance Computer Architecture (HPCA)*, 2009, pages 301–312.

[4] N. Kapadia, and S. Pasricha, “VARSHA: Variation and reliability-aware application scheduling with adaptive parallelism in the dark silicon-era”, in *Design, Automation & Test in Europe Conference (DATE)*, 2015, pages 1060-1065.

[5] P. Raghunathan, Y. Turakhia, S. Garg, and D. Marculescu, “Cherry-picking: Explpoiting process variations in dark-silicon homogeneous chip multi-processors”, in *Design, Automation & Test in Europe Conference (DATE)*, 2013, pages 39-44.

[6] R. Teodorescu, and J. Torrellas, “Variation-aware application scheduling and power management for chip multiprocessors”, in *International Symposium on Computer Architecture (ISCA)*, 2008, pages 363–374.

[7] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, “Razor: A low-power pipeline based on circuit-level timing speculation”, in *International Symposium on Microarchitecture (MICRO)*, 2003, pages 7-18.

[8] Y. Zu, C. R. Lefurgy, J. Leng, M. Halpern, M. S. Floyd, and V. J. Reddi, "Adaptive guardband scheduling to improve system-level efficiency of the POWER7+", in *International Symposium on Microarchitecture (MICRO),* 2015, pages 308-321.

[9] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, C. Magdalinos, D. Gizopoulos, “Voltage Margins Identification on Commercial x86-64 Multicore Microprocessors”, in IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), 2017.

[10] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. S. Allen-Ware, B. Brock, J.A. Tierno, and J. B. Carter, “Active management of timing guardband to save energy in POWER7”, in *International Symposium on Microarchitecture (MICRO),* 2009, pages 1–11.

[11] A. Bacha, and R. Teodorescu, “Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors”, in *International Symposium on Computer Architecture (ISCA),* 2013, pages 297–307.

[12] A. Bacha, and R. Teodorescu, “Using ECC feedback to guide voltage speculation in low-voltage processors”, in *International Symposium on Microarchitecture (MICRO),* 2014, pages 306–318.

[13] A. Bacha, and R. Teodorescu, “Authenticache: Harnessing cache ECC for system authentication”, in *International Symposium on Microarchitecture (MICRO)*, 2015, pages 128–140.

[14] P. N. Whatmough, S. Das, Z. Hadjilambrou, and D. M. Bull, "An all-digital power-delivery monitor for analysis of a 28nm dual-core ARM Cortex-A57 cluster", in *International Solid-State Circuits Conference (ISSCC)*, 2015, pages 262-264.

[15] P. N. Whatmough, S. Das, and D. M. Bull, "Analysis of adaptive clocking technique for resonant supply voltage noise mitigation", in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2015, pages 128-133.

[16] M. Ketkar, and E. Chiprout, “A microarchitecture-based framework for pre- and post-silicon power delivery analysis”, in *International Symposium on Microarchitecture (MICRO)*, 2009, pages 179–188.

[17] Y. Kim, and L. K. John, “Automated di/dt stressmark generation for microprocessor power delivery networks”, in *International Symposium on Low Power Electronics and Design (ISPLED)*, 2011, pages 253-258.

[18] Y. Kim, L. K. John, S. Pant, S. Manne, M. Schulte, W. L. Bircher, and M. S. S. Govindan, “AUDIT: Stress Testing the Automatic Way”, in *International Symposium on Microarchitecture (MICRO)*, 2012, pages 212–223.

[19] M. S. Gupta, V. J. Reddi, G. Holloway, G.-Y. Wai, and D. M. Brooks, “An event-guided approach to reducing voltage noise in processors”, in *Design, Automation & Test in Europe Conference (DATE)*, 2009, pages 160-165.

[20] V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. D. Smith, G.-Y. Wei, and D. Brooks, “Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling”, in *International Symposium on Microarchitecture (MICRO)*, 2010, pages 77–88.

