Accurate Simulation of Data Movement in Modern Mobile Multicore Systems

Quentin Huppert

THESIS TO OBTAIN THE DEGREE OF DOCTOR
OF THE UNIVERSITY OF MONTPELLIER

In SyAM - Automatic and Microelectronic Systems

Doctoral School: Information, Structures and Systems Sciences (I2S)

Research Unit: LIRMM

Accurate Simulation of Data Movement in Modern Mobile Multicore Systems

Presented by Quentin Huppert
December 6, 2022

Under the supervision of
Lionel Torres and David Novo

In front of the jury composed of:

MUNIER Alix, Professor at Sarbonne University, LIP6	President
PETROT Frédéric, Professor at University of Grenoble, TIMA	Reporter
SENTIEYS Olivier, Professor at University of Rennes, IRISA/ENSSAT	Reporter
CATTHOOR Francky, Professor at KU Leuven, IMEC	Examiner
TORRES Lionel, Professor at University of Montpellier, LIRMM	Thesis Director
NOVO David, Research Scientist at CNRS, LIRMM	Thesis co-supervisor

UNIVERSITÉ DE MONTPELLIER

Abstract

Computer system architectures have become increasingly complex. Pushing for better performance and lower energy consumption, they include multiple cores, GPUs, accelerators, and sophisticated memory hierarchies. In this context, computer system simulators play a significant role in the research architecture community. They allow the evaluation of new architecture ideas while being quicker and avoiding the huge costs of manufacturing processes. The simulated architectures are often compared with real state-of-the-art baseline architectures. Thus, the relevance of the conclusions from such simulators is directly related to the faithfulness of the baseline model. We observe that having strong baselines is not a simple task due to the lack of detailed technical information about commercial architectures. In this thesis, we propose a systematic methodology to extract information from real commercial architectures for the calibration of their corresponding simulation models. This methodology is based on handcrafted microbenchmarks that leverage hardware performance counters for non-intrusive monitoring. We instantiate this methodology on two key components of the architecture.

First, we target the different levels of the memory hierarchy. We design the microbenchmarks to produce a signature of the memory system levels. The signature contains parameter values that are important to calibrate a simulation model of the real architecture. We implement the methodology with the gem5 simulator and an ARM Cortex-A53 CPU. Then, we evaluate our methodology with benchmarks from the SPEC CPU2006 suite. The benchmarks are executed on the commercial CPU and gem5 using our calibrated configuration. The results show that our calibration reduces the average and maximum simulation error by 43% and 62%, respectively.

Second, we instantiate the methodology on the data prefetching engine, which is a key component of the memory system that can generate memory requests in parallel to the core. This new instantiation of our methodology results in Pref-X, a framework that enables functional analysis of data prefetching engines in in-order cores. For that, the framework uses memory request sequences to stress the L1 data cache and expose

the changes in L1 cache content. From those changes, Pref-X deduces the functional specifications of the data prefetching engine. Finally, we use the extracted information to build a functional model of the prefetching engine. We apply Pref-X to two in-order ARM CPUs, the Cortex-A7 and the Cortex-A53, and evaluate its accuracy with memory traces extracted from the SPEC CPU2006 suite. We execute the traces on both the commercial CPUs and their corresponding functional models produced by our framework. The results show an average prefetching accuracy of 99.8% and 96.9% for the Cortex-A7 and the Cortex-A53, respectively.

This thesis builds a detailed understanding of how microbenchmarks can be used to improve the modeling accuracy of existing architectures. We believe our methodology has the potential to be applied to other components beyond the ones covered in this manuscript, but that is part of future work. We hope our results will enable the community to use better-calibrated baseline architectures in their simulations.

Résumé de la Thèse

Les architectures de systèmes informatiques sont devenues de plus en plus complexes. Dans le but d'améliorer les performances et de réduire la consommation d'énergie, elles comprennent plusieurs CPU, des GPU, des accélérateurs ainsi que des hiérarchies de mémoire sophistiquées. De ce fait, les simulateurs de systèmes informatiques jouent un rôle important dans la communauté des chercheurs. Ils permettent d'évaluer de nouvelles idées d'architecture, de façons plus rapides, tout en évitant les coûts des processus de fabrication.

Les architectures simulées sont souvent comparées à des architectures réelles de l'état de l'art. Ainsi, la pertinence des résultats obtenus depuis un simulateur est directement liée à la fidélité du modèle de référence. Nous observons qu'il n'est pas simple d'avoir des modèles de référence solides, ceci en raison du manque d'informations techniques détaillées des architectures commerciales. Dans le cadre de cette thèse, nous proposons une méthodologie systématique pour extraire des informations depuis des architectures commerciales réelles. Ces informations sont ensuite utilisées afin de calibrer les modèles de référence correspondants. Cette méthodologie est basée sur le design de microbenchmarks et l'utilisation de compteurs de performance qui permettent une mesure non-intrusive du système. Nous appliquons cette méthodologie sur deux composants clés de l'architecture.

Dans un premier temps, nous ciblons les différents niveaux de la hiérarchie de la mémoire. Nous développons des microbenchmarks qui permettent d'extraire une signature caractéristique du système de mémoire. Cette signature contient des valeurs de paramètres importantes que nous utilisons pour la calibration du modèle de référence. Nous implémentons la méthodologie sur le simulateur gem5 et un processeur ARM, le Cortex-A53. Puis, nous évaluons la méthodologie avec des benchmarks de la suite SPEC CPU2006. Ces derniers sont exécutés sur l'architecture ARM et simulés sur gem5 en utilisant notre configuration calibrée. Les résultats montrent que notre calibration réduit l'erreur de simulation moyenne et maximale de 43% et 62%, respectivement.

Dans un second temps, nous appliquons la méthodologie sur le data prefetcher, qui

est un composant clé du système de mémoire. Celui-ci pouvant générer des demandes de mémoire en parallèle du cœur du processeur. Cette nouvelle instance de notre méthodologie donne lieu à Pref-X, un framework permettant l'analyse fonctionnelle des data prefetchers présents dans les coeurs in-order. Pour cela, nous utilisons des séquences de requêtes mémoire pour stresser le data prefetcher et exposer les changements dans le contenu de la cache L1. A partir de ces changements, Pref-X en déduit les spécifications fonctionnelles du data prefetcher qui permettent la construction un modèle fonctionnel. Nous implementons Pref-X à deux processeurs ARM in-order, le Cortex-A7 et le Cortex-A53. Nous évaluons les modèles fonctionnels résultants avec des traces mémoire extraites de la suite SPEC CPU2006. Nous exécutons les traces à la fois sur les CPU commerciaux et sur les modèles fonctionnels produits par notre framework. Les résultats montrent en moyenne une précision des modèles de 99,8% pour le Cortex-A7 et de 96,8% pour le Cortex-A53.

Cette thèse permet de comprendre en détail comment l'utilisation de microbenchmarks permet une amélioration de la précision des modèles de référence au sein des simulateurs. Aussi, nous pensons cette méthodologie est applicable à d'autres éléments de l'architecture, ceci pouvant faire l'objet de travaux futurs. Finalement, ces travaux contribuent à la communauté scientifique en permettant l'utilisation de modèles de référence d'architecture commerciale mieux calibrés.

Acknowledgements

I would like to thank all the jury members for having evaluated my thesis. I want to thank Professor PETROT Frédéric and Professor SENTIEYS Oliver for having reviewed this manuscript. And, I thank Professor MUNIER Alix for having accepted to be the president of this jury.

I would like to express my gratitude to my thesis director, Professor Lionel Torres for having supervised and advised me during those last three years. I met him during my engineering school years. With his lectures, he showed me the world of computer system architecture and made me want to go deeper into its exploration.

He introduced me to Doctor David Novo who has been my co-supervisor. I especially want to thank him for all the things he has shared with me. His knowledge, experience, and also his own office. This manuscript reflects all the methodologies that he has taught me through those years.

I also want to thank Professor Francky Catthoor for having welcomed me at imec during my engineering school internships. He allowed the collaboration between imec and the university of Montpellier during my thesis.

Thanks to my co-workers who always have been helpful and supportive. Especially during the covid 19 pandemia. I am very grateful to all laboratory members for their kindness and solidarity.

Lastly, thanks to my family and my fiancé for being supportive and understanding all along the challenges of this thesis.

List of Figures

List of Tables

Contents

I

Introduction

Modern computer systems have become increasingly complex. In order to keep improving both performance and energy efficiency, modern computer architectures include multiple processing units such as CPUs, GPUs, or accelerators. All the processing units share the same memory resulting in a meticulously orchestrated hierarchy of distributed private and shared memories. Additionally, the utilization spectrum of computer systems becomes very wide, from very energy-efficient embedded systems to high-performance computing.

1.1 Context

In this section, we present how researchers continue improving modern architecture by exploring new designs of computer architectures. In particular, we see that the memory system is a critical component of the architecture, leading to many specific explorations. We further introduce computer architecture simulators as an alternative to chip manufacturing for architecture exploration. We finally expose simulator drawbacks and limitations, which motive this thesis.

1.1.1 Computer Architecture Exploration

In order to keep improving performance and energy efficiency, researchers continue exploring new designs across all components of the architecture components. Indeed, multiple components other than the CPUs play a significant role in global architecture performance, such as the GPUs or the memory system. The main challenge of such explorations is evaluating new designs' benefits against state-of-the-art references. A straightforward solution would be to conduct the exploration by manufacturing new chips that implement the new design. However, a strong drawback would be the cost of such explorations. Chip manufacturing is a very long and expensive process mainly reserved for commercial platforms. In addition, during architecture exploration, we may evaluate non-existing technologies to identify ideal cases and motivate exploration directions. Also, the monitoring is often limited with actual manufactured chips and can disturb normal behavior.

1.1.2 Memory System

The memory system is an essential part of computer system architecture. It provides necessary data around all the processing units present in the architecture. As a result, the memory system significantly impacts global architecture performance. Indeed, a slow reactive memory system can slow down the complete architecture. Additionally, the memory system represents a big part of the system's energy consumption [1]. Consequently, Modern computer architectures implement complex memory organizations, including different memory technologies, layouts, and protocols to keep high performance. Regarding all these aspects, the memory system is a prevalent topic of exploration within the system architect community.

Moreover, emerging non-volatile memory technologies provide new opportunities for memory system improvement [2]. However, those technologies have different technical specifications than existing commercial memory technologies. Thus, the different memory protocols and organizations must be rethought. Many explorations on using such non-volatile technologies have already been done [3, 4, 5]. Nevertheless, some more still need to be made as most of those technologies are not mature yet. They may become even more attractive in the future.

Architecture simulators are essential tools used by academic and industrial researchers to address all these challenges in modern memory systems.

1.1.3 Architecture Simulator

Computer architecture simulators have become an essential tool for computer architecture research [6, 7, 8, 9]. Contrary to chip manufacturing, they allow quick and inexpensive evaluations of new architecture ideas. Computer architecture simulators are already widely used by the research community. For instance, the gem5 simulator [10] has already been cited several thousand times in the last decade. However, computer simulators suffer from drawbacks such as the trade-off between simulation time and model accuracy. The most accurate simulations can achieve very long simulation times. Conversely, the simulation results from an inaccurate model would lead to misleading conclusions [11]. Thus, it is crucial to select an appropriate level of accuracy.

With computer simulators, the evaluation of new ideas is made against a reference baseline model. This baseline model represents a realistic state-of-the-art architecture. A common error with such evaluation is to use a flawed baseline model, which com-

promises the relevance of evaluation results. However, calibrating a baseline model can be difficult. Indeed, key commercial architecture technical information is often not public, hindering baseline model calibration.

1.1.4 Simulation Verification

To evaluate the accuracy of a simulator, researchers verify them against real architectures. They first calibrate the simulation model before simulating and executing realistic benchmarks on both the simulator and the target. Thus, the simulation error is the difference between the simulator's and target's results. In order to prove the accuracy of the simulator, the simulation error needs to be the lowest. In this way, several works [12, 13, 14] propose verifying the gem5 simulator against architectures. The average error is around 2% within 18%. However, those evaluations cover only a few points of the all gem5 architecture coverage. Moreover, the empirical calibration methods used during those evaluations cannot be generalized to other architectures, limiting the verification of the rest of the architecture coverage. This situation worsens as state-of-the-art architectures keep advancing, making previous evaluations outdated.

1.2 Contributions

In this thesis, we tackle the problem of accurate computer simulations. Specially, we focus on a particular type of error that is recurrent in computer architecture simulators, the flawed calibrations of the simulation baseline. The memory system has a significant role in general, but more importantly, in multicore architectures. Consequently, our goal is to calibrate precisely the memory system of multicore architectures against real state-of-the-art commercial architectures to provide more accurate simulations.

To this end, we propose in this thesis three main contributions that we introduce in the continuation of this section.

1.2.1 Calibration Methodology

The first contribution is a systematic methodology that we propose to calibrate the memory system of a computer architecture simulation. The calibration is made against a real state-of-the-art architecture that we call *target* architecture. The methodology is composed of two phases. We determine the simulator parameters we need to calibrate

during the first one. Then, we calibrate them in the second phase of the methodology. For that, we start using first-party documentation. However, as part of the technical information is not public, we propose a method base on handcrafted microbenchmarks and hardware monitoring to extract missing technical information from the target architecture. In parallel, we use a running example to illustrate the different steps throughout the methodology description. This contribution is presented in Chapter 4.

1.2.2 Memory System Simulator Calibration

The second contribution is an instantiation of the proposed methodology on the reactive part of the memory system, i.e., the cache levels and the main memory. We illustrate the path of memory requests through the memory hierarchy and model it as a graph of conditions and delays. We then use it to detail the design of microbenchmarks that we execute to extract technical information from target commercial architectures. We implement the resulting instantiation on the gem5 simulator [10] and one Cortex-A53 of the MediaTek Helio X20 SoC that we select as target state-of-the-art architecture [15]. Finally, we evaluate our methodology with benchmarks from the SPEC CPU2006 suite. We execute them on the board to have a reference, and then we simulate them on gem5 using the default and calibrated models to expose our methodology's benefits. This contribution is introduced in Chapter 5.

1.2.3 Analysis of In-Order CPU Data Prefetcher

The last contribution is Pref-X [16], a framework to analyze functional characteristics of data prefetching in commercial in-order cores. Pref-X instantiates the proposed methodology on a commercial data prefetcher engine. It exposes the data prefetching activity by *X-raying* the cache memory content after triggering the data prefetcher. That way, a complete functional data prefetcher model can be extracted from the target commercial architecture. The functional model is verified in a second phase with realistic benchmark memory traces. Finally, we demonstrate the feasibility of this methodology by implementing Pref-X on two ARM in-order cores, the Cortex-A7, and the Cortex-A53. We show a functional accuracy of 99.9% and 96.8% for the Cortex-A7 and the Cortex-A53, respectively. This contribution is described in Chapter 6.

1.3 Outline

We structure this thesis around seven chapters. Chapter 2 introduces relevant background about multicore memory systems. It also introduces two simulators that we further use during evaluations. Chapter 3 discusses related work about computer simulators. E.g., their different categories, their verification or utilization. Chapter 4 introduces our systematic methodology and details the two phases that compose it. Chapter 5 and Chapter 6 present two instances of the methodology on the memory system and the data prefetcher, respectively. Finally, Chapter 7 concludes this thesis by summarizing the key achievements and discussing future work.

II

Background

In this chapter, we introduce the main components present in the memory system of modern multicore systems. Also, we present key processes of the memory system, such as data prefetching. Finally, we present the simulators we use to evaluate new architectural ideas and especially new memory architecture designs.

2.1 Memory System

Modern computer architectures are composed of multiple processing units such as CPUs, GPUs, or accelerators. The memory system has the role of providing data and instructions to each processing unit. Consequently, the memory system has a significant impact on global architecture performance [17, 18, 19]. Indeed, a slow data access time or a low memory bandwidth can slow down the whole architecture.

2.1.1 Memory Hierarchy

Depending on its organization and technology, memory can have very different characteristics [20, 21, 22], e.g., small and fast, or big and slow. In this way, the memory system is composed of different types of memories. These memories are organized in a hierarchy as illustrated in Figure 2.1. The first level of the hierarchy is the process unit registers. Then there are three different kinds of memories:

- On-chip Caches. The memory caches are small but very fast. They can be private or shared between several processing units. They are divided into different levels that we call L1, L2, and L3. The L1 is the closest memory to the processing unit. The Last Level Cache (LLC) is the farthest memory cache from the processing unit. Additionally, modern heterogeneous SoCs implement System Level Cache (SLC), which is beyond the LLC and shared between all the system components, i.e., CPUs, GPUs, Neural Processing Units (NPUs), and other accelerators. Usually, they are implemented with an SRAM technology. Their sizes go from tens of kilobytes for the L1 to tens of megabytes for the LLC and SLC.

- The main memory is a fundamental element of the memory system, as this memory is shared between all the processing units of the architecture. This memory contains all the data necessary for Operating System (OS) routine and program execution. The main memory is mostly implemented with DRAM technology using standardized interface, e.g., DDR4, LPDDR4. The main memory size goes

from a few gigabytes for embedded systems to hundreds of gigabytes for big servers.

- The storage memory is the largest memory of the memory system but also the slowest. Contrary to the main memory and the caches, the storage memory is Non-Volatile Memory (NVM). Thus, the memory can retain the stored data even without power. The common non-volatile technology used for storage memory is the NAND-Flash. The size of the storage memory is usually at least a few gigabytes. However, the maximum size cannot be defined as we can always increase the size, e.g., plugging a new memory disk or using online storage.

This distribution of technologies in memory hierarchy is also due to other metrics such as the cost per bit or cell endurance. For instance, the L1 cache has high endurance, contrary to the storage memory. Indeed, the L1 cache is more written than the storage memory. In the same way, the cost per bit of the L1 is higher than the storage memory. Thus, the memory hierarchy's purpose is to balance each technology's benefits and drawbacks.

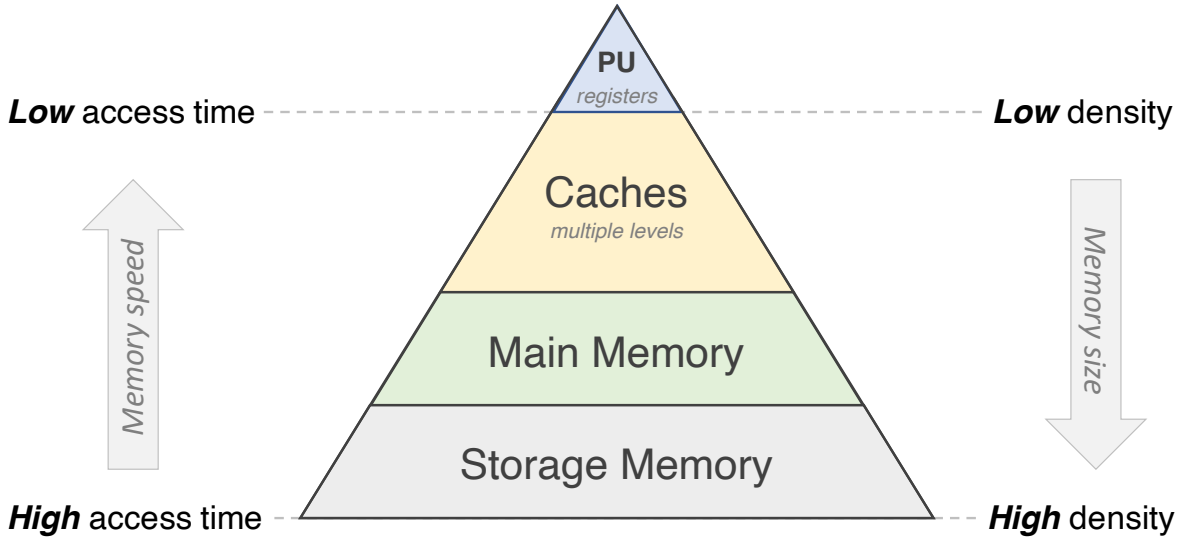

Figure 2.1: *Memory hierarchy diagram.*

2.1.2 Data Movement

The general purpose of the memory hierarchy is to place *hot* data, i.e., frequently used data, close to the processing units to reduce their access times. For that, the memory hierarchy is based on two principals [20] that characterize data utilization during program execution: First, the temporal locality. If a program uses data once, it is likely to

use it again in a short time. Second, the spatial locality. If a program uses data, it is likely to use its neighbor.

Thus, when a *word* of typically eight bytes is requested by the processing unit, a complete *cache line* of typically sixty-four bytes, including the word's neighbors, is moved to the L1. That way, if the processing unit further requests one of them, the word would already be present in the L1, exploiting both temporal and spatial localities. As the size of the L1 is limited, when the L1 cache is full, one cache line needs to be moved back to a lower level before the L1 receives a new one. The *cache replacement policy* is the set of rules that dictate which address is evicted from the cache. For instance, the Last Recent Used (LRU) cache replacement policy selects the coldest data to evict from the cache.

When a cache line is requested to a cache level, a *cache hit* occurs if the cache line is present in the requested cache. Contrary, a *cache miss* occurs when a requested cache line is missing in the requested cache. Thus, if a memory request causes an L1 cache miss, the request is then sent to the L2 cache, where it can either produce a cache hit or a cache miss. Furthermore, we call *miss (hit) rate* the ratio of miss (hit) out of the total number of accesses to a specific cache level. This metric is used to measure memory system activity and optimization. During program execution, an ideal case would be an L1 miss rate of 0%. I.e., all the memory requests are issued with the minimum access time.

Finally, the granularity of data transfers between the main and storage memory is a *page* which generally contains four kilobytes. A *page fault* occurs when the requested address is missing in the main memory. In this case, a complex routine is run by the Operating System (OS) to evict one page from the main memory and replace it with the one containing the requested address. This operation is very long (e.g., thousands of CPU cycles) compared to access to the caches or the main memory.

2.1.3 Illustrating Example

We use an illustrating example to present an existing commercial memory hierarchy. Figure 2.2 shows the architecture of the MediaTek Helio X20 SoC [15]. This commercial platform contains three clusters providing different performance and energy consumption ratios. ARM introduced this organization with the big.LITTLE concept [23]. Clusters 0 and 1 contain four Cortex-A53 each. They are the low and middle-performance clusters. Cluster 2 contains two Cortex-A72. It is a high-performance cluster. All the clusters contain a dedicated L2 cache shared between all the CPUs of the cluster. An

interconnect allows connecting all the processing units, i.e., the clusters, GPUs, and accelerators, to the main and storage memory.

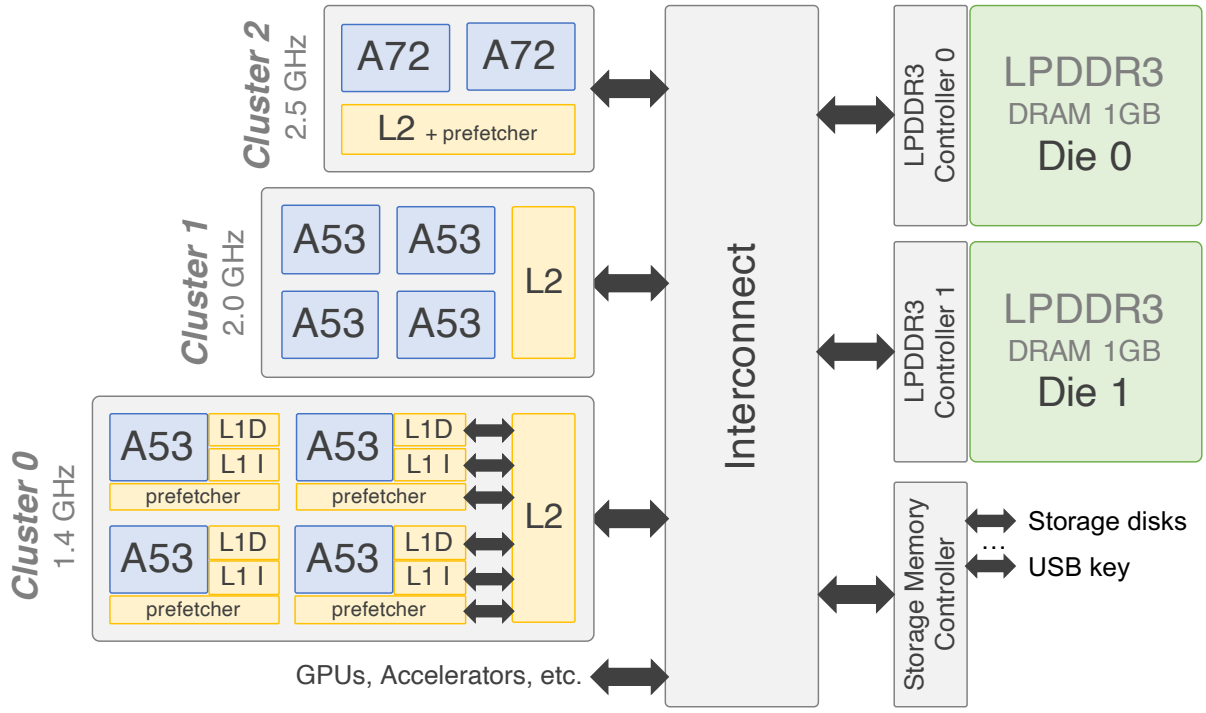

Figure 2.2: *Heterogeneous multi-core architecture of the MediaTek Helio X20.*

As illustrated with cluster 0, each CPU contains a private L1 cache. This one is separated into two caches: the L1 data (L1D) cache and the L1 instruction (L1I) cache. Thus, at this level of the memory hierarchy, the data and the program instruction are stored in two different memories.

2.2 Memory Components

In this section, we introduce important elements and features of the memory system. We explain what is the virtual address space and how the memory system manages it. Then, we describe the organization of the set-associative caches and the main memory. Finally, we introduce the data prefetching detailing its purpose and main characteristics.

2.2.1 Virtual Space and the Memory Management Unit

When a program is launched, the OS generates a *virtual* address space, which defines the range of addresses the program can access. Every program has its own virtual

address space. Contrary, the *physical* address space corresponds to real addresses accessible in the main memory. The Memory Management Unit (MMU) is the memory system component mapping virtual addresses to physical addresses. MMU does the translations at the page granularity. That way, it maps virtual pages to physical pages, as illustrated in Figure 2.3. As the main memory size is limited, all the virtual pages are not mapped to physical ones. When a page fault occurs, i.e., a requested virtual address is not present in the main memory, the OS updates the mapping in the MMU, replacing one physical page from the main memory.

This abstraction has multiple benefits.

1. It isolates each program from the other, ensuring better security. For instance, malware cannot use another virtual address space. Additionally, the OS randomizes the mapping to prevent malware attacks.
2. It prevents programs from dealing with shared memory issues between programs. The OS manages such shared memory portions.
3. Contrary to the size of the physical address space, which corresponds to the size of the main memory, the OS can create an infinite number of virtual address spaces.

The MMU stores all *translation* data used to translate virtual addresses in the main memory. However, to avoid accessing the main memory for every memory operation, the MMU contains a Translation Lookaside Buffer (TLB). The TLB can cache a limited number of translation data close to the processing unit. Thus, each processing unit has a TLB to process the address translations. A *TLB miss* occurs when an address translation data is missing in the TLB. In this case, the translation data is requested to the Page Walk Unit (PWU), adding an extra translation delay. Finally, a request is sent to the memory system if the translation data is not present in the PWU.

2.2.2 Caches Associativity

The cache memories are the highest levels of the memory hierarchy before the processing unit registers. They usually use the SRAM technology allowing quick read and write operations. The caches contain parts of the main memory data. When requesting an address to a cache, the first step is to check if the address is either present or not in the cache, i.e., if there is either a cache hit or a cache miss. This process can be very long and energy-consuming. Thus, to mitigate this cost, *set-associative* cache limits an address's possible locations in the cache. That way, only the possible locations

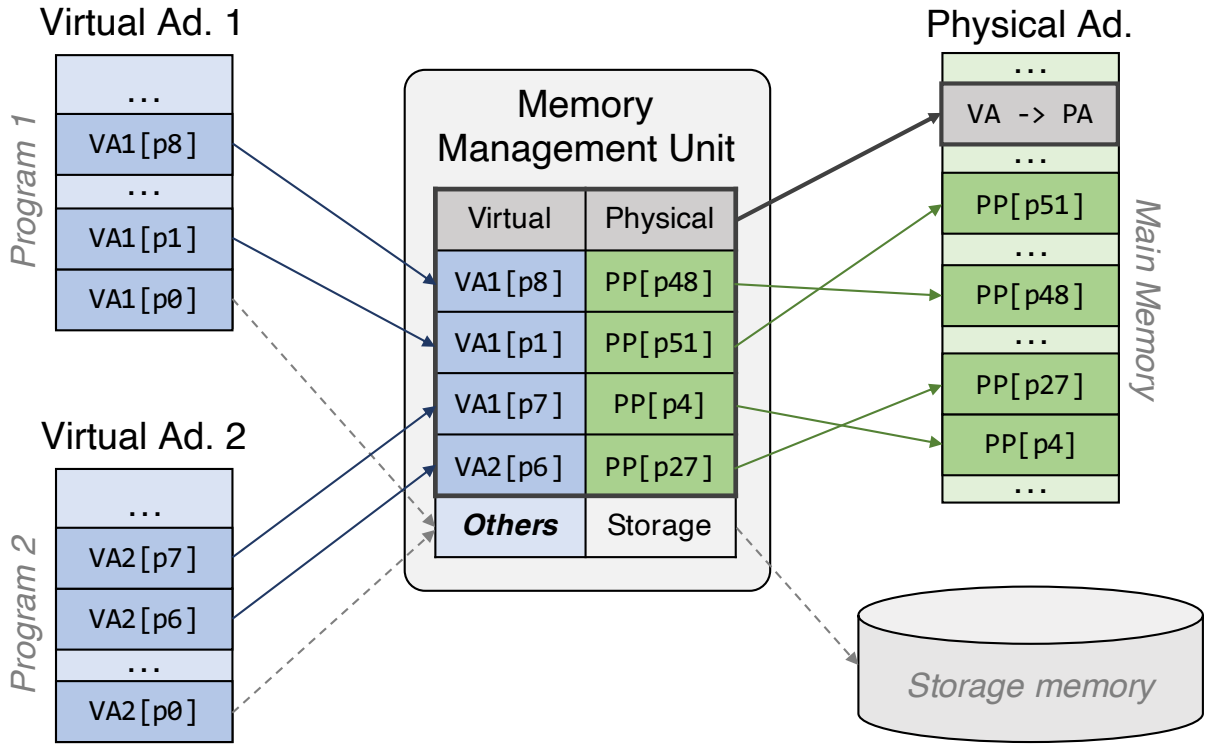

Figure 2.3: *Virtual to physical address space mapping with memeory management unit.*

are checked instead of checking the entire cache, limiting the time and energy cost. A direct-mapping cache limits an address to only one location, allowing the lowest time and energy cost but increasing the cache miss rate. Conversely, a fully-associative cache does not limit any address's location. The cost is very high in this case, with a low cache miss rate. Finally, N-way set-associative cache limits the number of possible locations to N. Depending on the size, the reasonable value of N allows a trade-off between low cost and a low miss rate.

Figure 2.4 illustrates the case of a 4-way set-associative 32KB cache. This organization corresponds to the L1 data cache of one Cortex-A53 [24] present in the MediaTek Helio X20 SoC introduced in the previous Section 2.1.3. The addresses are indexed in the cache using a *tag*, in four arrays, i.e., four ways. The size of one way corresponds to the size of the cache divided by the number of ways. In this case, the size of one way is eight kilobytes corresponding to 128 cache lines.

When requesting an address to the cache, the address is first translated into a tag and an index. The six first Last Significant Bits (LSB) correspond to the cache line offset. Then, the seven next bits constitute the index. Finally, the rest of the address represents the tag. To check if the address is present, we check within the four ways at the corresponding index if one of the four tags matches the requested address tag. If one tag matches, there is a cache hit. The last step is to load/write the data from/to the corresponding way.

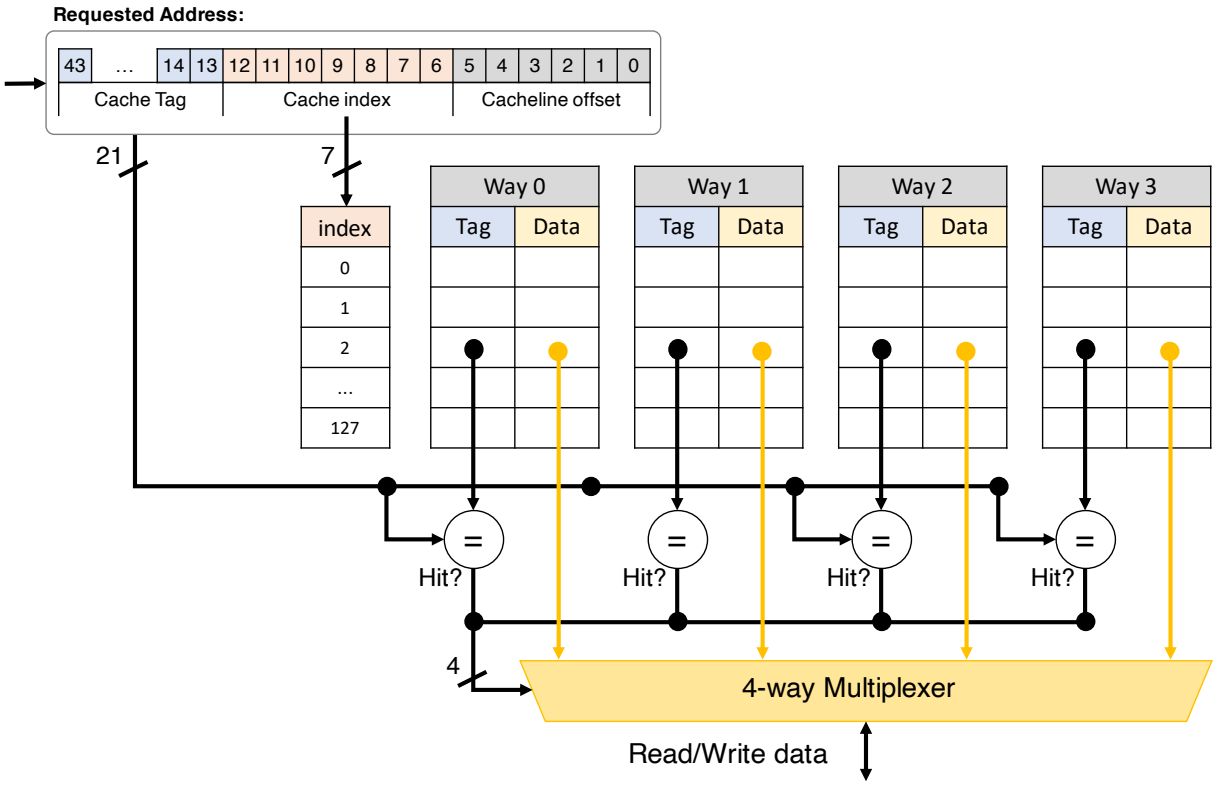

Figure 2.4: *4-way associative cache organization.*

2.2.3 Main Memory Organization

The main memory can implement several organizations [25, 26]. However, the SDRAM organization remains the standard organization using JEDEC interface protocols, such as DDR4 or LPDDR4. Some interfaces, such as HBM [27] or HMC [28] also propose to use 3D stacking implementation.

The main memory organization is divided into multiple channels [29]. Each channel is driven by a dedicated DRAM controller, as illustrated in Figure 2.5. A single channel can contain multiple ranks [30]. Usually, a Dual In-line Memory Module (DIMM) includes one rank in each face. A rank contains multiple DRAM memory devices depending on the device output width. For example, according to the JEDEC standard, a typical rank output width is 64 bits. When using DRAM devices of 8-bit output, a rank is composed of 8 devices. All devices of the same rank execute the same commands from the controller in parallel. Then, the outputs of the devices are concatenated to form the 64-bit rank output.

Furthermore, a DRAM device contains multiple banks. A bank is an array of DRAM cells where one bit is stored as a charge in a capacitor. To read data from a bank, a whole bank row is moved to a row buffer. Then, the right column is read or written using the I/O gatting. The row buffer acts as a cache. Thus, consecutive

memory accesses to the same bank row will be faster than to different banks. A *row buffer hit* occurs when memory access uses a row already present in the row buffer. Conversely, a row buffer *conflict* occurs when memory access requests a different row than the opening one. As the transfer from the bank to the row buffer removes the row data from the bank (i.e., destructive read), we need to write back the opening row to bank, and then transfer the requested row to row buffer to be read. This process is the worst-case scenario.

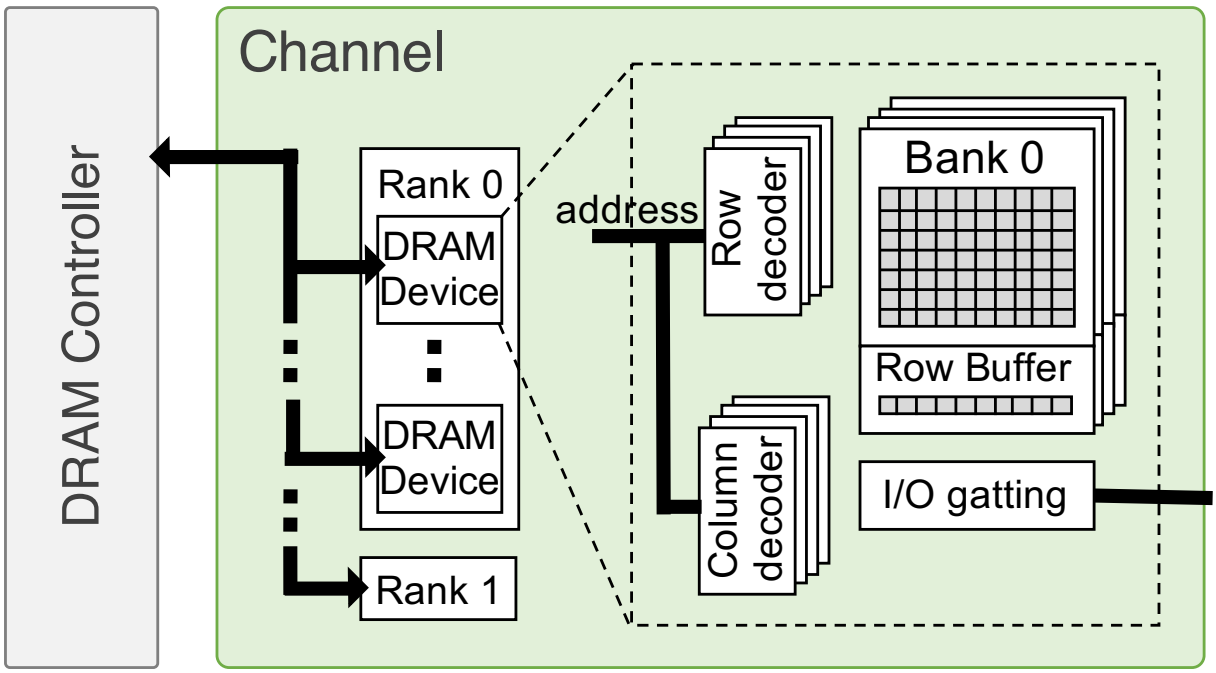

Figure 2.5: *A DRAM channel architecture.*

2.2.4 Data Prefetching

The data prefetching engine is another important component of the memory system. Its role is to monitor the accesses made by the processing unit in order to predict the future necessary addresses. Once the prediction is made, the predicted addresses are prefetched from the lower levels of the memory hierarchy directly to the caches. If the prediction is accurate, data prefetching reduces the cache miss rates and minimizes memory access time, consequently increasing the system performance [31, 32, 33, 34, 35]. Thus, a *prefetch* is a memory request generated by the data prefetching engine.

The data prefetching engine is characterized by two metrics: coverage and accuracy. The coverage defines the number of avoided misses thanks to data prefetching. The accuracy corresponds to the number of useful prefetches, i.e., prefetches that are the

source of cache hits. The coverage and accuracy depend on the prediction process and parameters, such as the number of prefetches generated from one prediction. Ideally, a perfect data prefetcher would have high coverage and high accuracy. However, in reality, a trade-off remains between those two metrics.

For instance, we use only the high confidence degree predictions to get high accuracy but limiting the prefetcher's coverage. Conversely, an aggressive data prefetcher achieves high coverage based on low-quality predictions. However, it pollutes the caches with useless cache lines, degrading the accuracy. Similarly, the number of prefetches generated from one prediction can impact those metrics. For example, the ARM Cortex-A53 exposes several data prefetcher configurations [36] to the firmware. Several parameters, such as the number of outstanding prefetches (from one to eight) or the number of consecutive cache misses triggering data prefetching (between 3 or 4), can also be tuned to obtain different coverage and accuracy values.

2.3 Computer Architecture Simulators

In this section, we introduce two computer architecture simulators, the gem5 simulator and Ramulator, both event-driven open-source simulators. The gem5 simulator is a modular platform that can model a complete computer architecture. Contrary, Ramulator only simulates the main memory allowing faster simulations by abstracting the CPU cores with memory traces.

2.3.1 Gem5

The gem5 simulator comes from merging the M5 [37] and the GEMS [38] simulators. This simulator is very popular in the architecture research community as it has already been cited more than five thousand times. gem5 is a very modular platform that can simulate various ISAs [39]. i.e., Alpha, ARM, SPARC, MIPS, POWER, RISC-V, and x86. It provides a wide range of modules for every element of the architecture. A drawback of its great modularity and accuracy is its long simulation time. Thus, for the same architecture component, gem5 modules provide different levels of accuracy. For instance, a one-core CPU simulation does not need a high-accurate interconnect model, as a single core cannot generate cannot generate enough requests to saturate the interconnect. Conversely, an inaccurate interconnect model can significantly impact the simulation results in the case of multi-core simulation.

One very accurate model available on gem5 is the High Performance In-order (HPI) CPU model [40]. This model is based on the `MinorCPU` in-order CPU model, which has been tuned by ARM to be representative of a modern ARM in-order 64-bit CPU. This model also includes cache modules and TLB modules, which have also been tuned. Other equivalent computer architecture simulators exist [41, 42, 43, 44, 45, 46]. However, we use gem5 and its HPI model as simulation reference for the rest of this thesis.

2.3.2 Ramulator

Ramulator is an accurate DRAM simulator [30] from SAFARI Research Group at ETH Zurich and Carnegie Mellon University. It supports various DRAM commercial standards such as DDR4, LPDDR4, GDDR5, or WIO2. A few corresponding parameters are available for each standard and can be set up using pre-listed values in the simulator code. Contrary to the high modular gem5 simulator, Ramulator is a memory-dedicated simulator [47, 48, 49, 50, 51]. In this way, it can provide precise and fast main memory simulations. We use it as a DRAM simulator reference in this thesis.

The standard usage mode of Ramulator is the memory trace-driven mode. Ramulator uses memory traces from an input file and simulates the sub-system DRAM controllers plus DRAM memories. Other modes, e.g., the CPU trace-driven mode, which includes a simple CPU model, are also available. Additionally, the gem5 driven mode allows using Ramulator as a module of the gem5 simulator. Thus, gem5 simulates the complete computer architecture but the main memory, i.e., the DRAM controllers and DRAM memories, which Ramulator simulates. For that, Ramulator includes a gem5 wrapper module that links both simulators. Ramulator receives the memory requests from gem5, simulates main memory responses, and sends them back to gem5. This co-simulation of gem5 and Ramulator allows a cycle-accurate simulation of a multicore system including the main memory.

III

Related Work

Computer architecture simulators have become an essential tool for architecture researchers [6, 7, 8, 9]. They are used to evaluate new architecture ideas and avoid long and expensive manufacturing processes. That way, their reliability is essential for the community. Indeed, we need to guarantee the relevance of our results. Some work already points out that inappropriate use of those simulators may lead to misleading conclusions [11, 6, 52]. Thus, this chapter provides an overview of related work on simulator error sources and mitigation approaches.

3.1 Sources of Error in Simulator Baselines

As discussed in Section 2.3, we use simulators to evaluate novel architectures against a state-of-the-art baseline model. This baseline must represent a real state-of-the-art architecture called the *target* architecture. Thus for a workload, the simulation error is the difference in the metric of interest (e.g., workload execution time) between the execution on the target architecture and the simulation model. There are two sources of error in computer simulations: modeling and parametrization errors.

Computer architecture simulators provide many parameters to configure the simulation model properly. Thus, we tune the simulator parameters to get a model closest to the target architecture. For instance, we set up parameters such as the associativity or size of the L1 data cache. However, due to the lack of information about commercial architectures [7], the simulator may not be appropriately tuned. Consequently, while comparing results from the target architecture and the simulation model, part of the error comes from the fact that the simulator is configured with the wrong parameters. Note that the same simulator can have be more accurate with an adequate parameter configuration. We define this component of the total simulation error as parametrization error.

We define modeling error as the remaining error after removing the parametrization error with an adequate configuration. The modeling error comes from the simulator model accuracy, independently of the parameters. For instance, a simulator can abstract away processes happening in the real architectures, such as main memory bank conflicts (see Section 2.2.3). Thus, even with all the necessary technical information, i.e., removing the parametrization error, it would not be possible to faithfully represent the real target main memory. In this case, we can only reduce the error by extending the simulator with more detailed models (e.g., adding a component that models bank conflicts).

Interestingly, some of the available simulators, such as gem5 and Ramulator, already include very complete and detailed parametric models to cover a wide spectrum of architectures. Hence, in this thesis, we focus on reducing the parametric error as a first necessary step to achieve accurate simulations.

3.2 Simulator Verification

In order to verify the accuracy of simulators, several works evaluate the simulation errors against target architectures [53, 54, 55, 56, 30]. They configure the simulators to get a representative simulation model of the target architecture, i.e., reduce the parametrization error. Then, they use realistic benchmark suites such as SPEC CPU2006 [57] or PARSEC [58] to execute on the target architectures and calibrated simulation model. Different metrics are used to compare the execution and simulation results. Assuming that the simulation model is ideally calibrated, only the modeling error remains. If this one is low enough, the simulator is verified.

3.2.1 gem5 Verification

Previous works propose a verification of the gem5 simulator [10]. Endo et al. [12] uses as target architecture the in-order Cortex-A8 [59] and the Out-of-order Cortex-A9 [60]. They compare the execution time of benchmarks from the PARSEC suite against execution on the real target architecture (between 7% and 17% error). Butko et al. [13] use the same target, i.e., the Cortex-A9, but also the Cortex-A7 [61] and the Cortex-A15 [62] present in the ARM big.LITTLE [23] architectures [63]. They measure a mismatch between 1.4% and 17.9% with different benchmarks. In the same way, Gutierrez et al. [14] use the same ARM target architectures but include an OS in the simulation. In order to reduce the error, they disable on both the target and the simulation some components that are not appropriately modeled (e.g., the data prefecher). They finally get an error between 5% and 17% depending on the benchmarks. Akram et al. propose a verification of an X86 architecture [64]. They use the *perf* tool [65] to monitor the target architecture behavior and faithfully configure the simulator (136% remaining error). Then, they reduce the modeling error by modifying the simulator code. For instance, they remap some micro-operations to other functional units. Like that, they reduce the error from 136% to 6%.

3.2.2 X86 Simulator Error Comparison

Akram et al. [8] instead of validating simulators, provide a cross-comparison of multiple simulators: gem5 [10], Multi2sim [45], MARSS×86 [42], PTLsim [66], Sniper [41], and ZSim [43]. After detailing simulator features, they select four of them for a detailed study. They first calibrate them with the same target, Intel's Haswell architecture (core i7-4770). Then, they compare the execution and simulation of SPEC CPU2006 [57] and MiBench [67] realistic benchmark suites. The results show an error of 9.5%, 38.2%, 44.6%, and 47.6% for respectively Sniper [41], PTLsim [66], gem5 [10], and Multi2sim [45]. They use *PAPI* [68] to monitor internal target metrics such as the number of L1 data cache misses and compare them to the simulator results. That way, they expose which parts of the simulators need to be improved, e.g., the branch predictor model. Finally, they conclude that the accuracy of the simulators can significantly change depending on the target architecture. Thus, a verification with one target cannot prove the full simulator's accuracy.