[21] M. S. Gupta, K. K. Rangan, M. D. Smith, G.-Y. Wei, and D. Brooks, “Towards a software approach to mitigate voltage emergencies”, in *International Symposium on Low Power Electronics and Design (ISPLED)*, 2007, pages 123-128.

[22] R. Joseph, D. Brooks, and M. Martonosi, “Control techniques to eliminate voltage emergencies in high performance processors”, in *International Conference on High-Performance Computer Architecture (HPCA)*, 2003, pages 79–90.
[23] T. N. Miller, R. Thomas, X. Pan, and R. Teodorescu, “VRSync: Characterizing and eliminating synchronization-induced voltage emergencies in many-core processors”, in *International Symposium on Computer Architecture (ISCA)*, 2012, pages 249–260.
[24] M. D. Powel, and T. N. Vijaykumar, “Pipeline muffling and a priori current ramping: architectural techniques to reduce high-frequency inductive noise”, in *International Symposium on Low Power Electronics and Design (ISPLED)*, 2003, pages 223-228.
[25] M. S. Gupta, K. K. Rangan, M. D. Smith, G.-Y. Wei, and D. Brooks, “DeCoR: A Delayed Commit and Rollback mechanism for handling inductive noise in processors”, in *International Conference on High-Performance Computer Architecture (HPCA)*, 2008, pages 381–392.
[26] B. Gopireddy, C. Song, J. Torellas, N. S. Kim, A. Agrawal, and A. Mishra, “ScalCore: Designing a core for voltage scalability”, in *International Conference on High-Performance Computer Architecture (HPCA)*, 2016, pages 681–693.
[27] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, “Trading off cache capacity for reliability to enable low voltage operation”, in *International Symposium on Computer Architecture (ISCA)*, 2008, pages 203–214.
[28] Z. Chishti, A. R. Alameldeen, C. Wilkerson, W. Wu, and S.-L. Lu, “Improving cache lifetime reliability at ultra-low voltages”, in *International Symposium on Microarchitecture (MICRO)*, 2009, pages 89–99.
[29] H. Duwe, X. Jian, D. Petrisko, and R. Kumar, “Rescuing uncorrectable fault patterns in on-chip memories through error pattern transformation”, in *International Symposium on Computer Architecture (ISCA)*, 2016, pages 634–644.
[30] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos, “Software-based self-test for small caches in microprocessors”, in *IEEE Transactions on Computer-Aided Design and of Integrated Circuits and Systems*, Vol. 33, Issue 12, Dec. 2014, pages 1991-2004.
[31] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda, “Microprocessor software-based self-testing”, in *IEEE Design and Test in Computers*, Vol. 27, Issue 3, May 2010, pages 4-19.
[32] ARM® Cortex®-A Series: Programmer’s Guide for ARMv8-A, Version: 1.0, March 2015. http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf
[33] BUILD A 64-BIT KERNEL FOR YOUR RASPBERRY PI 3. https://devsidestory.com/build-a-64-bit-kernel-for-your-raspberry-pi-3/
[34] John L. Hennessy, David A. Patterson, “Computer Architecture, Fifth Edition: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design).”
[35] Smith, Alan Jay (1982-09-01), "Cache Memories” in ACM Computing Surveys (CSUR), Vol. 14, Issue 3, Sept. 1982, ages 473-530 .
[36] Norman P. Jouppi. “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”, in ISCA '90 Proceedings of the 17th annual international symposium on Computer Architecture, pages 364-373.
[37] Yan Solihin, “Fundamentals of Parallel Multicore Architecture.”
[38] Taylor George, Davies Peter, Farmwald Michael, "The TLB Slice - A Low-Cost High-Speed Address Translation Mechanism", Proceeding ISCA '90 Proceedings of the 17th annual international symposium on Computer Architecture, pages 355-363.
[39] Matthew Dillon, "Page Coloring", Design elements of the FreeBSD VM system, FreeBSD Foundation, Retrieved 19-05-2017.
[40] Yehuda Afek, Dave Dice, Adam Morrison, “Cache index-aware memory allocation” in ISMM '11 Proceedings of the international symposium on Memory management, pages 55-64.