3.3 Dedicated Simulators

The architecture coverage of a simulator defines the range of architecture simulations it can perform. Ideally, a simulator must have the most extensive architecture coverage. However, significant coverage results in a complex parametrizable simulator and very long simulation times. Thus, simulators can limit their coverage to provide accurate simulations while having reasonable simulation times and simulator parametrizable complexity. In this case, we call them dedicated simulators.

The architecture coverage can be described using three dimensions:

- Architecture component. It corresponds to elements of the architecture that the simulator models. For instance, the simulator can model the whole architecture of just one component, such as the main memory.
- Architecture variance. It describes the variations of the same component that the simulator can model. For instance, a CPU can have an x86, RISC-V, alpha, or ARM architecture.
- Simulation metric. It represents the different metrics measured during the simulation. For instance, simulators only measure performance metrics such as IPC while others focus also on energy metrics.

Thus, dedicated simulators limit their architecture coverage. That way, more effort is spent modeling specific parts or features characteristic of the dedicated architectures. Also, as the choice of target architectures is limited, the default baseline model calibration is more likely to fit another target architecture instance that belongs to that restricted domain. Hence, this allows reducing the parametrization error.

3.3.1 Component-Specific Simulators

The first example of component-specific simulators is the GPU simulators [69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81]. They limit the simulation to one part of the architecture, the GPU, which already includes many processing units and memories. Some simulators, such as ATTILA [76], are very modular, providing a high architecture variance and simulation output. Others like [73, 74, 75], focus on energy consumption. Thus, PowerRed [74] provides details GPU power simulations, including an interconnect power modeling.

Another kind of component-specific simulator focus on memory parts of the architecture [30, 82, 48, 47, 83, 84, 85, 86, 87, 51, 49, 50, 88, 89]. Those simulators allow detailed simulations of the multiple memory components. For instance, Tavakkol et al. propose MQsim [82], an SSD simulator, which they evaluate against four real SSDs (less than 18% error). Kim et al. propose Ramulator [30], which provides highly accurate DRAM models. They verify it by comparing it against RTL simulations of DDR3 commercial Verilog models. Then, they show that it performs 2.5x/3.0x speedup compared to the next fastest simulator [51].

Additionally, we give more examples of component-specific simulators. For instance, accelerator simulator [90, 91, 92], e.g., deep neural network simulations. Or, Network-on-Chip (NoC) simulators [93, 44, 94, 95, 96].

3.3.2 ISA-Specific Simulators

Some simulators limit their architecture coverage by fixing the Instruction-Set Architecture (ISA). For instance, x86 architecture simulators [66, 42, 41]. By limiting the ISA, more effort is spent on a dedicated architecture providing better accuracy reducing modeling error. Also, as the choice of target architectures is limited, the default baseline model calibration is more likely to fit another target architecture that belongs to the simulator architecture coverage. Thus the Sniper simulator [41], which is an x86 architecture simulator, provides better accuracy than the modular gem5 simulator on

this particular architecture, as illustrated in Section 3.2.2. In the same way, Bruschi et al. introduce GVSoC [97] dedicated event-driven simulator for RISC-V architectures which is 15% more accurate than gem5.

3.3.3 Metric-Specific Simulators

Dedicated simulators can also limit the architecture coverage by focusing on specific simulation output, such as power consumption. [98, 99, 100, 101, 102, 103] Power consumption is a crucial element of modern architecture. Thus, simulators such as McPAT [98] or eSimu [100] are used to precisely estimate the power consumption of computer architectures. Those simulators also need to be calibrated with target architectures. Thus, Lee et al. propose PowerTrain [101] a McPAT calibration against an ARM Cortex-A15 present in a Samsung SoC, the Exynos 5422. Finally, simulators can be dedicated to other metric such as the system security. Forcioli et al. present a framework based on gem5 to evaluate system security at the architecture level [104].

3.4 Simulation Time Mitigation Techniques

The simulation time remains a problem in computer architecture simulation. Thus, this section introduces an overview of some mitigation techniques used to reduce the simulation time. Research still needs to evaluate the necessary accuracy not to simulate unnecessary processes that may slow down the simulation time.

3.4.1 Workload Sampling

A fundamental problem with computer architecture simulation is that it is difficult to parallelize in host machines. Indeed, the simulation respects time order execution and intricate dependencies between components. One solution is to sample the simulation, e.g., using a checkpointing method [105], and run the samples in parallel [106]. In order to reduce the noise caused by cold starts, a warmup is added to each sample. Additionally, due to redundant sample behaviors, works [107, 108] propose to select a subset of the samples that already significantly represents the benchmark behavior. For instance, the SimPoint method [108] proposes to use the K-means algorithm and Basic Block Vectors (BBV) to analyze samples and select this kind of subset. The evaluation of the SimPoint methodology shows an extra error of 3%.

3.4.2 Trace-driven Simulator

Another common way to mitigate simulation time is to use traces as input for the simulators, i.e., trace-driven simulators. The traces are prerecorded fixed inputs. For instance, traces can be recorded from memory request addresses. Then, memory trace-driven simulators can directly replay the traces without simulating processing units [30, 82, 48, 47]. Some traces can be more sophisticated and include dependencies such as the elastic traces [109, 110, 111] reproducing an out-of-order CPU execution. However, trace inputs are still different from realistic execution, which includes many dependencies in the program execution flow.

3.4.3 FPGA-Accelerated Simulation

Field-Programmable Gate Array (FPGA) allows fast evaluation of new designs. Thus, several works propose to use them to accelerate the simulation of architecture parts [112, 96, 93, 113, 114, 115, 116, 117]. For instance, Papamichael et al, propose FIST [93] which uses FPGA to emulate NoC designs. That way, they reduce the simulation by 3 to 4 orders of magnitude speedup against software-based NoC simulators. Similarly, simulators propose to simulate complete multicore architecture such as ProtoFlex [114] which achieves an average speedup of 38x compared to Simics [46] software-based simulator. Despite their promising accelerations, FPGA-accelerated simulators have only been successfully employed by those who designed them. They lack the user-friendliness of software simulators. Most previous works focused on efficiently mapping more of the target to a single FPGA. Unfortunately, the resulting multithreaded models became more challenging to implement than the architectures they model, significantly undermining their usability. Some FPGA-based simulators like FireSim [112] mitigate this problem by running full-system simulations on could FPGAs, providing a more user-friendly interface equivalent to software-base simulators.

3.4.4 High-Level Simulation

Multiple levels of abstraction exist [118] to model computer architectures. The first is the digital abstraction that interprets the analog signals as digital. The next one is the cycle of abstraction. In this case, the time is no longer continuous but counted using clock cycles. This abstraction is mainly in the computer architecture domain with cycle-accurate simulators, e.g., the gem5 simulator [10] or Ramulator [30]. The

last example of a simulation time mitigation technique is to use a higher abstraction in architecture models [55, 119, 120]. For instance, Genbrugge et al. propose interval simulations [119] instead of cycle-accurate simulations. Analytic models describe intervals between architecture events, such as branch prediction misses or cache misses. By increasing the level of abstraction, they reduce both the development time and the simulation time. Thus, they show a reduction of the simulation time of one order of magnitude for an average extra simulation error of 4.6%. In the same way, many simulators [10, 41, 120, 109] propose different levels of accuracy depending on the experiment's focus to reduce the simulation time. For instance, the gem5 simulator provides multiple in-order CPU models, from the SimpleCPU model, a purely functional in-order CPU model, to the very accurate HPI model described in section 2.3.

3.5 Remaining Challenges

This chapter illustrates that simulators are widely used in the architecture research community. Due to the remaining tradeoff between accuracy and simulation time, simulators offer various modeling approaches, e.g., dedicated simulators or simulation time mitigation techniques. Some of the approaches can mitigate the modeling error or, conversely, increase it while reducing the simulation time. Thus, they provide different points on the accuracy versus simulation time Pareto. However, the parametrization error is common to all those simulators and needs to be mitigated following calibration methodologies.

Previous works provide calibrate methods to reduce parametrization error to measure the modeling error against real state-of-the-art architectures. However, those calibrations follow empirical methods, which cover a few targets of the whole simulator architecture coverage and cannot be generalized to others. Moreover, the continuing advancement of state-of-the-art commercial architectures pushes extending the architecture coverage of the simulators, making previous empirical calibration methods outdated.

One presumption with evaluation using computer simulations is that simulators do not need precise calibrations against target architectures as long as the simulation baseline trend is realistic. For instance, multiple memory exploration works [121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 5, 4] use simulators with non-calibrated baseline models to evaluate new memory designs. However, as pointed out by some works [11, 6, 7], this may lead to misleading conclusions, making the simulator calibration a key element of the evaluation.

Consequently, the calibration of the computer architecture simulators remains an important challenge for the community. It remains essential from the verification of simulators to their utilization during new design evaluations. Thus, this thesis proposes a systematic methodology to calibrate computer architecture simulators.

A critical limitation of architecture simulator calibration is the lake of technical information about commercial architectures. Several works [136, 56, 14, 7, 137] propose to use hardware performance monitoring on the target architecture to extract missing technical information necessary for the baseline model calibration. Alves et al. propose dedicated microbenchmarks to evaluate and calibrate specific parts of the SiN-UCA simulator [56]. Other works [138, 139, 140, 140, 139, 138] related to security vulnerabilities propose to use microbenchmarks to discover hidden technical information to exploit the target vulnerability. However, all the proposed methods remain empiric and cannot easily be extended to other architectures. Open-source generic microbenchmarks have been developed to extract technical information from commercial architectures, such as LMbench suite [141] or STREAM [142]. However, they are not part of a complete methodology allowing a direct simulator calibration.

Instead, our methodology describes a complete workflow, including simulator parameters analysis and the detailed design of microbenchmarks used to reveal missing technical information from the target architecture.

3.6 Summary

To summarize, this chapter introduces several simulators and how they are verified. Thus, the popular gem5 [10] simulator shows such an average simulation error against ARM commercial architecture under 20%. However, comparison [8] between several simulators [10, 45, 42, 66, 41, 43] shows that the simulation error depends on the targe architecture. In this case, an x86 architecture. Thus, the Sniper simulator [41] shows an average simulation of 9.5%, contrary to the gem5 simulator with 44.6%.

We further see different kinds of simulators. The component-specific simulators allow faster simulation abstraction parts of the architecture. For instance, Ramulator [30] allows faster accurate memory simulation (2.5x/3.0x speedup compared to the next fastest simulator [51]) by abstracting the CPU with memory traces. ISA-specific simulators allow more accurate simulations, as illustrated in the previous paragraph with Sniper [41]. Finally, metric-specific simulators increase simulation accuracy by targeting specific output metrics.

As the simulation time remains an important limitation for simulators, this chapter introduces simulation time mitigation techniques. Thus, workload sampling with the SimPoint method [108] allows a reduction in simulation time and parallelizes it for an average extra error of 3%. Trace-driven simulator abstract processing units by pre-recorded trace files, e.g., illustrated in the previous paragraph with Ramulator. FPGA-accelerated simulations offer good perspectives but are still not modular enough to be widely used. Finally, high-level simulations use higher abstraction-level models to reduce the simulation time. Thus, interval simulations provide speedup the simulation by one order of magnitude for only an average extra error of 4.6%.

As a result, we see that simulator calibration remains a significant problem for all kinds of simulators. The calibration approaches proposed in previous works follow an ad hoc method making them not extendable to other architectures. Hence, this thesis contributes by proposing a systematic calibration methodology.

IV

Microbenchmark-Based Timing Calibration Methodology

In this chapter, we propose a methodology to calibrate the memory system of a computer architecture simulation. The methodology is composed of two phases. The first phase consists of determining the different simulator parameters in need of calibration. In the second phase, we then properly calibrate those parameters. To this end, we start by using information from first-party documentation. Then, we design handcrafted microbenchmarks and use hardware performance counters to extract the missing parameter values from the real target state-of-the-art commercial platform.

4.1 Motivation

Computer system simulators [10, 42, 82, 30, 41, 43, 45, 47] are widely used by researchers. They allow quick evaluations of new ideas avoiding long expensive manufacturing processes. Those new ideas are evaluated with respect to state-of-the-art baseline architectures. The choice of a baseline depends on the kind of architecture we target, e.g., low-power mobile architecture or high-performance server architecture. The relevance of the simulated results is directly related to the quality and the choice of that baseline. The use of inaccurate baselines can add unbounded noise to the experimental methodology and lead to erroneous conclusions. Unfortunately, simulation models calibrated with real architectures are rarely available to the research community.

4.1.1 Memory System Modeling

The memory system plays a key role in all instruction-processor based compute platforms [1]. A slow data access time directly impacts the instruction execution flow and reduces the whole system performance. This statement is even more true with modern partly heterogeneous multicore architectures which contain many components like cores, GPUs or programmable accelerators that compute data at Gigahertz frequencies. Thus, the data movement in the multiple levels of the memory hierarchy needs to be fast with low access time but also to provide high bandwidth.

Improving the memory system is not an easy task to do due to its high complexity, which includes different components, memory technologies, organizations and access protocols. For example, some new emerging non-volatile memory technologies seem a promising alternative to reduce memory system leakage energy, providing the same or better level of performance [2, 143, 3]. However, those technologies have different technical specifications than the usual memory technologies implemented. This means that

a straightforward replacement is not efficient [144, 145, 146]. Hence, researchers need a complete understanding of all the processes/sub-processes running internally in the memory system. Faithful reference models incorporating all the workload-dependent effects are important for relevant improvement/optimization of the memory system.

4.1.2 Running Example

To illustrate our methodology's different elements, features, and processes, we use a simple running example. We consider a very simple fictional simulator that models the interconnect, DRAM controller, and the main memory as a fixed latency. We use the Broadcom BCM2837B0 SoC present on the Raspberry Pi 3B+ development board [147] as the reference state-of-the-art architecture. This architecture comprises four Cortex-A53 with four individual L1 data caches and one shared L2 cache.

The objective of our methodology is to find the best instantiation of the simulation parameters (e.g., the fixed latency) to produce a calibrated simulation (i.e., a simulation that behaves as close as possible to the reference architecture). Obviously, the quality of the resulting simulation is limited by the level of detail in the simulation model. Nevertheless, our objective is not to question the simulation model but to find the best possible instantiation of its parameters.

4.2 Simulator Parameters Identification

In this section, we describe the first phase of the methodology. The purpose of this one is to identify the simulator parameters that we need to calibrate. This process can be very different depending on the kind of simulator we use in the instantiation. The simulator can either model a single memory component (e.g., Ramulator [30]) or a complete architecture (e.g., gem5 [10]). Also, they can be modifiable, modular, or configurable by the user. Even with the same kind of simulator, the parameters may change accordingly to the modeling approach implemented in the simulator. For instance, two different simulators could have different parameters for modeling the same component. Also, due to the complexity of modeling all the processes of real architecture, some processes are modeled in a simpler way or are not modeled at all. The parameters of these simpler models cannot directly be related to a specific architecture feature. For example, we can model the main memory as a simple component with fixed bandwidth and latency. However, as we describe in Section 2.2.3, the main memory is a complex component, including many processes, such as bank conflicts and a

complex scheduling protocol. In this case, a fixed latency has no realistic meaning, but its value may be extrapolated to best match reality.

Frequently, simulators provide different models with different levels of accuracy for the same component. As the trade-off between simulation time and accuracy remains, researchers must properly choose the right ratio to get relevant results with a reasonable simulation time. From the previous example, in case of very low main memory activity during the simulation (e.g., simulating only one core executing a cache-friendly application), using a very accurate main memory model is not relevant. Hence, it does not make sense to needlessly extend the simulation time due to the very low impact of the main memory on key simulation metrics. This example is further detailed in Section 5.4.2.

4.2.1 Generic Parameter Template

To provide flexibility and a high degree of accuracy, simulators have become complex, including hundreds of parameters. Thus, we need to identify the main parameters and understand how they define the behavior of the simulated memory system. For that, we determine the path that a memory request follows inside the simulator memory system from the execution of a memory instruction to the last level of the memory hierarchy.

As explained in Section 2.1, requests are first generated by the load/store unit and sent to the TLB to be translated from the virtual to the physical address space. Then, the request travels through the different cache levels and finally goes to the main memory. In parallel to the load/store unit, the data prefetcher can generate requests based on its intern prediction process and send them to one of the cache levels. We divided this path into six blocks as illustrated in Figure 4.1. For each block, we list the generic parameters we have to identify in the simulator. For instance, what is (are) the simulator parameter(s) that control the number of outstanding requests the load/store unit can generate? Thus, we need to map each generic parameter in Figure 4.1 to real simulator parameters.

It is important to notice that a straightforward translation is not possible depending on the model's accuracy, as illustrated with the running example. In this case, the last three blocks of the data path, i.e., the Interconnect, the Memory Controller, and the Main Memory, are modeled as a fixed simple latency. This way, all the generic parameters present in those blocks are mapped to one basic simulator parameter. Also, some frameworks, such as Ramulator or MQsim, only simulate a portion of the memory sys-

tem, not the full computer architecture. Consequently, we need to limit the template to the respective blocks. At the end of this process, we have a complete mapping of the simulator parameters to the generic parameters listed in Figure 4.1. The rest of the methodology consists of finding the correct value for each simulator parameter.

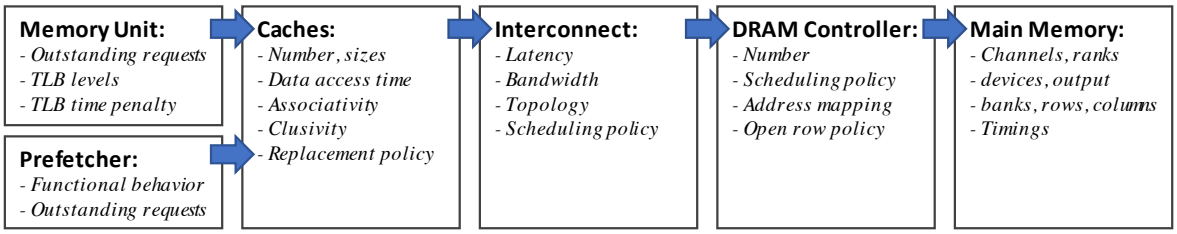

Figure 4.1: *Memory system component blocks with corresponding generic parameters.*

4.2.2 Running Example Simulator

With the running example, our fictional simulator is quite simple. It models the interconnect, the DRAM controller, and the main memory as a single element. This one returns the requests with fixed latency and an infinite bandwidth. For that, the simulator contains only one parameter that we call the `memory latency`. As there is only one parameter, the first phase of the methodology is straightforward. We use only three blocks from the template, i.e., the interconnect, the DRAM controller, and the main memory block. The only simulator parameter `memory latency` is selected and mapped to the two generic parameters: the interconnect latency and the main memory timings. As we use a very simple model, many generic parameters are not mapped. Consequently, we need to extrapolate the value of the `memory latency` to best match the reference architecture.

4.3 Simulator Parameter Discovery

To find appropriate values for the selected parameters, we start by looking at public first-party documentation. Unfortunately, not all information is available in the first-party documentation. For instance, the Raspberry Pi 3B+ documentation [147] indicates that it has a 1GB LPDDR2 SRAM main memory. We can deduce from it the main memory timings as LPDDR2 is a standard protocol. However, the interconnect latency is still not documented. Consequently, we cannot find the right `memory latency` value for our simulator.

Hence, to discover the undisclosed parameter values, we propose in the second phase of the methodology to execute handcrafted microbenchmarks on the real hardware. The key process of this phase is to properly design the microbenchmark to stress specific features of the memory system. Then, we monitor architecture events using hardware performance counters to expose memory behavior. We finally deduce the parameter values from the resulting memory behavior.

4.3.1 Architecture Event Monitoring

The hardware performance counters are specific registers implemented in modern architectures. They can count architecture events such as accesses/misses to the L1 data cache, executed instructions, and branch predictions. The list of available events and the number of performance counters may change from one architecture to another. For instance, the Cortex-A53 contains six performance counters and fifty-nine architecture events [148].

Before monitoring, we need to assign one event to each performance counter. Then, we start monitoring event occurrences. Contrary to software profiling, the hardware performance counters allow non-intrusive monitoring. The performance counters can also be used to indirectly measure the execution time by fixing the clock frequency and counting the number of CPU execution cycles. However, we can alternatively use OS libraries, e.g., the `linux time` library, to measure execution time as the number of hardware performance counters is limited.

4.3.2 Inputs and Scenarios

In order to calibrate the parameters of the list established in Section 4.2, we first need to establish a strategic calibration order, as some dependencies may exist between parameters. For instance, if we want to monitor the access time to the L2 cache, we can generate a request to the L2 and measure its traveling time. However, this time is the addition of the L1D access time and the L2 access time. Consequently, our strategy would be to determine the L1D access time first and then the L2 access time.

Once all the dependencies are identified and the calibration order established, we start designing the microbenchmark. The microbenchmark executes purposeful memory request sequences to stress specific components of the memory system. At the same time, it uses hardware performance counters to monitor the memory component behavior. For each memory sequence, we need to select the suitable architecture events

to monitor in order to expose the targeted memory component behavior when memory requests travel through the memory hierarchy. Then, we determine the possible scenarios generated by one memory sequence.

We define a scenario as a memory system behavior resulting from the memory sequence execution. We describe each scenario as a set of conditions depending on the component we target in the memory system. Thus, we use the performance counters to expose the scenario conditions during sequence execution. Finally, we use the scenarios to deduce parameter values. For that, we differentiate between two cases:

- One target scenario. In this case, the purpose is to generate a desired memory behavior. Then, we use the measures to deduce the parameter value. For instance, we generate a request to the L1D cache. We use the performance counters to verify that we have the expected behavior, i.e., the request goes to the L1D cache, and the address is not missing. Then, we use the time measurement to deduce the L1D cache access time.

- Multiple possible scenarios. Before executing the memory sequence, we do not know which scenario will happen, i.e., what will be the memory behavior. In this case, it is the resulting scenario, out of all the possible ones, that is used to deduce parameter value. For instance, we want to know the maximum level of parallelism in the L2 cache. We already know the access time to the L2 cache. We generate in parallel two requests to the L2 cache. We have two possible scenarios: the average access time is identical, or the average access is faster. We execute the sequence and see which scenario happens using the performance counters. If the average access time is identical, the L2 cache cannot handle two requests in parallel. If the average access time is faster, the L2 cache can as least handle two requests in parallel. Then, we use more sequences to deduce the exact value, more details in Section 5.3.5.

There are two reasons why a memory sequence would not generate a behavior fitting a predefined scenario. The first one is when we do not have an a priori knowledge of all the possible scenarios. Accordingly, we use the results from the performance counters to draw new insights about the memory behavior and the memory sequences to generate. The second reason is when there is noise interfering with the monitored values. For instance, the core could vary its clock frequency during the execution of the microbenchmark. In the same way, we use the results from the performance counters but this time to define the source of the noise and find a strategy to mitigate it.

4.3.3 Running Example Scenarios

With our running example, we want to calibrate the `memory latency` simulator parameter. As there is only one parameter, there is no calibration order to establish. Hence, we start directly with the design of the microbenchmark. We target a memory sequence where all the requests go to the main memory. Then, we time the execution to expose the request traveling time and deduce the parameter value. Figure 4.2 illustrates the possible scenarios. There are three conditions. The first and the second conditions correspond to cases where all requests miss or hit the L1D and the L2 cache, respectively. The last condition corresponds to whether there is a conflict or not in the main memory, e.g., a rowbuffer conflict as introduced in Section 2.2.3. As we want all the requests to go to the main memory, both scenarios have the same two first-condition values. For the last condition, the first scenario (green) represents the case where there is no conflict. In contrast, the second scenario (yellow) represents the case where there is a conflict in the main memory between the memory requests.

As we want all the requests to go to the main memory, if either the first or second condition does not match one of the two scenarios, it indicates that we have unexpected behavior. For instance, the data may be prefetched into the caches. Finally, as the simplified simulator used in this running example does not model main memory conflicts, we decide to calibrate our parameter to mimic the no-conflict scenario. It results in a best-case latency, which can be a good enough approximation in memory-friendly workloads. Consequently, we use only one request in the memory sequence. That way, we remove the case where requests could have a conflict. On Figure 4.2, only the first scenario (green) remains. Once we have verified that the sequence matches the scenario, we use the time measure to deduce the `memory latency` parameter value.

4.3.4 Microbenchmark Features

In order to create appropriate memory request sequences and reduce noise during memory system monitoring, our method to build microbenchmarks implements three important features:

1. Data pinning. The microbenchmark initializes data that is requested by the memory sequence in a particular part of the memory hierarchy. That way, it forces memory requests to access a predetermined path through the memory hierarchy. We call this data pinning. The data could be pinned in the different cache levels as well as in the main memory. We can force the location of data by controlling

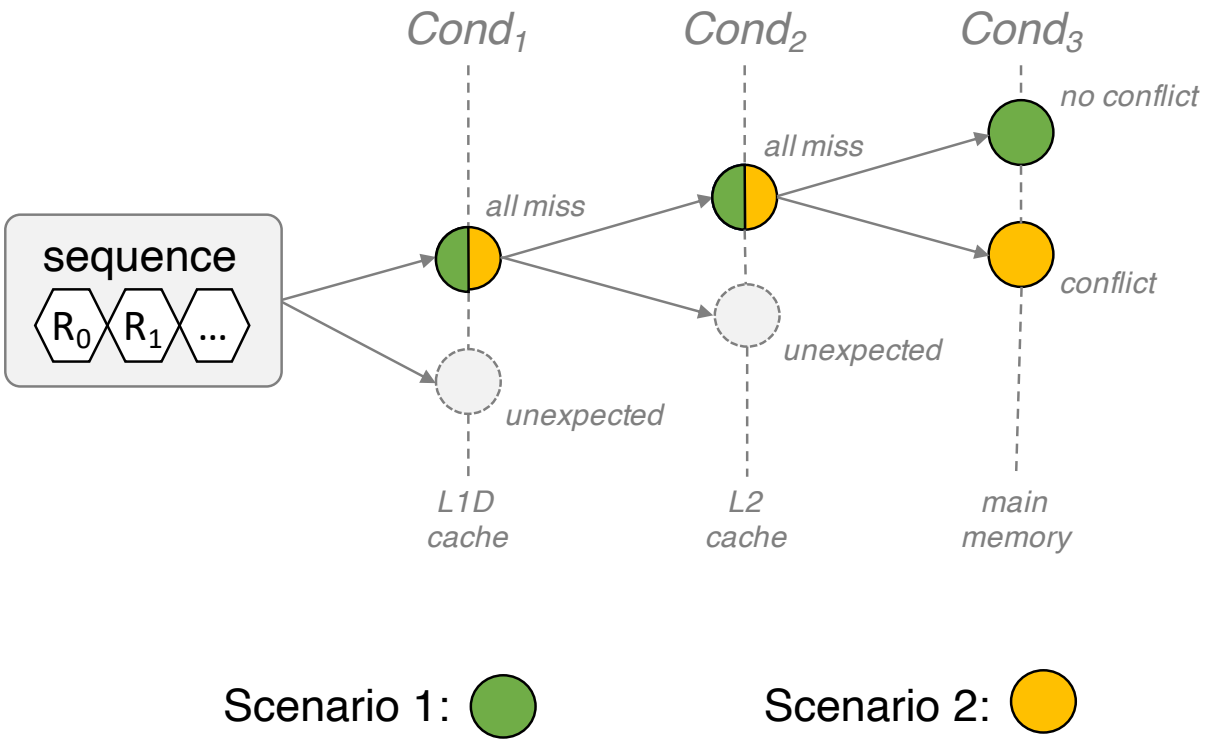

Figure 4.2: *Running example scenarios and conditions.*

the size of an array we repeatedly access with the memory sequence, e.g., if the array is small enough to fit in the cache, consecutive accesses will not miss that cache. Also, we can use dedicated cache flushing instructions (e.g., `dc civac` in ARMv8 ISA) to directly pin the data in the main memory. The data pinning should also consider the TLB. We can play with the gap between requests to use more or fewer pages with the same number of loads to pin the address translation information in a target level of the TLB. When pinning data in the main memory, it is also useful to query the OS to know the physical addresses via system calls (e.g., `pagemap` in Linux). In this way, we can either target a specific portion of the main memory or deduce memory organization (e.g., address mapping of the main memory). Thus, data pinning is an important feature of the microbenchmark as it allows stressing specific elements of the memory system hierarchy.

2. Memory request dependency. Multiple accesses to the memory hierarchy can affect each other. Contention conflicts or data movement can occur and modify the path of the requests through the memory hierarchy. The microbenchmark needs to control the dependency between consecutive memory requests. For instance, we can use pointer chasing to create data dependency between memory requests. By playing with the number of pointers, we can select the number of parallel requests sent to the memory system. For instance, we can instantiate two pointers with different pointer chasings. Thus, we can load two addresses in

parallel as there is no data dependency between both pointers. However, three loads would not be possible. Also, we can use masks on either virtual or physical addresses to target a specific portion of the memory system (e.g., different main memory banks). In this way, the microbenchmark can avoid or force such conflicts, or data movements, depending on memory system feature that the microbenchmark stresses.

3. Noise minimization. There are two types of noise during microbenchmark execution. The first one is the unexpected memory activity generated by the input. For example, a memory sequence that generates a large unexpected number TLB misses will increase the measured access time. Predictable memory access patterns may trigger data prefetching and change the data location inside the memory hierarchy. To reduce this type of noise, we need to identify the source using results from the performance counters. Then, we use the new insights to correct the memory sequence in order to generate the expected scenario. The second type of noise is generated by processes other than the memory sequence execution, such as the OS routine. For instance, starting and reading the performance counters impact the number of accesses and misses in the memory hierarchy. We iterate the memory sequence execution in a measured loop to reduce this noise. That way, the overall execution is large enough to average out punctual noise and estimate execution time and events occurrences at a coarse granularity.

4.4 Implementation of the Running Example

In this section, we describe the implementation of a microbenchmark, applying our methodology to the running example.

4.4.1 PAPI library

In order to monitor a specific portion of the microbenchmark, i.e., the region of interest or the measured loop, we use the Performance Application Programming Interface (PAPI) library [68]. We first use PAPI to initialize the performance hardware counters with the designated architecture events. And then monitor the measured loop using dedicated PAPI functions. The specific PAPI functions we use for the counters initialization and the monitoring are described in Section 7.3 of the Appendix. Table 4.1 lists the different architecture events and their descriptions that we use in this instantiation. We select the event 1, 2, 3, and 4 to verify that the memory requests arrive at the main

memory. This corresponds to condition 1 and 2 in Figure 4.2. We use event 5 to verify that the effect of TLB misses is negligible compared to the number of memory requests. Finally, we use event 6 to verify that there is no data prefetching activity.

Table 4.1: *Architecture events monitored with PAPI.*

N.	Event name	Description
1	L1D_CACHE_ACCESS	Accesses to the L1 data cache
2	L1D_CACHE_REFILL	Refills in the L1 data cache due to missing data
3	L2D_CACHE_ACCESS	Accesses to the L2 from the L1 data cache
4	L2D_CACHE_REFILL	Refills in the L2 due to missing data
5	L1D_TLB_REFILL	TLB refills due to missing translation data
6	L1D_CACHE_REFILL_PREFETCH	L1 data cache refills due to data prefetching

4.4.2 C-code Implementation

As we run the microbenchmark on a CPU, the C programming language offers a good compromise to design it. It is a relatively low-level language that allows the programmer to control (to a certain degree) the instructions executed by the CPU hardware. Also, we can easily add to the C-code lines of assembly code in selected places to fully control the compiled binary. Additionally, the C-code allows a precise utilization of the address space, which is very useful for implementing the data pinning and data dependency features described in Section 4.3.4.

Figure 4.3 shows a snippet of the microbenchmark C-code designed for our running example. To generate the desired memory sequence, we access the address stored in a pointer we previously flushed from the caches. For that, we need first to initialize a pointer chasing of one element. Then, we create a pointer that points on the first element of the pointer chasing (lines 12 and 13). The use of a pointer going through a pointer chasing allows us to reduce the number of none desired operations in the measured loop. For instance, a complex access pattern can be implemented in the pointer chasing using many for-loop and conditional operations during the initialization. Then, to generate a load in the measured loop, the pointer takes the value of the address it points on (lines 21, 23, 25, 27, and 29), avoiding the overhead of executing address generation instructions in the measured loop. As we want to pin the data in the main memory, we use the flush function, described in Section 7.3 of the Appendix, to remove the corresponding data from the caches (lines 20, 22, 24, 26, and 28) before loading the same address again.

We start monitoring before executing the measured loop, stop monitoring right after, and store the monitoring results for later inspection. Section 7.3 of the Appendix

describes the specific functions from the PAPI and `time` standard library. We execute two hundred thousand iterations of the measured loop (lines 2 and 18). This number of iterations seems a good compromise between the execution time and the averaging out of the noise. To further reduce the impact of executing auxiliary instructions (e.g., loop control instructions) in the measured loop, we can unroll the measured loop. For instance, in the example code, we execute five flush and load operations sequences per measured loop iteration (from line 20 to line 29). Hence, our code generates one million load requests to main memory.

```
1  /* Number of iterations in the Measure loop */
2  #define LOOP    200000
3
4  long long int main(){
5
6      /********** Initialization **********/
7      long long int addr = 0;
8      long long int *ptr = NULL;
9
10     /* Create pointer chasing with one element */
11     addr = (long long int ) &addr;
12     ptr  = (long long int*) &addr;
13
14     papi_init();
15
16     /********** Measure LOOP **********/
17     papi_start();
18     for(long long int loop=0; loop < LOOP; loop++){
19         flush(ptr);
20         ptr = (long long int*) *ptr; // 1
21         flush(ptr);
22         ptr = (long long int*) *ptr; // 2
23         flush(ptr);
24         ptr = (long long int*) *ptr; // 3
25         flush(ptr);
26         ptr = (long long int*) *ptr; // 4
27         flush(ptr);
28         ptr = (long long int*) *ptr; // 5
29     }
30     papi_read();
31
32     return (long long int) *ptr;
33 }
```

Figure 4.3: *Running example microbenchmark C-code implementation.*

4.4.3 Assembly Code Verification

In order to verify the instructions executed on the real architecture corresponding to the measured loop, we disassemble the binary and inspect the assembly code. Figure 4.4 shows the assembly ARMv8 code of the measured loop. We can observe the five sequences of flush and loads operations (from lines 1 to 11). Lines 10 and 12 are conditional operations due to the for loop we want to limit. We can see that they are less present than the flush and load operations. In this case, as the main memory access time is very long, we do not observe an impact of decreasing/increasing the number of operation sequences. However, increasing the number of operation sequences can have a significant impact on the measurement when targeting faster levels of the memory hierarchy, such as the L1 data cache.

```
1  4004b0:   dc  civac, x0
2  4004b4:   ldr x0, [x0]
3  4004b8:   dc  civac, x0
4  4004bc:   ldr x0, [x0]
5  4004c0:   dc  civac, x0
6  4004c4:   ldr x0, [x0]
7  4004c8:   dc  civac, x0
8  4004cc:   ldr x0, [x0]
9  4004d0:   dc  civac, x0
10 4004d4:   subs  x1, x1, #0x1
11 4004d8:   ldr x0, [x0]
12 4004dc:   b.ne  4004b0 <main+0x198>
```

Figure 4.4: *Measured loop assembly code of the running example microbenchmark.*

4.4.4 Results

We execute the microbenchmark on one of the four Cortex-A53 of the Raspberry Pi 3B+. Table 4.2 shows the results for the average measure time, i.e., the measured loop execution time divided by the number of accesses and the occurrences of the selected architecture events. We determine the expected event values corresponding to the scenario conditions regarding the desired scenario. For the architecture events, we expect a number of accesses to the L1 data cache equal to the number of loads, i.e., one million loads. As defined in the scenario, we want the memory requests to travel to the main memory. Thus, we expect a number of misses in the L1D close to one million. In the same way, we expect the number of accesses and misses to the L2 to be around one million. Finally, we want the number of TLB misses and prefetches close to zero as they add undesired latency (i.e., noise).

Table 4.2: *Measure time and hardware performance counters outputs from microbenchmark execution on Raspberry Pi 3B+.*

Monitored event	Expected value	Monitored value
Average access time	N/A	153 (ns)
L1D_CACHE_ACCESS	∼1000000	1000069
L1D_CACHE_REFILL	∼1000000	1000066
L2D_CACHE_ACCESS	∼1000000	1000282
L2D_CACHE_REFILL	∼1000000	1000084
L1D_TLB_REFILL	∼0	4
L1D_CACHE_REFILL_PREFETCH	∼0	8

We observe that the monitored results are very close to the expected values. The noise monitored by the performance counters is low enough. Consequently, we can assume we have successfully monitored the desired scenario. We can deduce the `memory latency` parameter value from the results. For that, we use the average access time monitored, removing the time needed by the memory request to go through the L1D and L2 caches. In this architecture, this time is 22 (ns) (see Section 5.3.7 for a complete analysis). So the calibrated parameter value is $153 - 22 = 131$ (ns).

4.5 Summary

In this chapter, we propose a methodology to calibrate simulator memory systems using a real state-of-the-art reference architecture. We introduce the two phases of the methodology. The first one, the simulator parameter identification, allows identifying the different simulator parameters that need calibration. For that, we propose a method based on a memory system template listing all the generic key parameters of the main components composing a memory system. At the end of this phase, we have a list of simulator parameters we need to calibrate.

In the second phase, the simulator parameter discovery, we start calibrating the simulator parameters using first-party documentation. Then, we detail the design of microbenchmarks that we execute on the real state-of-the-art architecture to deduce parameter values. The microbenchmark executes memory request sequences to stress specific elements of the memory system. At the same time, it uses hard performance counters to monitor memory system behavior and deduce parameter values. To achieve this goal, we introduce the methodology's different features to stress the memory system purposely and monitor the memory system behavior, reducing the different types of noise.

We use a running motivational example to instantiate and illustrate each methodology phase. In particular, we use a fictional main memory simulator and the Raspberry Pi 3B+ as the reference architecture. We detail the design of the microbenchmark that we execute on the Raspberry Pi 3B+, and we show the results from the performance counters. We verify first that we generate the desired memory behavior, then deduce the simulator parameter value.

In the following of this thesis, we instantiate the methodology described in this chapter on two practical use cases: (1) the calibration of the reactive part of a memory system in Chapter 5, and (2) the functional characterization of an L1 data prefetchers in Chapter 6.

V

Memory System Timing Calibration

In this chapter, we propose to instantiate the methodology described in Chapter 4 in order to realize a timing calibration of simulator memory systems. We first detail the specifications of the instantiation. Then, we implement it with the gem5 simulator and one Cortex-A53 of the MediaTek Helio X20 SoC as real reference state-of-the-art architecture [15]. Thus, we extract from the target architecture multiple technical information such as the access times to the different cache levels, the main memory, or the TLB. Finally, we evaluate the methodology instantiation with benchmarks from the SPEC CPU2006 suite. We execute them on the target architecture to have a reference, and then we simulate them on gem5 using the default and calibrated models to expose our methodology's benefits.

5.1 Background and Motivation

Computer system simulators are widely used by researchers. They allow quick evaluations of new ideas avoiding long expensive manufacturing processes. Those new ideas are evaluated with respect to a state-of-the-art baseline corresponding to the target architecture, e.g., low-power mobile architecture or high-performance server architecture. Consequently, the relevance of the simulated results is directly related to the quality and the choice of that baseline.

5.1.1 Memory System Modeling

The memory system plays a key role in all instruction-processor based compute platforms. A slow data access time directly impacts the instruction execution flow and reduces the whole system performance. This statement is even more true with modern multicore architectures containing many components like cores, GPU, or programmable accelerators that compute data at Gigahertz frequencies. Thus, the data movement in the multiple levels of the memory hierarchy needs to be fast and provide high bandwidth. However, improving the memory system is not an easy task to do due to high complexity of the memory hierarchy which includes different memory technologies, organizations or access protocols. New emerging non-volatile memory technologies look like a good opportunity [2, 145] to reduce memory system energy consumption providing the same or better level of performance. However, those technologies have different technical specifications than the usual memory technologies implemented. This means that a straightforward replacement is not possible. That way, researchers need a complete understanding of all the processes/sub-processes

running internally in the memory system. Faithful reference models incorporating all the workload-dependent effects are important for relevant improvement/optimization. I.e., we avoid "black box models" or calibrated analytical models which do not capture those detailed effects.

5.1.2 Motivational Example

To motivate our work, we execute and simulate, on a real CPU and gem5, benchmarks from the SPEC CPU2006 suite [57]. We use the Cortex-A53 from the MediaTek Helio X20 SoC [149] as the reference architecture. This SoC is implemented in many smartphones and has the typical ARM mobile architecture, including different performance-level clusters. For the gem5 simulator, we use the High Performance In-order (HPI) CPU model provided by ARM. For each benchmark, we plot in Figure 5.1 the number of Instructions Per Cycle (IPC). Then, we normalize the IPC difference using the real CPU results. We plot the results, i.e., the absolute IPC error of the simulation model relative to the execution on the real processor, in Figure 5.2.

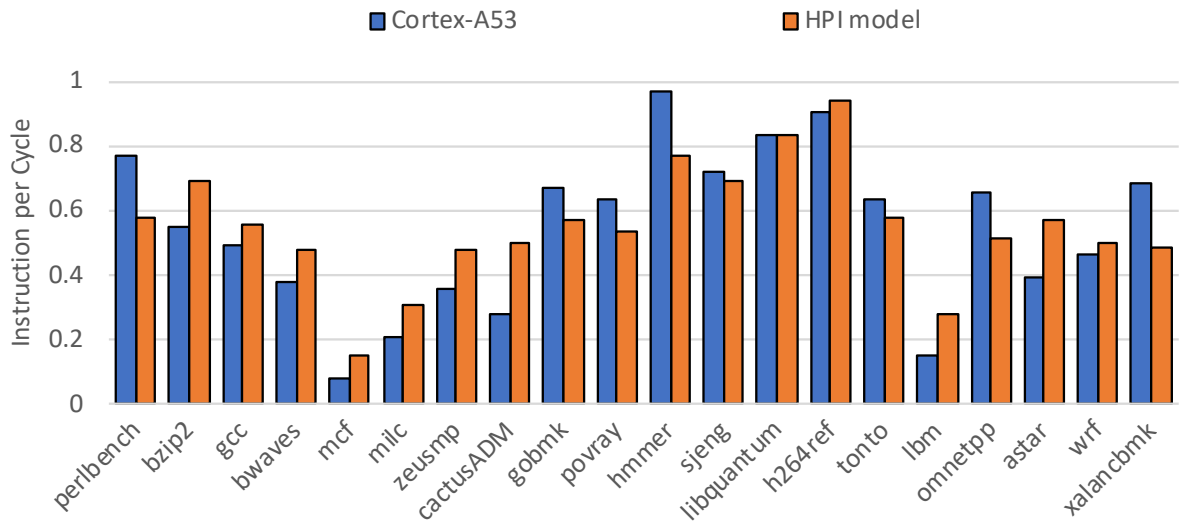

Figure 5.1: *IPC for the SPEC CPU2006 benchmarks executed on the MediaTek Helio X20 architecture and simulated with the default HPI gem5 configuration script.*

We add to Figure 5.2, the average and maximum IPC error values. We observe that even with the great detail level in the HPI model, the IPC error is higher than 80% for two of the twenty benchmarks. Those values come from benchmarks with very low IPC values, probably due to high memory activity, which might slow down the complete system. The average IPC error comes from absolute IPC error values. Looking at Figure 5.1 we observe that the error is not always positive or negative. This

variation may indicate that not only the access latency is different but also the behavior of the memory system.

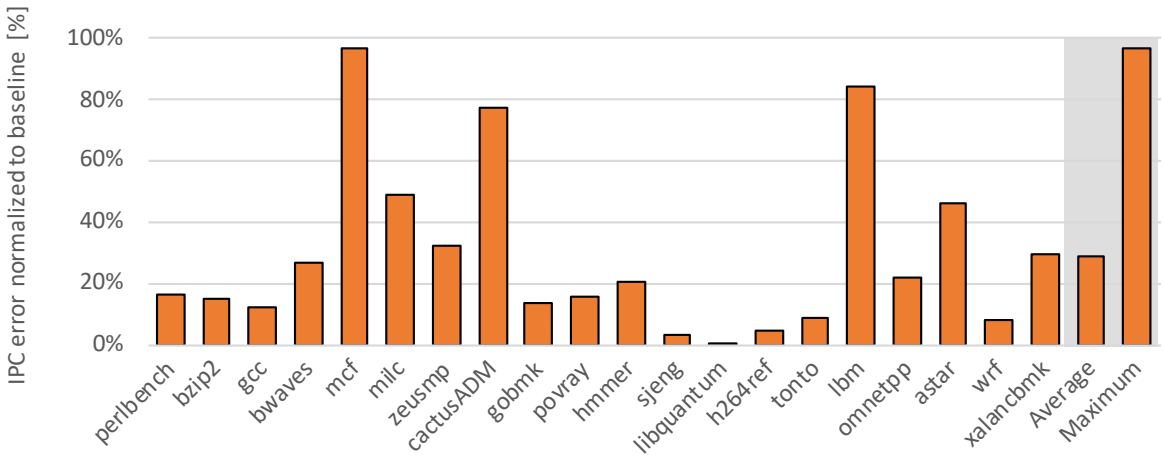

Figure 5.2: *Absolute IPC error for the SPEC CPU2006 benchmark suite simulated with the default HPI gem5 configuration script and normalized by execution on the MediaTek Helio X20.*

Our goal in this chapter is to reduce the average and maximum IPC errors, as they are too high to compare options properly in a design space exploration activity. Indeed, the proof of concept provided by the simulator would not be sufficient. For that, we propose to instantiate the methodology introduced in Chapter 4 to calibrate a portion of the simulator memory system to match the behavior of real state-of-the-art architecture faithfully.

5.2 Methodology Instantiation

In this section, we detail key elements of the instantiation of the methodology described in Chapter 4. First, we define the reactive memory system. Then, we detail the path of the data memory requests through the reactive memory system. Finally, we model it as a set of multiple conditions and delays. Depending on the value of each condition, a memory request accumulates more or less delay traveling through the reactive memory system.

5.2.1 Reactive Memory Subsystem

A typical memory system contains multiple components such as caches, main memory, memory-management unit, hardware data prefetchers, interconnect, etc. We group those components into two categories: the proactive components and the reactive components. The proactive memory subsystem includes all the elements that issue new memory requests speculatively expecting that the program might need in the future, e.g., data prefetchers. On the other hand, the reactive memory subsystem includes all the components that issue the memory requests based on event triggers from the context and environment of the programmable cores. In this chapter, we focus on the reactive memory subsystem while we discuss the proactive memory subsystem in Chapter 6.

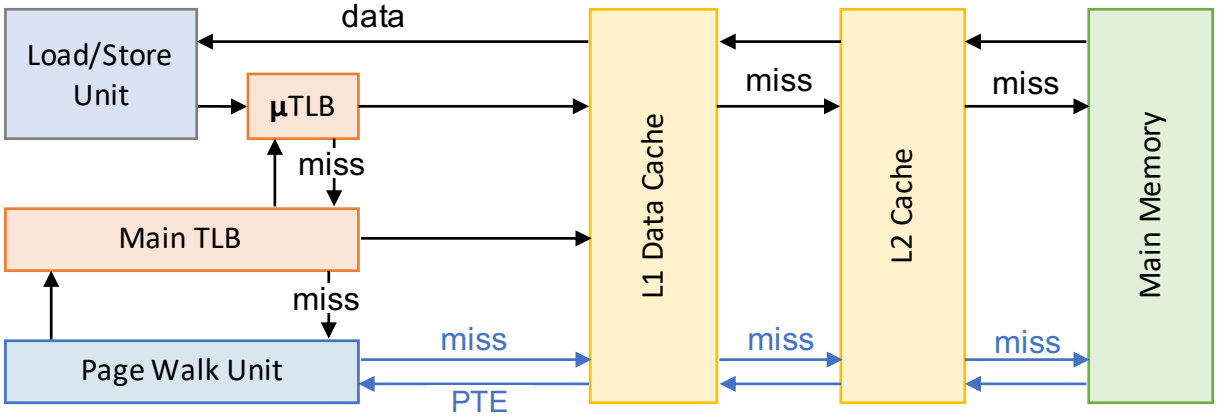

Figure 5.3: *Reactive memory system template. Translation from virtual addresses to physical addresses through multiple TLB levels and the page walk unit.*

Figure 5.3 shows the generic way a request goes through the reactive memory subsystem. First, the request is generated by the load/store unit. This request goes to the Translation Lookaside Buffer (TLB) to translate the virtual address into a physical address. The TLB can have multiple levels of hierarchy. In the figure, the first level of the TLB is the micro TLB. If the requested address translation is missing in this micro TLB, i.e., micro TLB miss, the request goes to a larger main TLB. If the address translation is still missing, i.e., TLB miss, the TLB propagates the request to the Page Walk Unit (PWU). The address translation data are stored in a page table. Every Page Table Entry (PTE) corresponds to one address translation. The PWU contains a Page Table Walker (PTW) and a page walk cache. The PTW compiles missing PTE by executions several memory accesses. The page walk cache reduces the memory accesses caused by the PTW. It happens that a process accesses a virtual page for which there is no PTE in the page table, i.e., a page fault occurs. In that case, the Memory Management Unit (MMU) raises an exception and hands control over to the OS kernel. The latter

brings the page from the disks to the main memory and updates the PTE with the corresponding physical address in the page table. This process interrupts the application for several 1000's of CPU-cycles Once the PTE is present in the PWU, the TLBs are updated, and the memory request is sent to the L1 data cache. Depending on where the data sits, the request can travel through the different levels of the cache hierarchy and the main memory. In the case of a load request, the request goes back to the load/store unit with the requested data updating the cache content on the way. We believe this template to be generic enough to represent most of the modern reactive memory subsystems inside the SoC, i.e., excluding outside components like storage memory. Some features, like the number of caches or TLB levels, can be easily adapted to represent a specific target architecture more precisely.

5.2.2 Delay Model

To design adequate microbenchmarks, we study the access time of a request in the reative memory system. This one departs from the load/store unit of the processor core and accumulates delay as it travels deeper in the target memory hierarchy. The sum of these individual delays constitutes the access time of the request, as illustrated in Figure 5.4. The path followed by a particular request depends on a set of conditions, such as where the data sits in the memory hierarchy or the current state of the memory components. For instance, if the data is located in the main memory, the state (i.e., either open or close) of the target bank significantly impacts the main memory access time, as explained in Section 2.2.3. We call this representation in Figure 5.4, the delay model.

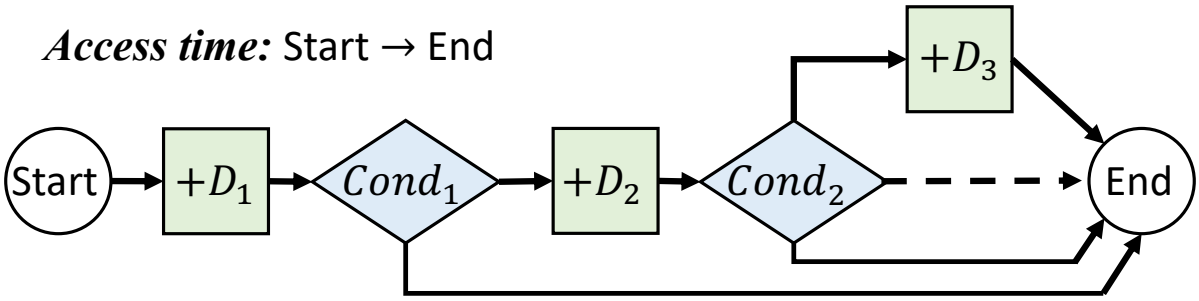

Figure 5.4: *The delay of a memory request increases as it deepens in the hierarchy.*

Each particular path on the delay model corresponds to a scenario. In order to determine every delay, we need to first measure the access time of the paths with the minimum unknown delay. Then, we step by step fill the delay model with the missing information. For that, we use the microbenchmark features to pass or fail delay model conditions and isolate one delay. For instance, in Figure 5.4 we configure the

microbenchmark to pass the $Cond_1$. Thus, we can measure D_1 as it is directly equal to the access time. The D_1 value can next be used to isolate D_2 by failing $Cond_1$ and passing $Cond_2$. This way, we can fill one by one the delay values and complete the whole delay model. Finally, we use the delay model to properly set up the simulator parameters we identify in the first phase of the methodology.

5.3 Methodology Implementation

We implement the described methodology's instantiation to the gem5 simulator that we further extend with Ramulator to model the working main memory. An introduction to those tools is present in Section 2.3. We use the Cortex-A53 as the target reference CPU architecture. Specifically, we use a development board from 96boards [15], which includes the MediaTek Helio X20 SoC [149], and we execute our microbenchmarks on one of the eight Cortex-A53 cores included in the SoC[1]. As described in Chapter 4, we start by identifying the gem5 simulator parameters. Then, we detail the design of the microbenchmarks we use to extract missing parameter values from the MediaTek Helio X20 SoC.

5.3.1 Gem5 Memory System

We start by creating a first model of the target SoC on the gem5 simulator instantiating all the components. To model the Cortex-A53 of MediaTek Helio X20, we use the High-Performance In-order (HPI) CPU model available in gem5. This model is provided by ARM and represents a modern high-performance ARM in-order core. Section 2.3 provides more details about this model. It includes three cache modules: `HPI_DCache`, `HPI_ICache` and `HPI_L2`. They respectively represent the L1 data, L1 instruction, and L2 caches. Those modules are derived from the gem5 basic `Cache` object instantiated with different parameter values. By default, the interconnect and the main memory are modeled with the `SimpleMemory` module. This module simply issues requests with fixed latency and bandwidth. Regarding the TLB, the `ArmTLB` module is used to model the TLB. However, because we use gem5 in Syscall Emulation (SE), we do not simulate a complete operating system. Instead, gem5 SE implements a simplified address translation process: memory requests always hit in the TLB, adding a fixed latency. In the same way, we do not model the page faults and the storage subsystem. This baseline gem5 model is represented in Figure 5.5 with the name *Default*.

[1]The MediaTek Helio X20 SoC also includes two Cortex-A72 cores to amount to a total of ten cores.

During the calibration in Section 5.3.7, to properly model the interconnect as a single module, we add a `Bridge` module between the `SimpleMemory` and the `HPI_L2`. This module is a fixed-size buffer that holds memory requests for a determined time. This new model is named *CalibratedV1* in Figure 5.5. Finally, we propose a last gem5 model, *CalibratedV2*, in Section 5.3.10. We use Ramulator to replace the SimpleMemory module in CalibratedV1, allowing a more detailed simulation of the main memory.

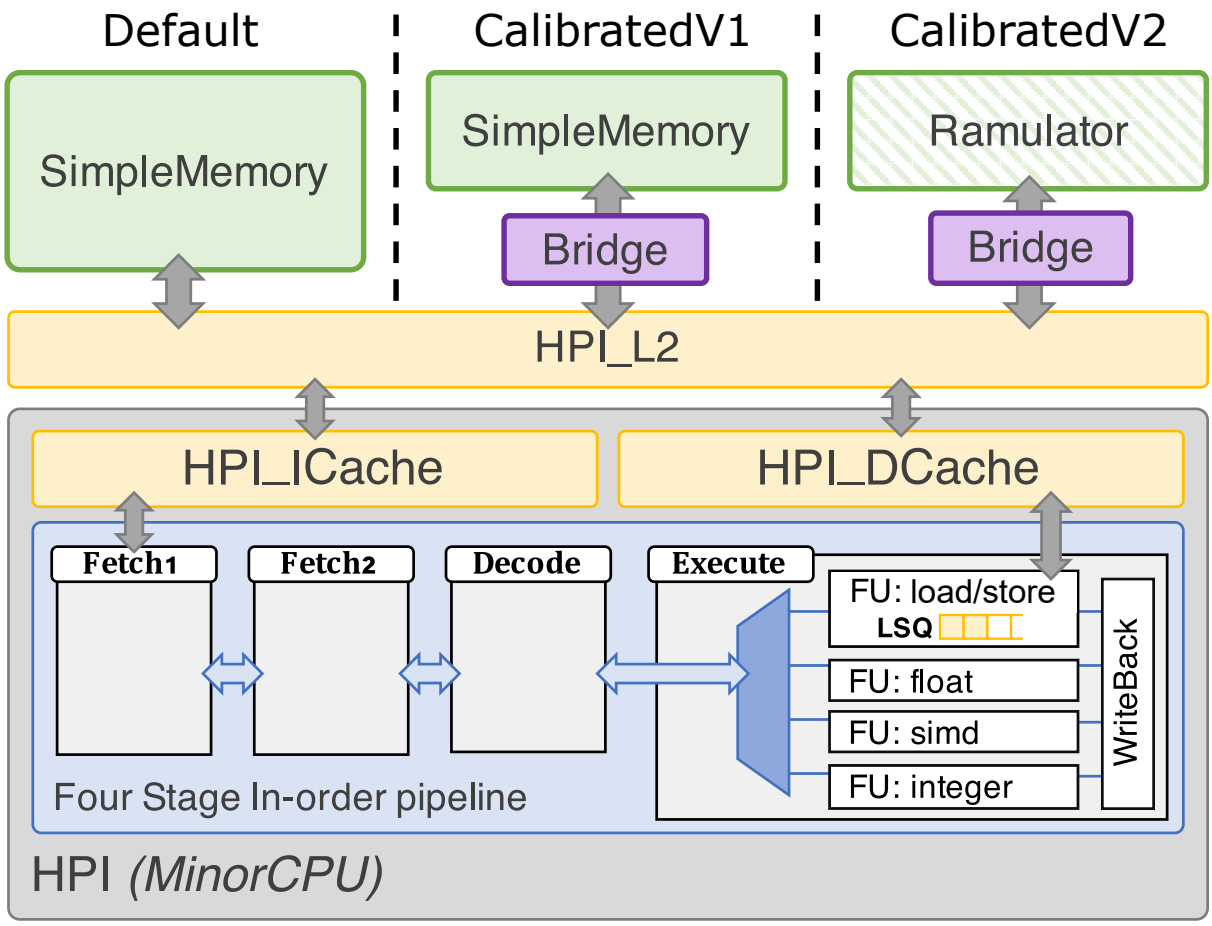

Figure 5.5: *Gem5 simulated models: CalibratedV1 results from applying our methodology to the on-chip memory system, CalibratedV2 extends the scope to the main memory, and Default is the default gem5 model.*

5.3.2 Gem5 Key Parameters

As described in the methodology, we analyze the path that the memory requests follow in the memory system to identify key simulator parameters. First, the memory instruction is executed by the load/store unit of the HPI core, the instruction is issued to a Load/Store Queue (LSQ), and a memory request is generated. The instruction stays in the LSQ until the memory request is fully executed. Two parameters are important: the maximum number of memory accesses and the size of the LSQ. Those

parameters allow the choice of the number of outstanding requests the core can generate. We list the relevant modules and their parameters in Table 5.1. Once a memory request is issued, it goes to the HPI_DTB, which issues the request with a fixed latency. However, as the simulator does not model TLB processes, we do not identify any key parameter in the TLB. Its fixed latency is implicitly considered in calibrating the L1 cache access latency. Then, the memory request goes through the different levels of caches depending on where the data sits. For each HPI cache module, we select five key simulator parameters: size, associativity, data access latency, replacement policy, and clusivity, which define the inclusion policy. If a request misses both cache levels, it goes to the main memory through the interconnect. By default, the SimpleMemory is instantiated to model that part. As we simulate only one CPU, only the latency is important in this module. However, in the case of multi-core simulations, the bandwidth would also be important. We now need to find the correct values for each parameter to best match the timing of the real hardware platform memory system.

5.3.3 First-party documentation

We start the parameter discovery phase with public first-party documentation. The first document is Cortex-A53 technical reference manual [148]. This document is provided by ARM and gives an overview of the Cortex-A53 structure and features. For instance, we can find the number of outstanding requests the core can generate, which is three. This value is already different than the default one we have in the HPI CPU model. We use those new values to do a first calibration of the default model. This first calibration is listed in the column CalibratedV1 of Table 5.1. This document also gives some information about the caches. We can find the associativity, inclusion, and replacement policies. As we can see in Table 5.1, the new parameter values are also different from the default ones. Regarding the sizes of the caches, multiple sizes can be implemented. Thus, we need to refer to the MediaTek Helio X20 SoC functional specification documentation [149] to know which size is implemented in this SoC. The functional specification documentation is also useful to know which DRAM devices are used in the SoC. In this case, LPDDR3 devices are implemented and divided into two channels with two 32-bit buses. We highlight in bold in Table 5.1 the parameter values that we find in the first-party documentation. Importantly, those parameters are provided by the manufacturer, which can choose whether or not to make them public. Hence, the public parameter list changes depending on the SoC. At the end of this step, several key parameters are still missing (e.g., the cache memory access latency, the number of parallel access that a cache can process, or the memory controller buffering latency).

Table 5.1: *List of key parameters with default and calibrated values.*

gem5 Module	gem5 Parameter	Default	CalibratedV1	CalibratedV2	Source
HPI	executeMaxAccesses-InMemory	2	**3**	**3**	both
	executeLSQTransfers-QueueSize	2	**3**	**3**	both
	enableIdling	True	False	False	µbench.
	srcRegsRelativeLats[a]	[2]	[0]	[0]	µbench.
HPI_DCache	size	32KB	32KB	32KB	both
	data_latency	1	2	2	µbench.
	assoc	4	4	4	ref. manual
	replacement_policy	LRURP	RandomRP	RandomRP	both
	clusivity	incl	excl	excl	ref. manual
	writeback_clean	False	True	True	ref. manual
HPI_L2	size	512KB	512KB	512KB	both
	data_latency	13	10	10	µbench.
	assoc	16	16	16	ref. manual
	replacement_policy	LRURP	RandomRP	RandomRP	both
Bridge	delay	NA	48ns	30ns	µbench.
SimpleMemory	latency	30ns	30ns	Ramulator[b]	ref. manual

[a]Parameter from the HPI_DefaultMem64 submodule
[b]Ramulator replaces the SimpleMemory, its parameters are listed in the Table 5.2 .

5.3.4 Memory Level Microbenchmark

In order to find missing parameter values, we design a microbenchmark following the structure introduced previously in the section. Thus, to pin the data in a particular region, we use an array of different sizes. We incrementally increase the array size from smaller than the first level cache size to larger than the last level cache (LLC) size. That way, the data is gradually pinned deeper in the memory hierarchy. The granularity of data transfers in the memory subsystem is a cache line, usually 64 bytes. When a word is requested by the load/store unit, the complete cache line is first transferred to the L1 data cache, and then the specific word is issued to the core. To ensure that the data transfer of a previous load does not affect next load, the microbenchmark only reads the first word of a cache line. In addition, pointer chasing is implemented in the array, meaning that each first word of a cache line stores the address of the next cache line to be accessed. That way, two consecutive loads to the same array cannot be executed in parallel due to the data dependency. We use multiple arrays implementing independent pointer chasing to control the number of memory requests the core can issue in parallel. We designed the microbenchmark to generate one to four independent loads, exposing the amount of parallelism at each memory system level. Also, we add permutations in the pointer chasing to generate a random sequence of memory accesses and prevent triggering hardware data prefetching mechanisms. Figure 5.6 shows the microbenchmark source code for two independent loads.

```
long long int array_1[MAX_SIZE], array_2[MAX_SIZE];
long long int *ptr_1, *ptr_2;

init_hw_perf_counters(); // Initialization of the HPCs
for(int size = MIN_SIZE; size < MAX_SIZE; size += IT_SIZE){

    // INITIALIZATION
    init_ptr_chasing(array_1); // Randomly linked cache lines
    init_ptr_chasing(array_2); // Randomly linked cache lines
    ptr_1 = array[0]; // Init 1st pointer to a cache line
    ptr_2 = array[0]; // Init 2nd pointer to another line

    // MEASURED LOOP
    start_hw_perf_counters(); // Reset HPC
    for(i = 0; i < ACC/NUM_LOADS; i++){
        ptr_1 = *ptr_1; ptr_2 = *ptr_2; // Loads 1 and 2
        ptr_1 = *ptr_1; ptr_2 = *ptr_2; // Loads 3 and 4
        ...
        ptr_1 = *ptr_1; ptr_2 = *ptr_2; // Loads 15 and 16
    }
    read_hw_perf_counters()(); //Read HPC
}
```

Figure 5.6: *Microbenchmark C code designed to extract memory level signatures*

We structure the microbenchmark with an initialization phase and the measured loop described in the methodology. We iterate those phases with different array sizes, from the smaller to the bigger. For each iteration, we first fix the size of the array(s) and implement the pointer chasing. Figure 5.7 illustrates the pointer chasing implementation with an array of 1KB and eight words per cache line, i.e., 64-bit word. We create as many pointers as arrays and point them on the corresponding array's first element. The measured loop is simply implemented as a for loop. To generate a memory load, we ask the pointer to take the value of the address it is currently pointing to. We link the last element of the pointer chasing to the first one. That way, when the pointer reaches the end of the array, it reads the array again from the beginning without any extra conditional instruction. We do multiple memory loads in the same loop iteration to reduce the interference of loop control instructions. After compilation, we verify that the monitored memory instructions dominate the loop's body by inspecting the disassembled binary code.

5.3.5 Memory Level Signatures

To execute the microbenchmark on the board, we first fix the frequency of one Cortex-A53 to 1.391 GHz using a Linux tool, i.e., the Linux `CPUFreq` subsystem. Then we

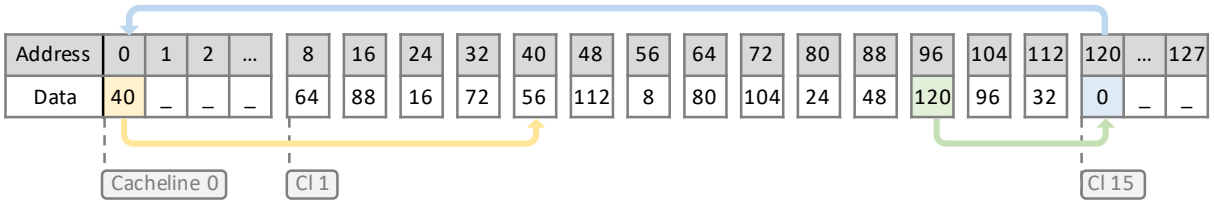

Figure 5.7: *Microbenchmark pointer chasing implementation.*

shut down the rest of the cores present in the cluster. To assign the microbenchmark to a particular core, we use the command `taskset`. We run the benchmark with sixteen loads by loop iteration and more than 10 million iterations [2]. Finally, we execute the microbenchmark ten times and take the minimum average access time for every array size. We repeat the process varying the number of independent loads from one to four. We show the results in Figure 5.8.

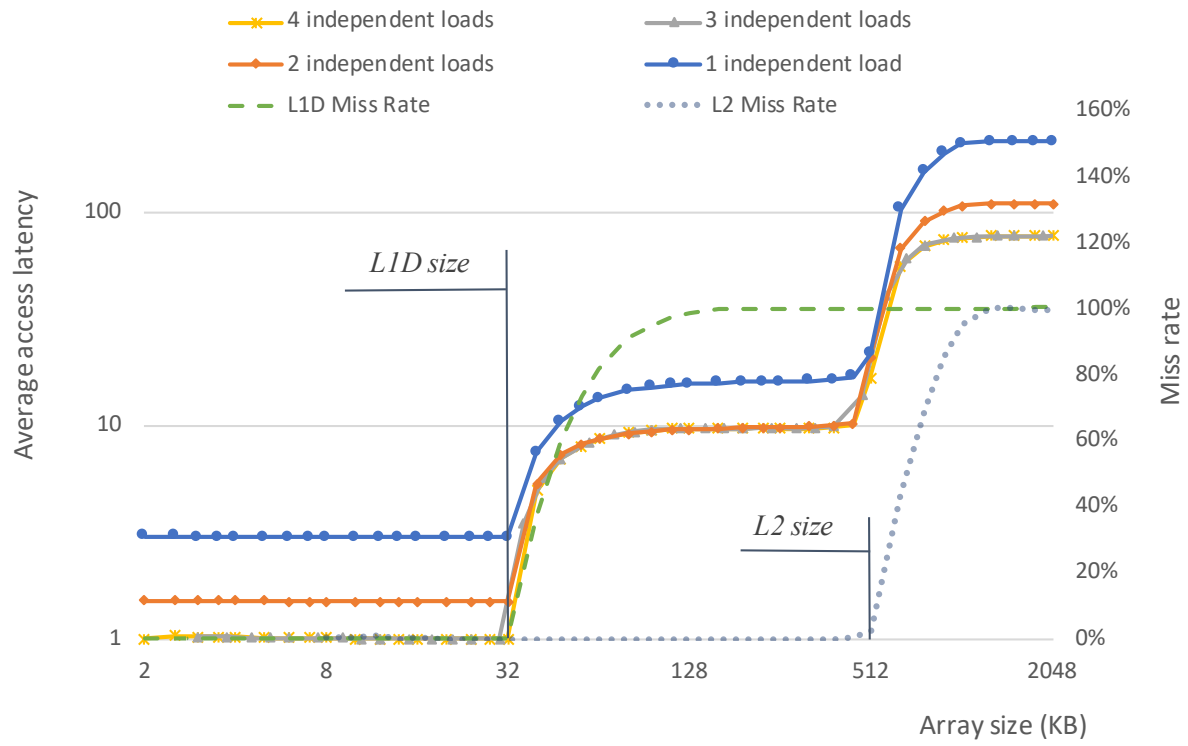

Figure 5.8: *Results from the execution of the microbenchmark on one Cortex-A53 of the MediaTek Helio X20.*

We can see that the average access becomes higher as the array size increases, which is the expected behavior. We identify three regions where the average access time is stable: from 2 to 32KB, from 128KB to 512KB, and around 2048KB. Those regions represent the cases where the array accessed in our microbenchmark is stored in the L1 data cache, L2 cache, and the main memory, respectively. We can verify this statement

[2] These parameters were tuned experimentally.

with the results from the hardware performance counters, i.e., the L1 data miss rate and the L2 miss rate. Thus, for the three regions the L1 data and L2 miss rates take respectively the values [0%, 0%], [100%, 0%] and [100%, 100%]. Hence, the results with no parallel loads executed (1 independent load) expose the data access times of the L1 data, L2, and the main memory. Hence, we need approximately 3 cycles to access the L1 data, 13 more cycles to access the L2, and 200 extra cycles to access the main memory. We see Subsection 5.3.7 how we can extract precise results from the curves. With 2 and 3 independent loads, the average L1 data cache access time is divided by two and three, but in the L2 region, we can observe a saturation. With 4 independent loads, the average latency is no longer reduced, indicating memory-level parallelism saturation. Hence, the benchmark exposes that the Cortex-A53 can generate up to 3 outstanding memory requests. Between the three stable regions, we have two transition regions when the array is stored in two levels. Those regions start when the array can no longer fit in the cache and goes to the next level. That way, we can deduce the size of the L1 data, 32KB, and the size of the L2, 512KB.

5.3.6 Cache Replacement Policy

The shape of the average access time curve when transitioning between two memory levels indicates a random replacement policy in the L1 data cache and the L2 cache. In order to illustrate how another replacement policy could impact the average access time, we create a simple python script cache model that implements two different replacement policies: a Random policy and a Last Recent Used (LRU) policy. We create exactly the same cache organization as the MediaTek Helio X20 L1D, i.e., 4way associative with a size of 32KB. We implement two replacement policies: a Random policy and a Last Recent Used (LRU) policy. We stress the cache with different array sizes as the microbenchmark does. For each size, we extract the miss rate depending on the policy and compare it with the results from the MediaTek Helio X20. Figure 5.9 shows the results. We can see that the miss rate results from a Random policy perfectly match the MediaTek Helio X20 L1D miss rate curve. Conversely, the LRU policy generates a real break in the miss rate, which is distinctive behavior.

5.3.7 Memory Level Access Times

In order to measure more precisely the access time of the different levels, we plot in Figure 5.10 the average access time depending on the L1 data miss rate. We focus on the transition between the L1 data and the L2. Each point represents an array size

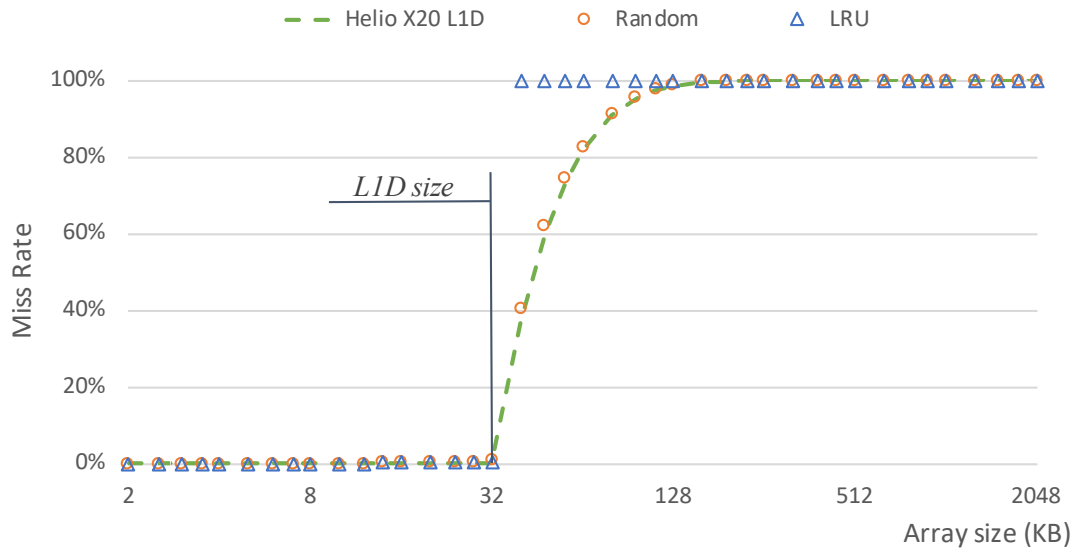

Figure 5.9: *Miss rate from the L1D of the MediaTek Helio X20 and two cache models implementing a Random and a LRU replacement policy. The size and the associativity are the same for the three caches.*

between 384KB and 6144KB. Like that, we can graphically understand the impact of the L1 data miss rate on the average access time. We also plot the linear regression line of the empirically measured data, its equation, and its R^2 correlation coefficient. We make two observations. First, the relation is strictly linear as the correlation coefficient is close to 1. This means no other event affects the access time, i.e., the access time follows the delay model. Second, we conclude that the L1 data and L2 access times 3 cycles (for x = 0%) and 13 cycles (for x = 100%), respectively.

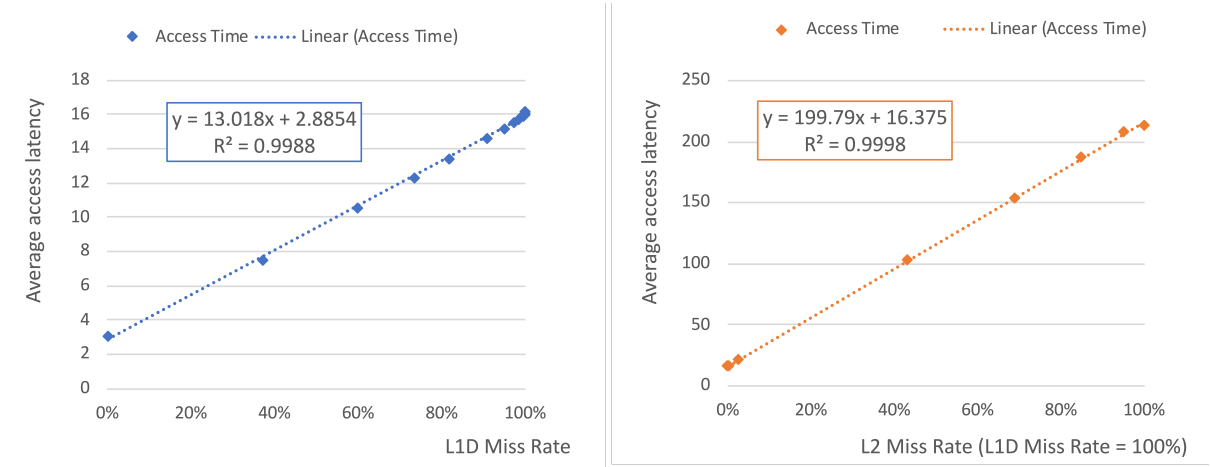

Figure 5.10: *Average memory access time on the MediaTek Helio X20 depending on the L1D and L2 miss rates.*

We repeat the same analysis but focusing the L2 and the main memory. This time, we plot the average access time depending on the L2 miss rate. The result is shown in Figure 5.10. Again, we observe a highly precise linear correlation. We need 16 cycles

to access the L2, which is coherent with results in Figure 5.10. Then we need 200 extra cycles to access the main memory.

Finally, we finish the first gem5 calibration using the new parameter values discovered with our microbenchmark. All the parameters of this calibration are listed in Table 5.1 under CalibratedV1.

5.3.8 Micro TLB Penalty

Even if the TLB processes are not modeled in the gem5 SE simulations used in this Thesis, we propose to design another microbenchmark to expose TLB information. gem5 also includes a Full-System (FS) simulation mode [150] that can be calibrated using the information exposed by our microbenchmark. The Cortex-A53 reference manual [148] provides information about the TLB as the number of levels, which is two: one instruction and one data micro TLBs and one main TLB. The numbers of entries are 10 for the micro TLBs and 512 for the main TLB. However, the TLB miss penalty latency is not public. Consequently, we first design our microbenchmark to pin the address translation data (i.e., information for virtual to physical address translation) in the data micro TLB using the same microbenchmark structure illustrated previously in the chapter. To pin the translation data in the micro data TLB, we implement pointer chasing, which loads only one cache line per page. To minimize conflict misses (cache misses due to limited associativity), we use all the L1 data cache indexes by changing the address page offset of each load. Then, we gradually increase the number of pages accessed by the microbenchmark.

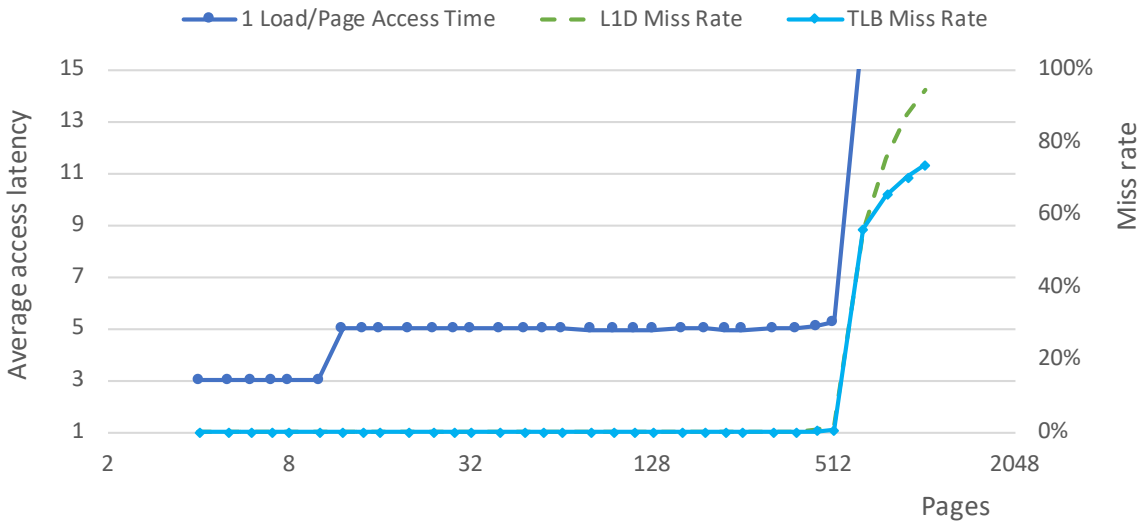

Figure 5.11: *Average access time depending on the number pages accessed with different page-offsets on the MediaTek Helio X20.*

We execute the microbenchmark on the MediaTek Helio X20. The results are shown in Figure 5.11. We identify three distinct regions. The first one is the region before 10 pages. The average access time is stable and equal to 3 cycles. In this region, the data is present in the L1 data cache, as we can see with the L1 data cache miss rate. Regarding the translation information, it is present in the micro TLB as this is the fastest possible address translation. The second region is between 10 and 512 pages. The data is still present in the L1 data cache, but we now observe a 2-cycle extra latency to access the L1 data cache. This extra latency comes from a micro TLB miss and the consequent access to the main TLB (see Figure 5.3). Another observation is that sudden transition between the first and second regions indicates a Last-Recent-Used entry replacement policy. The third region corresponds to a number of pages bigger than 512. In this case, we can observe with the TLB miss rate, reported by the performance counter, that the translation data start to miss in the main TLB. However, at the same time, the L1 data cache miss rate also increases as the L1 data cache can no longer store all the pointer chasing cache lines (32KB = 512 cache lines). Consequently, we cannot directly extract more information as we cannot isolate the impact of L1 data cache misses and TLB misses.

5.3.9 Main TLB Penalty

In order to expose the penalty latency of a TLB miss, we modify the microbenchmark pinning the data in the different levels of the memory system. This time, instead of using different address page offsets for each load, we use exactly the same page offset. That way, we use a limited number of indexes in the caches and with a low number of pages, we can pin the data in the main memory. Then, we increase the number of pages to generate TLB misses. Finally, we observe the impact on the average access time to measure the penalty latency of a TLB miss.

We execute the microbenchmark on the MediaTek Helio X20. The results are shown in Figure 5.12. We identify three regions. The first region is between 128 and 384 pages. As we can see with the L1 data and L2 cache miss rates, the data is not exclusively present in the main memory yet, i.e., the length of the pointer chasing is not long enough. The second region is between 384 and 512 pages. We observe that the data is now present in the main memory (i.e., 100% L2 miss rate) and the address translation process is still not missing in the TLB. The third region is after 512 pages. The TLB miss rate increases, adding extra latency to the average access time. We can observe that the L1 data and L2 cache miss rates remain stable, indicating only the TLB misses impact the average access time. As shown in Figure 5.9, how the TLB miss rate is increasing

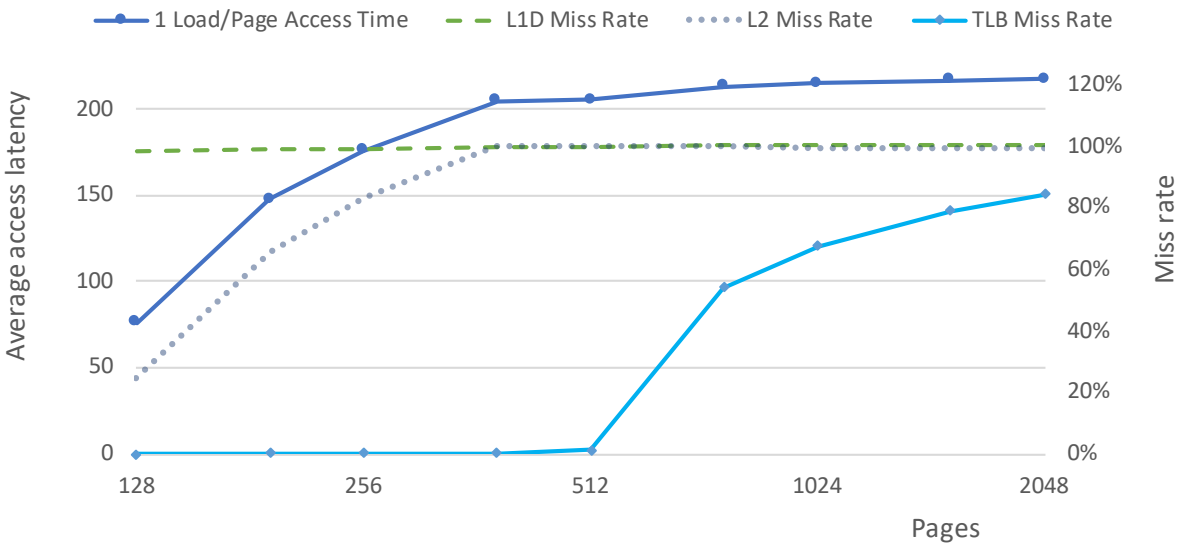

Figure 5.12: *Average access time depending on the number pages accessed with the same page-offset on the MediaTek Helio X20.*

indicates that a random entry replacement policy is implemented in the main TLB.

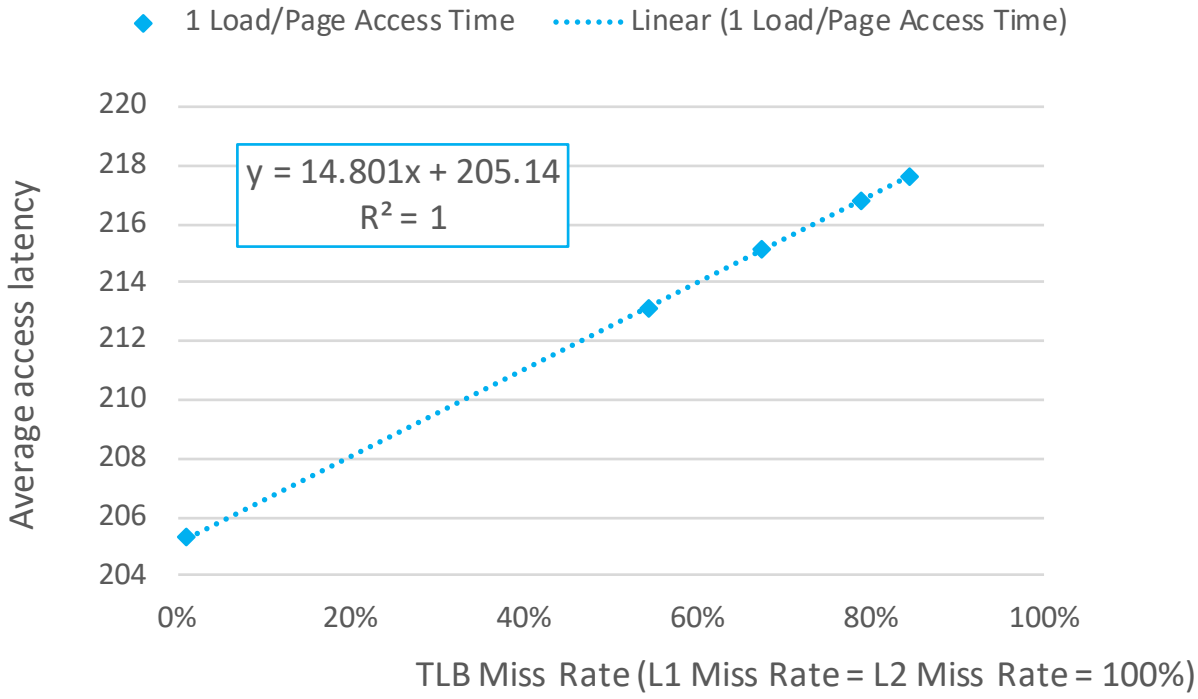

Figure 5.13: *Average access time on the MediaTek Helio X20 depending on the TLB miss with L1D and L2 miss rates equal to 100%.*

To expose precisely the penalty latency of a main TLB miss, i.e., the access time to the page walk cache, we plot in Figure 5.13 the average access time depending of the TLB miss rate of the third region. Each point corresponds to a different number of

pages, from 512 to 1024. We plot the corresponding linear regression line, its equation, and the R^2 correlation coefficient. As we can see with the correlation coefficient equal to 1, the relation is perfectly linear, i.e., only the TLB miss rate impacts the average access time. Additionally, the linear direction coefficient gives us the cost of a main TLB miss, which is 15 cycles.

To summarize, the delay model extracted from our microbenchmarks indicates that no extra delay is incurred in a micro TLB hit, a 2-cycle penalty is added in a micro TLB miss and an extra 13-cycle penalty in case of a main TLB miss.

5.3.10 Main Memory Address Mapping

In this section, we extend the CalibratedV1 structure made previously with an accurate main memory simulator as illustrated in Figure 5.5 with the CalibratedV2 structure. Basically, Ramulator [30] replaces the gem5 SimpleMemory module intending to provide more accurate modeling of the main memory subsystem. However, we still do not simulate the page fault effects. For that, we would need to use a storage disk simulator such as MQsim [82]. This would imply using another kind of simulation that includes the full OS routine, i.e., Full System simulation. We give more details in Section 2.3. We apply our methodology again to rebuild the final stages of a MediaTek Helio X20 delay model continuing from the interconnect to the main memory. Ramulator includes new parameters that need proper instantiation, such as the number of channels, ranks, DDR protocol, etc. We find the values of most of the parameters in the MediaTek Helio X20 documentation [149]. The main memory comprises two LPDDR3_1886 devices of 8Gb with a 32-bit output, which amounts to 2GB of main memory. These devices are divided between 2 channels with only one rank per channel. We list all the Ramulator parameters that are relevant to our study in Table 5.2.

Table 5.2: *List of Ramulator key parameters*

Ramulator parameter	Value	Source
standard	LPDDR3	ref. manual
channels	2	ref. manual
ranks	1	ref. manual
speed	LPDDR3_1886	ref. manual
org	LPDDR3_8Gb_x32	ref. manual
address mapping	Figure 5.16	µbenchmark

[a]Parameter hardcoded in Ramulator source code.

However, no information is provided about the address mapping scheme. The

address mapping indicates what bits of the physical address are used to identify the channel, bank, row, and column of a request. Although address mappings may be deliberately kept secret to dissuade security attacks, they are essential to properly model the performance of a main memory system.

In order to expose the address mapping, we use two memory requests that we pin the main memory. The first request has a fixed physical address, while the second address varies randomly at each iteration. A row-buffer conflict is triggered when both addresses sit in the same channel and bank but in a different row. Such row-buffer conflict adds extra latency to the main memory access time. Thus, by measuring the main memory access time, we can identify which address pair leads to a row-buffer conflict. Thus, we have two distinct scenarios, i.e., main memory accesses with and without row-buffer conflict.

We do ten thousand iterations using different address pairs (always keeping the same first address while the second one is generated randomly). We access each address pair one million times in one iteration to extract an average access time. We verify the data pinning in the main memory for all the pairs using the performance counters. We execute the microbenchmark on the MediaTek Helio X20 and plot the result in Figure 5.14. The figure shows a histogram of the average access time of the ten thousand address pairs. We make two main observations. First, most of the population has an average access time around 109 cycles which is coherent with the value in Figure 5.8 using two independent loads. Second, a small group of address pairs has an average access time close to 138 cycles. We identify them as the address pairs that trigger a row-buffer conflict in the main memory.

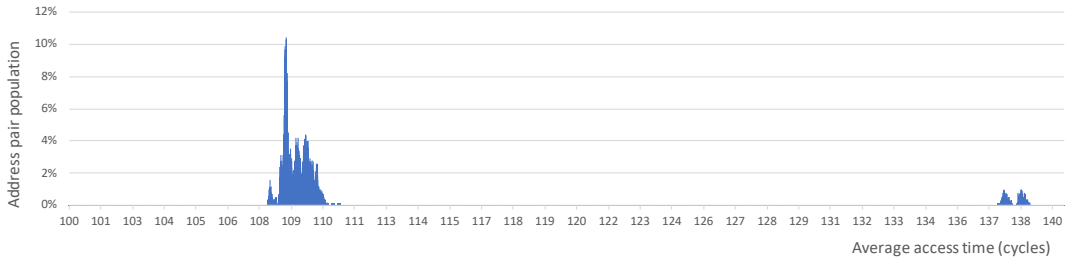

Figure 5.14: *Average delay distribution of main memory request pairs.*

In order to deduce the address mapping, we analyze the physical addresses of the pairs present in the 138-cycle group. This group contains n pairs of addresses. The first address is always the same. We call it $Addr_0$. The second address is different for each pair. We call it $Addr_k$ ($0 < k \leq n$). All the addresses in this group share the same channel, rank, and bank (only one rank in our case). Thus, the address bits mapped to the channel and bank are the same. With the goal of improving memory-level parallelism, modern memory controllers often XOR multiple bits in the address

mapping to distribute accesses randomly across banks, ranks, and channels. In this way, we look for similarities in the 31 address bits[3] and their XOR combination results. For that, we define $Addr_k(i)$ as the bit i of the address k. For each combination (i, j), we check if $XOR(Addr_k(i), Addr_k(j))$ is always equal to $XOR(Addr_0(i), Addr_0(j))$ for $0 < k \leq n$. We print the result on Figure 5.15. From the matrix result, we can make four observations:

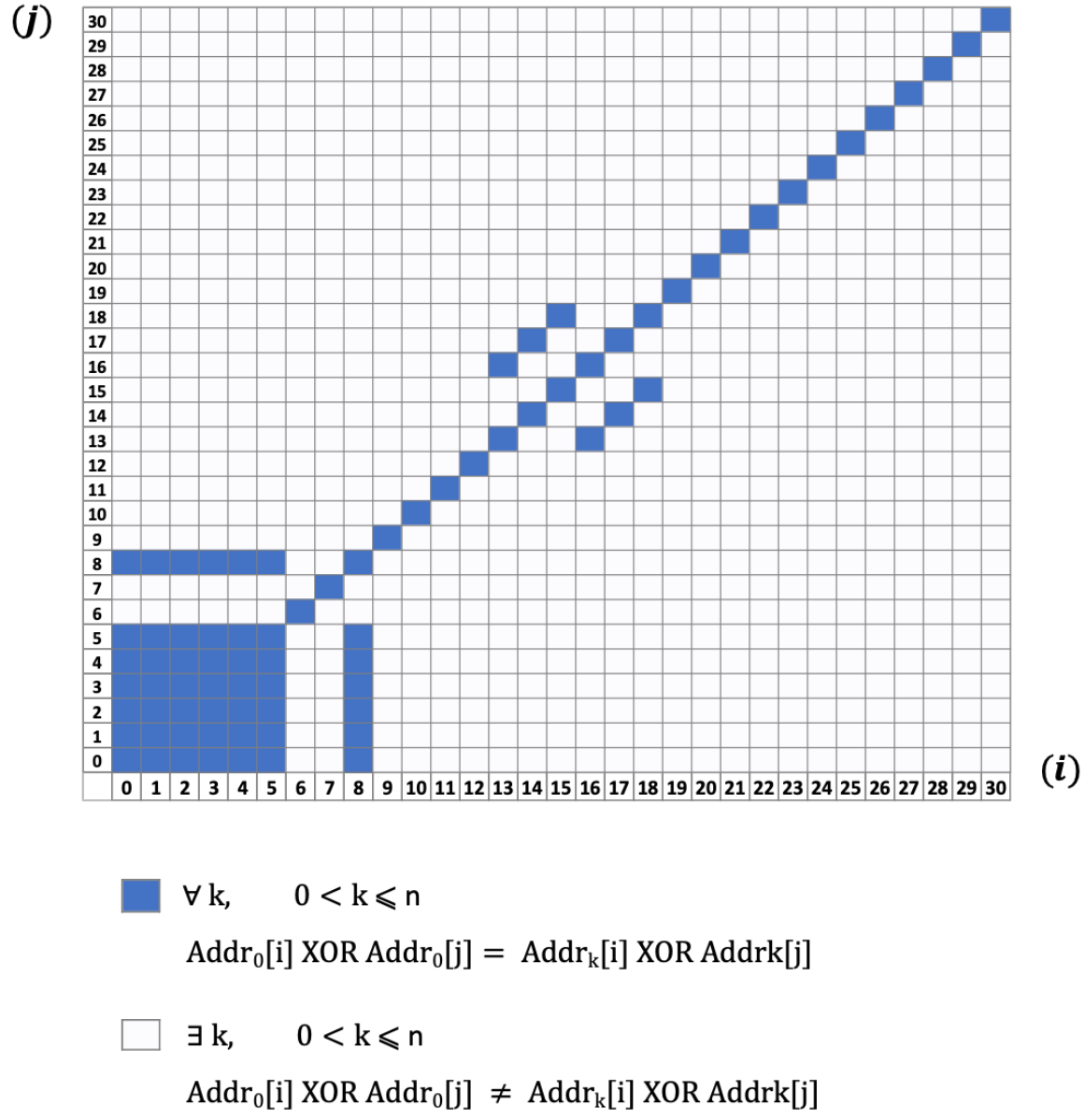

Figure 5.15: *Similarities between bit XOR combinations of the addresses present in the 138-cycle group of Figure 5.14. Experiments run on the MediaTek Helio X20.*

1. The resulting color matrix is diagonally symmetrical. This comes from the fact that the XOR operation is fully commutative. Also, the diagonal bits are XOR with

[3] 31 bits are needed to address the 2GB of DRAM memory in the MediaTek Helio X20 SoC

themselves, resulting in a 0 every time. Thus, 0 always matches with 0 for all the combinations.

2. All the combinations with the bits from 0 to 5 always match with the fixed address. That is because we generate the memory requests by loading the first word of the cache line. Consequently, the cache line offset, i.e., the six LSBs, is always identical.

3. When the bit 8 is `XOR` with one of the size first LSBs, the result is always the same. That means that for all the addresses in the 138-cycle group, bit 8 is always the same. Usually, as the LSBs change more often than the MSBs, the channel is mapped in the LSBs in order to get benefit of channel parallelism. Consequently, we deduce that bit 8 is used to select one of the two channels of the main memory.

4. The `XOR` results of the combinations (13, 16), (14, 17) and (15, 18) always match the results from the first fixed address. Therefore, we deduce that those three combinations of bits are used to map the eight banks of the main memory.

For the purpose of avoiding row-buffer conflicts, the rows are usually mapped on the MSBs. To take advantage of row locality, the columns are mapped with the LSBs. Considering that, we construct the corresponding address mapping that we expose in Figure 5.16. The first way to verify the address mapping is to see if we can reconstruct the organization of the main memory using only the address mapping. For instance, check that the number of rows multiplied by the number of columns equals the size of the LPDDR3_1886 bank. Once we finish this first quick verification, we can use the address mapping to generate many random address pairs that always trigger row-buffer conflicts. Then, we verify if the resulting average access time is always around 138 cycles. That way, we can effectively challenge the address mapping and guarantee its correctness.

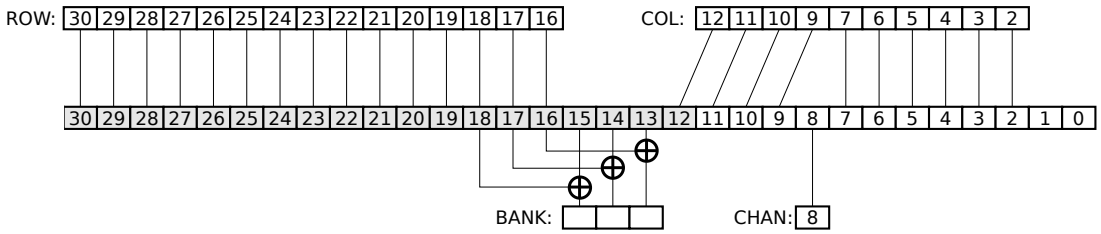

Figure 5.16: *Helio X20 main memory address mapping.*

In conclusion, we propose to improve the CalibratedV1 by replacing the SimpleMemory module with Ramulator, a very accurate main memory simulator. We calibrate Ramulator with technical information from MediaTek Helio X20 documentation. Then, we design a microbenchmark to expose the non-public address mapping of the

MediaTek Helio X20 main memory. Finally, we calibrate Ramulator with the resulting address mapping to finalize the last gem5 structure named CalibratedV2.

5.4 Experimental Evaluation

In this section, we evaluate both gem5 calibrations, i.e., the CalibratedV1 and the CalibratedV2, that we made in the previous Section 5.3. To this end, we first simulate our microbenchmarks to verify that the new parameter values lead to the expected behavior in the simulator. Once the calibrations are verified, we evaluate the instantiation of the methodology using benchmarks from the SPEC CPU2006 suite. We execute them on the MediaTek Helio X20 to get reference results. Later, we simulate them on gem5 models and compare the results with the ones obtained with the reference architecture.

5.4.1 gem5 Calibrated Models

In order to show how the simulator memory system behaves before and after the calibration, we simulate the microbenchmark described in Section 5.3.5. Thus, Figure 5.17 shows the result of the microbenchmark simulated on the Default gem5 configuration (refer to Figure 5.5 for details on the model). The parameter values for the Default configuration are listed in Table 5.1. We make three main observations. First, the L1 data cache and main memory access times are lower than those of the real hardware, while the opposite is for the L2 access time. Second, we find that the number of outstanding requests in the gem5 Default model is only two instead of the three indicated in the reference manual and confirmed by our microbenchmark. Third, the net increases in the average access time indicate an LRU replacement policy which is the one instantiated in gem5 HPI caches by default.

In order to validate the calibrated models, we also simulate the microbenchmark on them. After the first simulation, we observe some unexpected results. We inspect the simulator source code and we discover some incorrect behaviors in the HPI load/store unit: two processes, namely the operand forwarding and the idling state activation, are modeled in a way that obstructs the number of outstanding requests when we increase it from two to three. Accordingly, we turn off the idling state and disable the operand forwarding into the load/store unit. The modified parameters are listed in Table 5.1, which adds two new parameters to our list. Finally, we rerun the simulation to obtain the expected results. Figure 5.18 shows the microbenchmark simulation results with the CalibratedV1. We also run the microbenchmark with the CalibratedV2 to update

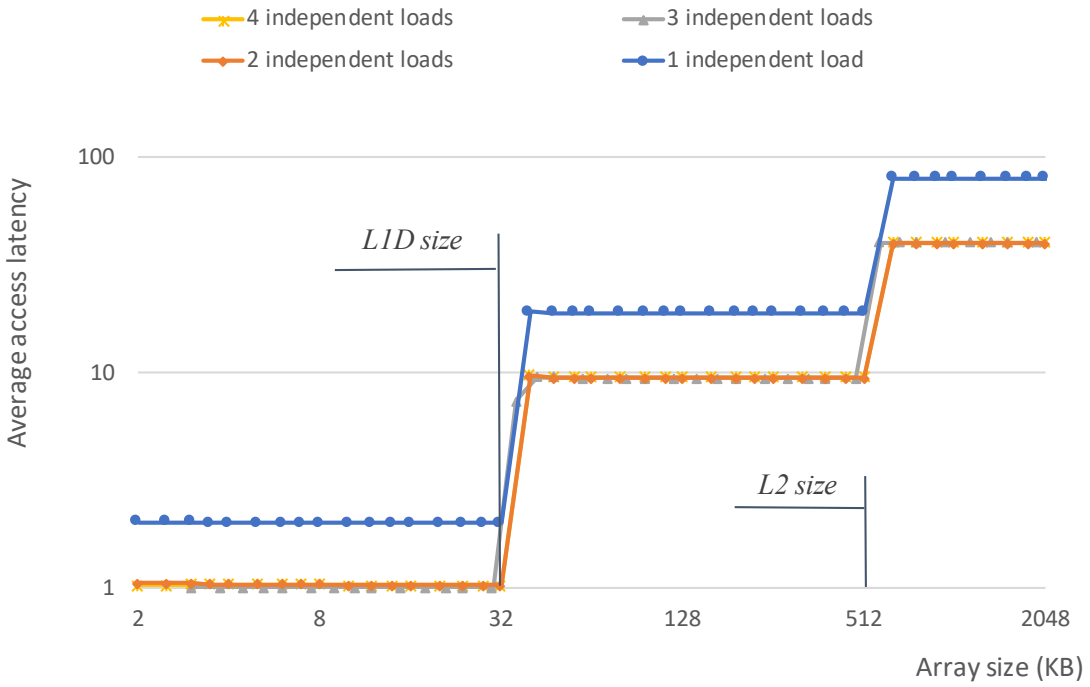

Figure 5.17: *Memory level signatures on gem5 of the Default HPI model configuration.*

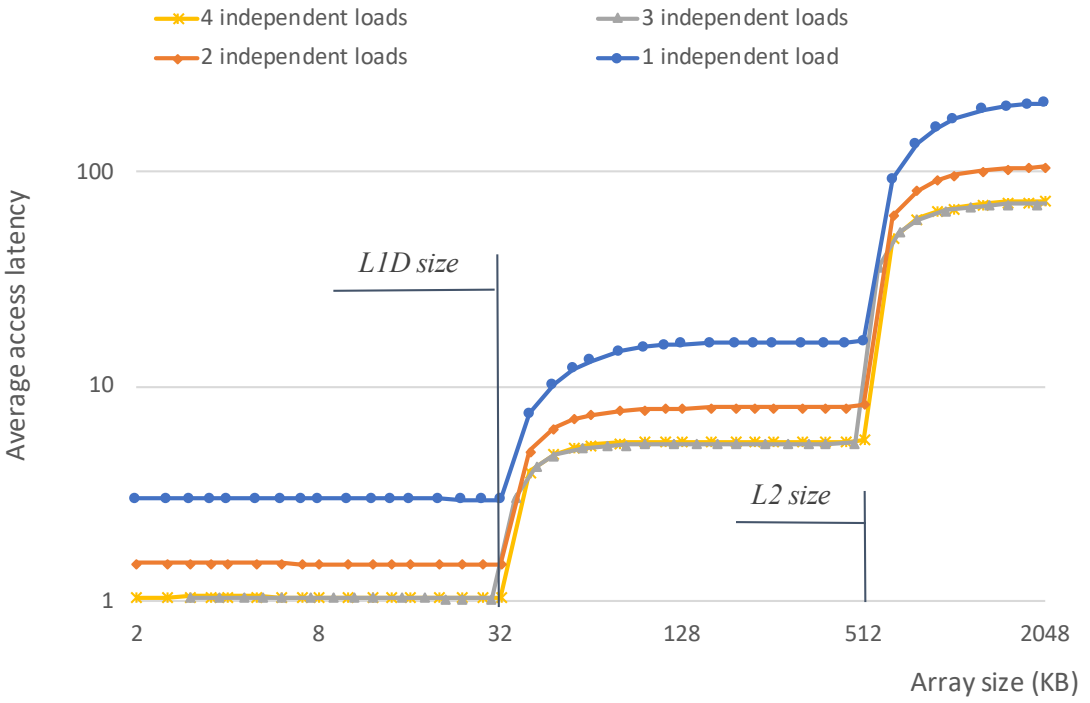

Figure 5.18: *Memory level signatures on gem5 of the CalibratedV1 HPI model configuration.*

the Bridge latency and obtain the same result as in Figure 5.18. We can see that the timings, the number of parallel memory accesses, and the replacement policies are very similar to the reference architecture (see Figure 5.8). However, due to the model limitation of the gem5 caches, we cannot replicate the contention measured on the real

L2 architecture cache. To reduce that error, we need to create a new contention-aware cache model, as in Evenblij et al. [151], which is considered future work.

The last model verification step is to run the address mapping benchmark on the CalibratedV2 simulation model. This way, we verify that the address mapping is rightly implemented in Ramulator and the proper integration of Ramulator in gem5 simulations. Furthermore, we can verify that the latency of a row-buffer conflict corresponds to the one obtained with the real reference architecture.

5.4.2 Evaluation of Simulation Accuracy

To evaluate the methodology, we use twenty benchmarks from the SPEC CPU2006 suite and the number of Instructions Per Cycle (IPC) metric. We first execute them on the MediaTek Helio X20 to generate the reference results. Then, we simulate the benchmarks on the three gem5 models, i.e., Default, CalibratedV1, and CalibratedV2. For that, we follow the standard SimPoint methodology [108] with 100 million instructions per slice and max K equal to 30. We plot the results in Figure 5.19. In order to better exhibit the error, we plot in Figure 5.20 the absolute error of each benchmark normalized by the results from the reference CPU. We also calculate the average absolute IPC error and the maximum absolute IPC error. We can see that the CalibratedV1 and CalibratedV2 reduce the average error by 39.6% and 43.3%, respectively. The CalibratedV1 reduces the maximum error by 61.5%, and the CalibratedV2 reduces it by an extra 62.5%.

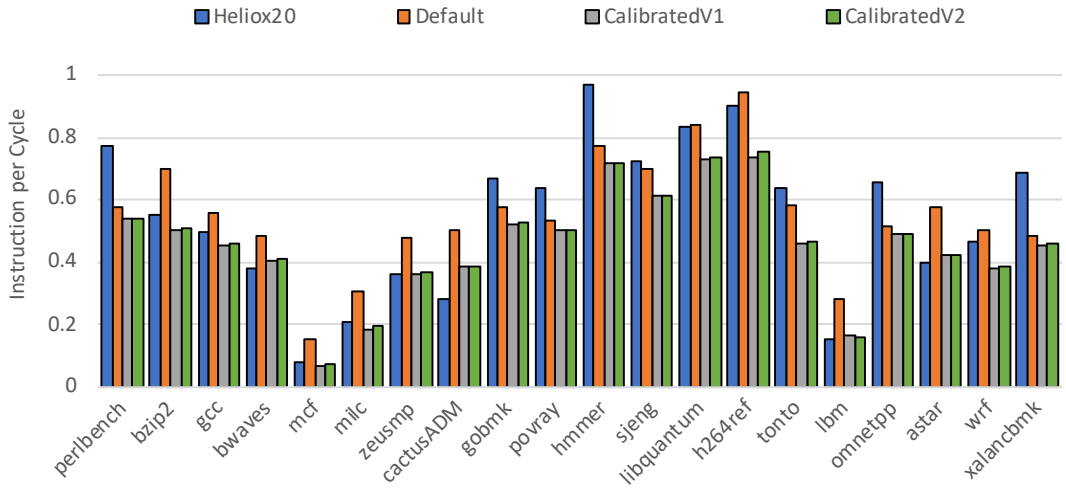

Figure 5.19: *IPC for the SPEC CPU2006 suite executed on the MediaTek Helio X20 and simulated on Default, CalibratedV1 and CalibratedV2 gem5 configurations.*

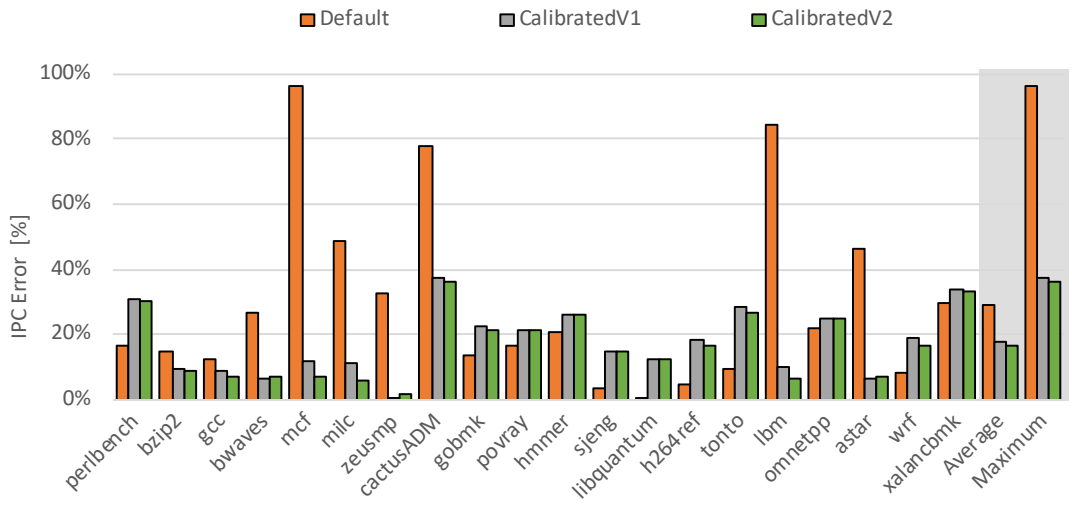

Figure 5.20: *Normalized IPC error of the Default, CalibratedV1 and CalibratedV2 gem5 configurations with respect to reference hardware, the MediaTek Helio X20.*

Table 5.3: *Cumulative simulation time of CalibratedV1 and CalibratedV2 gem5 configurations.*

gem5 Configurations	seconds	hours
CalibratedV1	463,311s	128h
CalibratedV2	739,159s	205h

Table 5.3 shows the simulation time of the two calibrated configurations, i.e., CalibratedV1 and CalibratedV2. We can observe that the CalibratedV2 model, which uses Ramulator to model the main memory, increases the simulation time by 60%. This overhead in simulation time reduces the absolute average IPC error by 6.2% and the maximum absolute IPC error by 2.5% between CalibratedV1 and CalibratedV2.

5.4.3 Analysis

With the evaluation of the methodology, we observe that we have reached our objectives of reducing the modeling error due to model parametrization (see Section 3.1. Consequently, we show the impact of the reactive memory system on global system performance. For instance, the big difference in main memory access time between the reference MediaTek Helio X20 (208 CPU cycles) and the Default gem5 configuration (80 CPU cycles) may explain the big IPC errors on Figure 5.20. For the five biggest IPC errors, i.e., mcf, milc, cactusADM, lbm and astar. We observe that the IPC is higher for the Default gem5 configuration than for the MediaTek Helio X20. With CalibratedV1 and CalibratedV2, we can see that for those five benchmarks, the IPC is now lower, showing that the memory system is slowing down the global performance. For four

of the five benchmarks, the IPC error is now under 12%, which is sufficiently accurate to perform proper design space exploration. Only cactusADM still has an error of 37% reducing already by 52% the average IPC error between the Default and the CalibratedV1 configurations.

The evaluation results also show the impact of using a more accurate model. CalibratedV2 is an upgrade of the CalibratedV1 configuration. We replace the main memory model of the CalibratedV1 with Ramulator, an accurate main memory simulator. We observe in Figure 5.20 that the main memory model upgrade still reduces the average and maximum IPC error by 6.2% and 2.5% respectively while the simulation time increases by 60%. This illustrates how important it is to adapt the modeling accuracy of the simulator to the type of application being considered. For instance, with only one core and applications with a residual main memory activity, a detailed main memory model may not add extra accuracy while incurring a longer simulation time. On the contrary, the calibration of the load/store unit, caches, and interconnect latency would have a larger impact on the simulation results' precision, as shown.

Even after the implementation of the methodology, we can still observe a modeling error. So for a few benchmarks, we are still well above 10% error. We observe that some error comes from modeling error, i.e., the model is not accurate enough. For instance, in Section 5.4.1, the L2 cache model cannot model the contention measured on the real L2 architecture cache. Some error also comes from no-modeled processes like the full virtual to physical page translation process or the page fault effects. Also, the reactive memory system elements as the L2 cache and the main memory, are shared with other components generating conflicts and the need for cache coherence protocols. Additionally, the SimPoint methodology we use to reduce the simulation time brings modeling unrelated to the target architecture.

Furthermore, the data prefetcher is a component of the memory system, part of the proactive memory subsystem. This component plays a key role in memory system data movement, as explained in Section 2.2.4. The gem5 HPI model includes a data prefetcher model, but the proposed methodology cannot be implemented for such component calibration. Hence, we propose in Chapter 6 a dedicated methodology to reveal the functional behavior of commercial reference CPUs.

5.5 Summary

To conclude, this chapter proposes an instantiation of the methodology to calibrate the reactive memory system of computer architecture simulators. For that, we first detail the composition of the reactive memory system. Then, we propose a delay model to represent how memory requests accumulate delay traveling through the reactive memory system. We use the delay model to describe the different scenarios we target during microbenchmark design.

In Section 5.3, we detail the design of the microbenchmark depending on the memory system parameters we target, e.g., the cache access times, TLB penalty latency, or the main memory address mapping. We use the results from the microbenchmarks of Subsection 5.3.5 and Subsection 5.3.7 to create a first calibrated configuration of the simulator we call CalibratedV1. We verify the memory system behavior of CalibratedV1 by simulating the microbenchmark and comparing the output signature with the reference one in Section 5.4.1. Additionally, we upgrade the CalibratedV1 configuration with an external main memory simulator, i.e., Ramulator, to fully model a working main memory. We call this configuration CalibratedV2.

Finally, in Section 5.4.2, we evaluate the methodology with benchmarks from the SPEC CPU2006 suite and the IPC metric. We execute them on the real reference CPU and both gem5 configurations. We observe that both CalibratedV1 and CalibratedV2 reduce the average IPC error by 39.6% and 43.3%, respectively, and the maximum error by 61.5% and 62.5%, respectively. However, Ramulator increases by 60% the simulation time by partially reducing the average and maximum IPC error. From these results, we can draw the following conclusions. First, the methodology that we propose reduces the model parametrization error. Second, the trade-off between accuracy and simulation time exists for each simulator component and needs to be properly decided depending on the simulated system.

VI

Data Prefetching Functional Model

In this chapter, we propose Pref-X, a framework to analyze functional characteristics of data prefetching in commercial in-order cores. Data prefetching is a memory process often not documented by the industry. Consequently, we propose our framework to reveal data prefetching by *X-raying* the cache memory and exposing changes made by the data prefetching. For that, Pref-X instantiates the methodology described in Chapter 4. Thus in a phase, we purposely design microbenchmarks to deduce data prefetching and create a representative prefetcher functional model. Then in a second phase, we verify this functional model using memory traces extracted from realistic benchmarks. To demonstrate the feasibility of this methodology, we implement Pref-X on two ARM in-order cores, the Cortex-A7, and the Cortex-A53. From that, we create two functional data prefetcher models that we evaluate using memory traces extracted from the SPEC CPU2006 suite.

6.1 Background and Motivation

One way to reduce the data average access time is to maximize the number of memory requests that produce a hit in the L1 data cache. That way, the memory requests issued by the core do not travel through the memory hierarchy's lower levels, delaying the access time. Section 2.1 gives more details about how the memory hierarchy works. With the goal of maximizing the L1D hit rate, the L1 data prefetcher monitors the core memory activity and predicts addresses that the core could request in the future. Then, the predicted addresses are fetched into the L1 data. If the prediction is correct, the L1D hit rate increases. If not, the L1D is polluted by useless addresses, and the L1D hit rate decreases. Section 2.2.4 explains in more detail the role of the data prefetcher and different implementation issues.

6.1.1 Stride Prefetchers

By not re-ordering the program instructions, the execution of a program on an in-order core generates predictable memory sequences, which facilitates the task of data prefetching. A common data prefetcher category usually implemented in commercial in-order cores is the *stride prefetcher*. A stride prefetcher uses a simple prediction process that inspects the distance between addresses. If a recurrent distance is detected between addresses, the prefetcher uses the pattern to predict future addresses. We call the fixed distance between the addresses the *stride*. We call *stream* the sequence formed by consecutive addresses with the same stride.

Stride prefetchers implement multiple features, such as the maximum stride length of a stream or the maximum number of streams that can be detected in parallel. Intending to capture the diversity in practical stride prefetchers, we propose a meta-model including the following list of parameters:

- `Trigger inputs`: The inputs of the prefetcher that are used to trigger a prefetching process. The prefetcher can monitor events such as misses or hits in the L1D. The monitored addresses can be either in the virtual or the physical address space.

- `Initial trigger conditions`: Conditions on the inputs to trigger the prefetching process. For instance, the number of misses on the same stream that are necessary to trigger a prefetching process. There can be multiple conditions based on the type of inputs, i.e., hit or miss,

- `Burst length`: The burst length corresponds to the number of prefetches generated by one trigger. This length can change depending on the trigger condition.

- `Maximum stride length`: The maximum stride in a stream that can be detected by the prefetcher.

- `Conditions to continue`: The conditions to generate additional bursts from an already detected stream. For instance, the prefetcher can generate more prefetches if the memory requests from the core hit the previous prefetched addresses, i.e., if the program execution validates the prefetcher predictions.

- `Burst hitting in L1D`: Define behavior in case the prefetcher generates prefetches that hit in the L1 cache.

- `Tracking across pages`: Define if the prefetcher is able to detect and continue streams that expand over one page.

- `Maximum number of streams`: The maximum number of interleaved streams the prefetcher can track simultaneously.

- `Conditions to stop`: The condition for a stream to stop being tracked by the prefetcher. For instance, if the condition to continue is no longer valid or a new trigger condition appears, stopping the previous tracking.

6.1.2 Motivation

The gem5 simulator implements by default a stride prefetcher in the High Performance In-order core model (HPI) described in Subsection 2.3. We are interested in understanding how this prefetcher compares with a real one. Hence, we use the microbenchmark described in Section 5.3.5 (with only one independent load) to compare the behavior of the gem5 and the Cortex-A53 stride prefetchers. The microbenchmark accesses all the addresses of an array, which gradually increases its size to drive the data accesses to the different levels of the memory hierarchy. We create two versions of the microbenchmark. The first one, *random accesses*, is the original access pattern using random address accesses in the array to avoid data prefetching. The second one, *consecutive accesses*, is a new access pattern using sequential address accesses. That way, the memory sequence generated by the microbenchmark is easy to predict by a stride prefetcher. We execute the microbenchmark on the Cortex-A53 of the MediaTek Helio X20 SoC that implements a stride prefetcher [152]. And, simulate it with the *CalibratedV1* gem5 configuration created in Section 5.3.5. We plot the results in Figure 6.1.

We identify three different regions on Figure 6.1. The first region is from 2KB to 32KB. In this region, the full array can be stored in the L1D. Thus, there is no need for data prefetching. The microbenchmark gives the same average access time for the four executions. The second region is between 32KB and 512KB. We can observe with the average access time increase that the arrays is now too large to be fully stored in the L1D. For the MediaTek Helio X20 and the gem5 simulation, we observe that the consecutive accesses versions have a lower average access time. This reflects a data prefetching activity that reduces the average access time. We also observe that the curve from the MediaTek Helio X20 is under the one from CalibratedV1. Meaning that the data prefetching of the MediaTek Helio X20 in this region provides a better performance. I.e., the gem5 stride prefetcher reduces the average access time from 16 cycles to 10 cycles while the MediaTek Helio X20 data prefetcher reduces it to 9 cycles. The last region is after 512KB, the array is now stored in the main memory. Again, we can observe that the data prefetcher of the MediaTek Helio X20 is more efficient as it divides the average access time by $5\times$ while the gem5 prefetcher only by $2\times$.

We conclude that the current gem5 data prefetcher can incur large modeling errors when modeling a real architecture such as the MediaTek Helio X20 shown here. Consequently, our goal in this chapter is to propose a method to unveil the key mechanism in existing stride prefetchers to improve accuracy in the simulation of existing architectures.

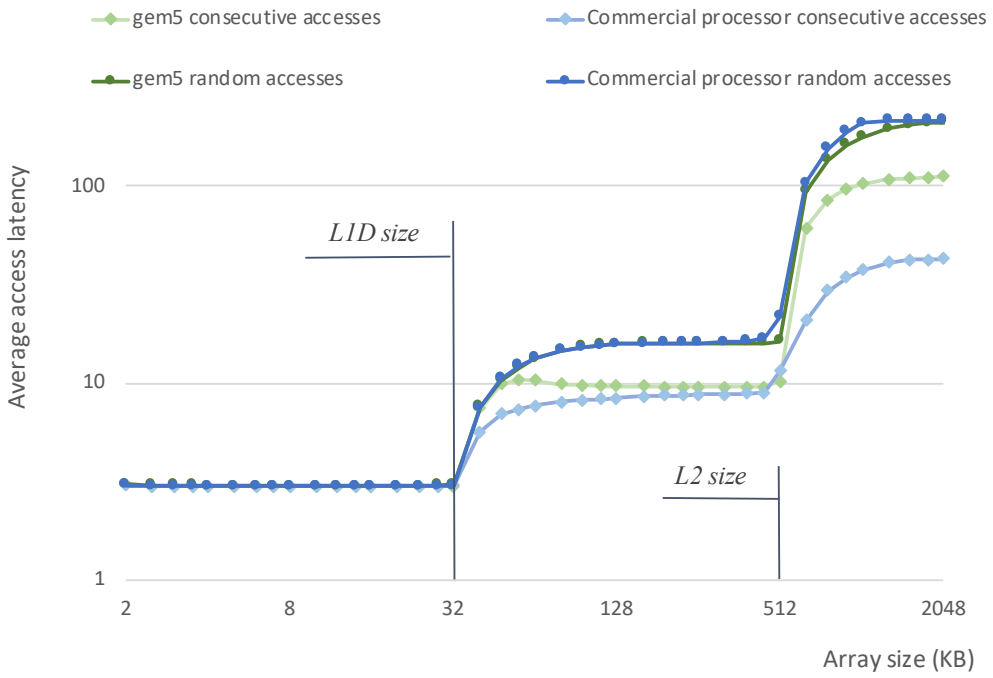

Figure 6.1: *Microbenchmark results for consecutive and random accesses to the array, simulated with CalibratedV1 gem5 configuration and executed on oneCortex-A53 of the MediaTek Helio X20 SoC.*

6.2 Key Insights

In this section, we give some insights about how we can deduce data prefetching by exposing changes in the L1 data cache. For that, we need first to monitor changes in the L1D and find which one comes from data prefetcher. Then, we need to identify the memory request sequence that has generated the data prefetching.

6.2.1 Exposing Data Prefetching

In order to deduce L1D content, we use the hardware performance counters to count the number of L1D accesses and misses. Figure 6.2 represents a typical CPU with the load/store unit, the PMU, the L1D and L1 data prefetcher. Outside the CPU we have the rest of the memory hierarchy. The first step is to flush the complete L1 data cache. That way, one addresses can only be present in the L1D if this address has been requested by the core or has been prefetched. Then, we reset the performance counters and generate a memory request sequence ① to trigger the prefetcher. In this example we have three memory requests, i.e., 0, 2 and 4. As we have a cold L1 data cache, all the addresses miss in the L1D and go to lower levels ②. Now, we want to know if

the L1 prefetcher has generated any prefetch following the stride of 2 in the memory sequence ③. For that, we access the next address in the stream ⑤. In this example, that is the address 6. We call this address the inspected address. Finally, we use the results from the performance counters⑥ to deduce if the inspected address has been prefetched. We have three possible scenarios:

1. There are as many misses than accesses, i.e., the inspected address is missing in the L1D. This address is not part of the memory sequence and has not been prefetched.
2. The inspected address is not missing in the L1D because the address is part of the memory sequence.
3. The inspected address is not missing but the address is not present in the memory sequence. In this case, the inspected address has been prefetched.

The address inspection can be done of any address of the L1D. Thus, depending on the monitored scenario, we can determine if the memory sequence trigger or not the prefetcher and deduce the prefetched addresses.

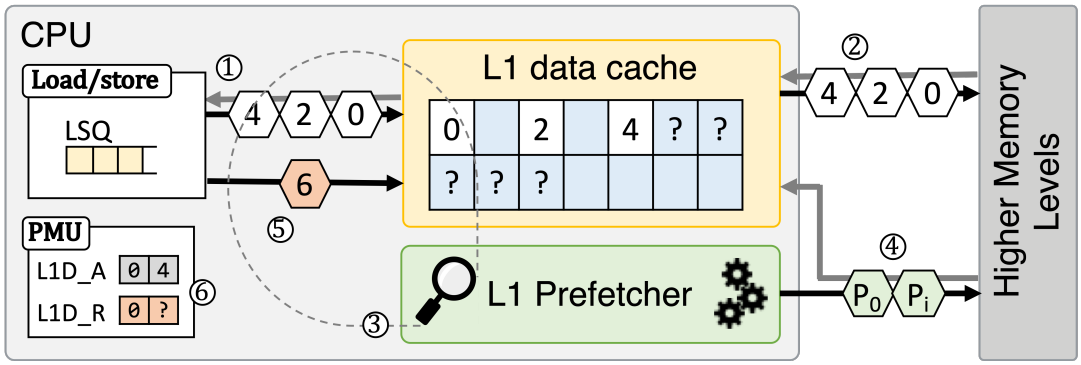

Figure 6.2: *Helio X20 main memory address mapping.*

6.2.2 Identifying the Trigger of a Data Prefetch

In order to deduce the complete functional behavior of the data prefetcher, we develop a technique based on the approach exposed in Figure 6.2. We first define a fixed area in the address space we call *memory zone*. Then, we generate a memory request sequence that belongs to the memory zone and we inspect one address. We iterate the operation and inspect all the addresses of the memory zone. Meaning that, for each address k of the memory zone, we flush the memory zone, we generate the memory sequence and we inspect the address k. That way, for a defined memory zone and memory sequence, we are able to know which address is present in the L1D cache due to a data prefetch.

Figure 6.3 shows a example result of the execution of the technique on the Cortex-A53. We define the memory zone as one page including 64 cache lines. We stress the memory zone with the memory sequence ⟨0, 2, 4, 6⟩. We gradually build-up the memory sequence by adding the memory request one by one starting from an empty sequence as it shows in the first row of Figure 6.3. That way, we can identify how each request triggers or not the data prefetcher. Thus, we observe at length 3 that we need up to three consecutive misses to trigger a prefetcher burst of three cache lines, ⟨6, 8, 10⟩. By adding another address to the sequence, i.e., length with the address 6, we also observe that a hit in a previous prefetched address generate another prefetcher burst of three cache lines, ⟨12, 14, 16⟩. The way how we generate Figure 6.3 implementing the technique is fully explained in Section 6.4.

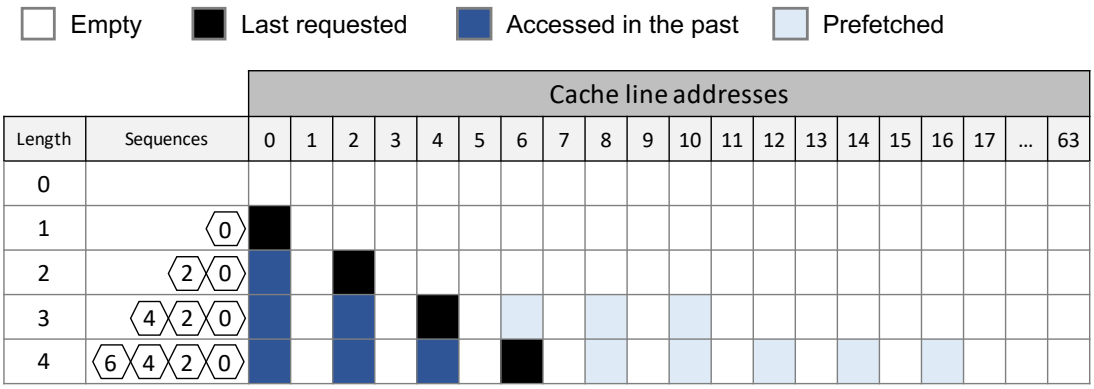

Figure 6.3: *L1D cache content when accessing memory sequence⟨0, 2, 4, 6⟩.*

6.3 The Pref-X Framework

In this section we introduce our framework, Pref-X. The tool is divided in two phases: the *Reconstruction phase* and *Verification phase*. The first phase aims to create a first functional model version of the target commercial CPU. Then, with the second phase, we verify the resulting functional model with real memory sequences extracted from SPEC CPU2006. That way, we stress the functional model with a different wide range of memory sequences exposing miss matching behaviors that we miss in the first phase.

6.3.1 Reconstruction Phase

The goal of the reconstruction phase is to create a functional model of the target commercial in-order CPU data prefetcher. We represent this phase and its different steps in blue on Figure 6.4. We start with a data prefetcher meta-model that includes the

common features of stride prefetcher described in Subsection 6.1.1. We use the meta-model to design synthetic memory sequences to expose the main feature parameters. For instance, we can determine the max stride the prefetcher can handle by designing a list of sequences with increasing stride values. That way, we start from the lowest stride value and increase it till the prefetcher can no longer detect the pattern which indicates the maximum stride value.

To execute the memory sequences on the commercial CPU, we use the *Prefetcher inspector*, which is a subtool of Pref-X. This subtool comprises two elements: the sequence runner executes the synthetic memory sequences on the commercial CPU. And the graph generator uses metadata generated by the sequence runner to generate the graphs as described in Subsection 6.2.2. We use the graph to see the data prefetching activity and extract the main feature parameter values, e.g., the maximum stride value. Then, we use those results to complete the meta-model and obtain a complete functional model of the target commercial prefetcher.

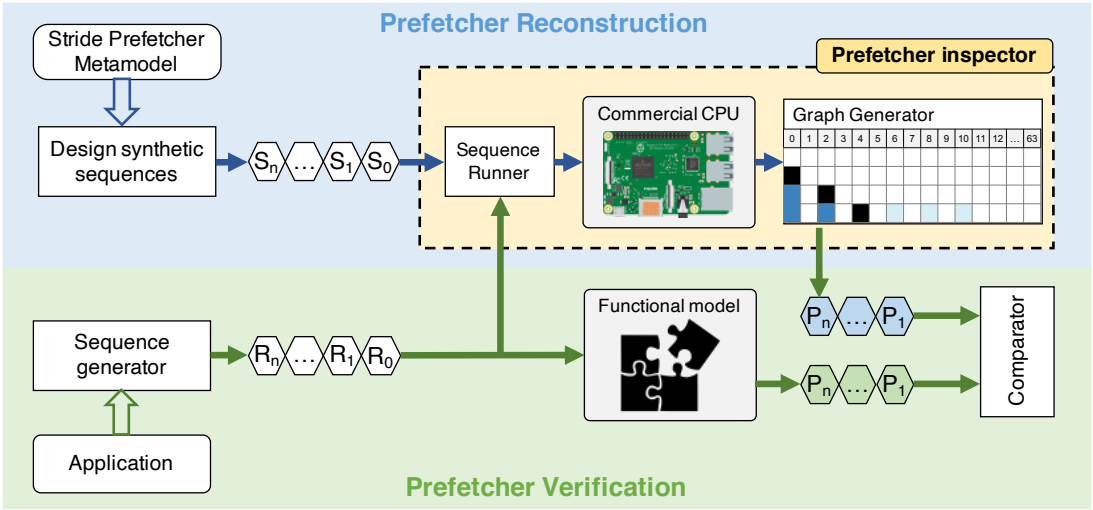

Figure 6.4: *Pref-X phases illustration.*

6.3.2 Verification Phase

We use the verification phase of Pref-X to verify the functional model created in the reconstruction phase. We represent this phase and its steps in green in Figure 6.4. Contrary to the first phase, we use realistic memory sequences that we extract from real applications. Those memory sequences are generated from real applications using a sequence generator. For instance, we can use programs such as the gem5 simulator or the Pintool from Intel to extract memory traces that we further split into smaller memory sequences. The memory sequences are executed on the functional model and the

commercial board with the prefetcher inspector. We compare each memory sequence results from both the real CPU and the functional model are the same.

If we detect meaningful differences, we identify the problematic memory sequences, draw new insights, and correct the functional model. Creating new synthetic memory sequences might be necessary to validate the new insights properly. We iterate this operation to converge to an accurate functional model. Hence, if we do not challenge the meta-model enough with synthetic memory sequences during the reconstruction phase, the verification phase would need many more iterations. Once the desired accuracy is obtained, the execution of Pref-X and the functional are finished.

6.4 Prefetcher Inspector

.

The prefetcher inspector is a subtool of Pref-X. It contains two different elements, the sequence runner and the graph generator. The sequence runner takes memory sequences as input. Then from one sequence, it generates a list of sub-sequences that it executes on the commercial CPU. It monitors memory system events using the performance counters of the PMU and generates output metadata files. Then, the second element, the graph generator, uses the metadata generated by the sequence runner to create the graph introduced in Subsection 6.2.2.

6.4.1 Sequence Runner

The objective of the sequence runner is to generate from a memory sequence the metadata necessary to build the corresponding request-by-request representation graph of the L1D memory zone content. Figure 6.5 represents the different phases of the sequence runner. The first phase divides the input sequence of N memory requests into $N + 1$ sub-sequences starting from an empty sub-sequence and incorporates one address at a time. We call Seq_n, the sub-sequence of length n such that $0 \leq n \leq N$. Figure 6.5 illustrates the process for the sequence$\langle$0, 2, 4, 6$\rangle$that is used in Section 6.2.

Then, the sub-sequences are sent to a microbenchmark executed on the commercial CPU. The goal of the microbenchmark is to execute the sub-sequence Seq_n and inspect the addresses of the defined memory zone. We call M the number of addresses in the memory zone, i.e., the number of cache lines. Thus, for k such that $-1 \leq k < M$, the microbenchmark executes the sub-sequence Seq_n and then inspect the address k. We

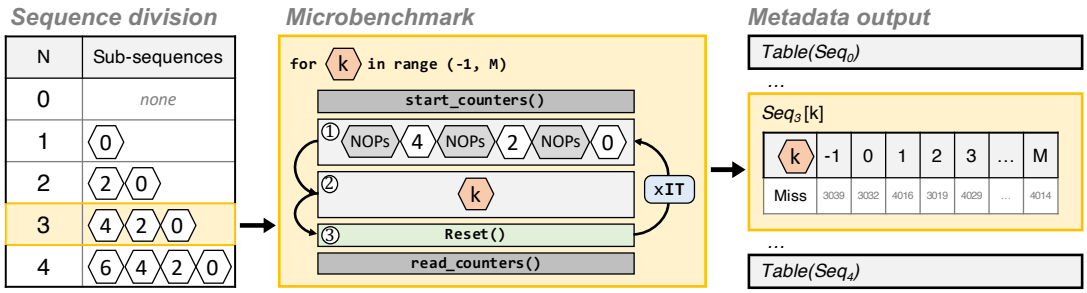

Figure 6.5: *Prefetcher inspector: sequence runner phases illustration.*

define k equals -1, the case where we do not inspect any address. This case is used as a reference when we generate the graph, as explained in the next Section 6.4.2. The microbenchmark follows the structure introduced in the methodology in Chapter 4 with the initialization and the measure loop. The measure loop comprises three steps that are iterated IT times to reduce the noise-to-signal ratio. The measure loop is executed with M different values of k, for each sub-sequence Seq_n.

① The microbenchmark loads the addresses present in the sub-sequence one by one. Those addresses are relative to the determined memory zone. For instance, the address 0 corresponds to the first cache line and 1 to the second cache line. We implement the memory zone as an array and align them as the first element of the arrays corresponds to the first cache line of a page. Then, we add to the array a pointer that goes through the addresses of the sub-sequence. As a cache line contains multiple words, we use the first one to implement the pointer chasing as illustrated in Figure 6.7. Finally, we add a lot of `NOP` operations between loads in each step1 of the measure loop to avoid any timing interference. We illustrate the code source on Figure 6.6.

② The second step consists of loading the inspected address. For that, we extend the pointer chasing of the step ① to load the address k. We use the second word of the cache line to be sure we do not overwrite an already-used array cell. The final pointer chasing is illustrated in Figure 6.7. In case of k is equal to -1, we just do not add any address, i.e., the microbenchmark does not execute the step 2.

③ The goal of step 3 is to reset the cache data prefetcher states before the next iteration. At the beginning of each iteration, we hypothesize that we have a cold L1 data cache, and the prefetcher cannot remember the previous iteration. Contrary to the other steps, this depends on the data prefetcher. For instance, we can use `flush` operations to reset the cache and stop active data prefetching. Another method would be using a different address region with the same relative memory zone and sequence for each iteration. That way, the addresses would be different for each iteration. The data cache and the prefetcher could not use previous data from previous iterations. Those examples are more detailed in the implementation Section 6.5.

```
long long int array[M * S]; // S: number of lld per cacheline
long long int *ptr;

init_hw_perf_counters();  // Initialization of the HPCs
for(int n = 0; n <= N; n++){
    for(int k = -1; k < M; k++){
        // INITIALIZATION
        init_step1_chasing(array, subSeq);
        init_step2_chasing(array);
        ptr = &array[subSeq[0]]; // Init 1st step1 pointer
        // MEASURED LOOP
        start_hw_perf_counters(); // Reset HPC
        for( int = it; i < IT; it++){
            // Step1: Execution of the sub-sequence
            for(int s1 = 0; s1 < n; s1++){
                ptr = *ptr1; NOP_500;}
            // Step2: Inspect address k
            if(k != -1){
                ptr = *ptr;}
            // Step3: Reset cache and prefetcher states
            reset();
        }
        read_hw_perf_counters()(); //Read HPC
    }
}
```

Figure 6.6: *Sequence runner microbenchmark C code.*

During the execution of the measure loop, we monitor the number of misses. In this manner, for each sub-sequence Seq_n, we have M measures for each value of k. We call $Seq_n[k]$ the number of misses of the sub-sequence Seq_n when we inspect the address k. The Graph generator uses those metadata to build the final graph, see Section 6.4.2.

6.4.2 Graph Generator

The graph generator composes the second phase of the prefetcher inspector. We use the metadata provided by the sequence generator to build an output graph like the one introduced in Section 6.2.2. Each row corresponds to a Sub-sequence, and each column to a cache line of the memory zone. We build the graph row by row, starting with the first sub-sequence Seq_0. Figure 6.8 shows the complete set of metadata generated at the first phase with the sequence$\langle 0, 2, 4, 6\rangle$and one thousand iterations of the measure loop steps ($IT = 1000$). The values are not normalized by the number of iterations and show very small variations due to the noise. For each sub-sequence, $Seq_n[-1]$ gives the number of misses that come from step ① as we do not inspect any address. For $Seq_n[k]$

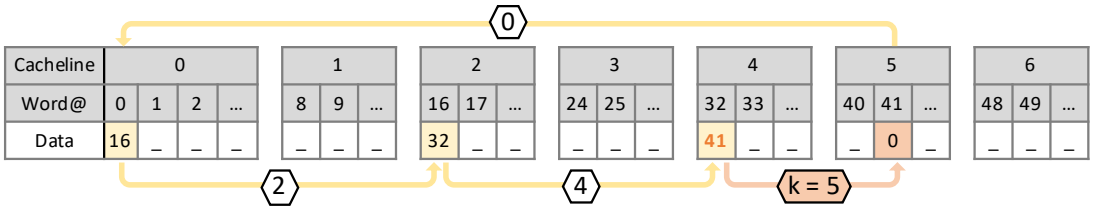

Figure 6.7: *Sequence runner pointer chasing for memory sequence ⟨0, 2, 4⟩and the inspection of the address 5.*

such that $0 \leq k < M$, we identify two cases:

- The address **k** is **not present** in the L1D. In this case, the execution of the step ② adds one miss at each iteration, i.e., it adds around IT miss to the reference value $Seq_n[-1]$
- The address **k** is **present** in the L1D. In this case, the execution of the step ② does not add any misses and the number of misses is close to the reference value $Seq_n[-1]$

Consequently, for each number of misses, we expect the number to be close to an integer value of IT. In our example, an integer value of a thousand. If the results are not close to an integer value of IT, that may indicate the number of iterations (IT) is too low, or the step ③ does not work as expected. Thus, we can define an acceptable distance to an integer value. If a value is floating between two integer values, that triggers an interruption in the graph generator execution.

Figure 6.8 uses the same color legend as the Figure 6.3. Thus, we verify that for the addresses which are present the sub-sequence, i.e., black and blue, have a number of misses that is close to the reference in grey. We also observe that the sub-sequences Seq_3 generates the prefetching of the addresses 6, 8 and 10. Hence, the sub-sequence Seq_4 generates one hit in the step ①. We confirm this with the value of $Seq_3[-1]$ and $Seq_4[-1]$ which are very close. Finally, we deduce that Seq_4 generates the prefetching of the addresses 6, 8, 10, 12, 14 and 16.

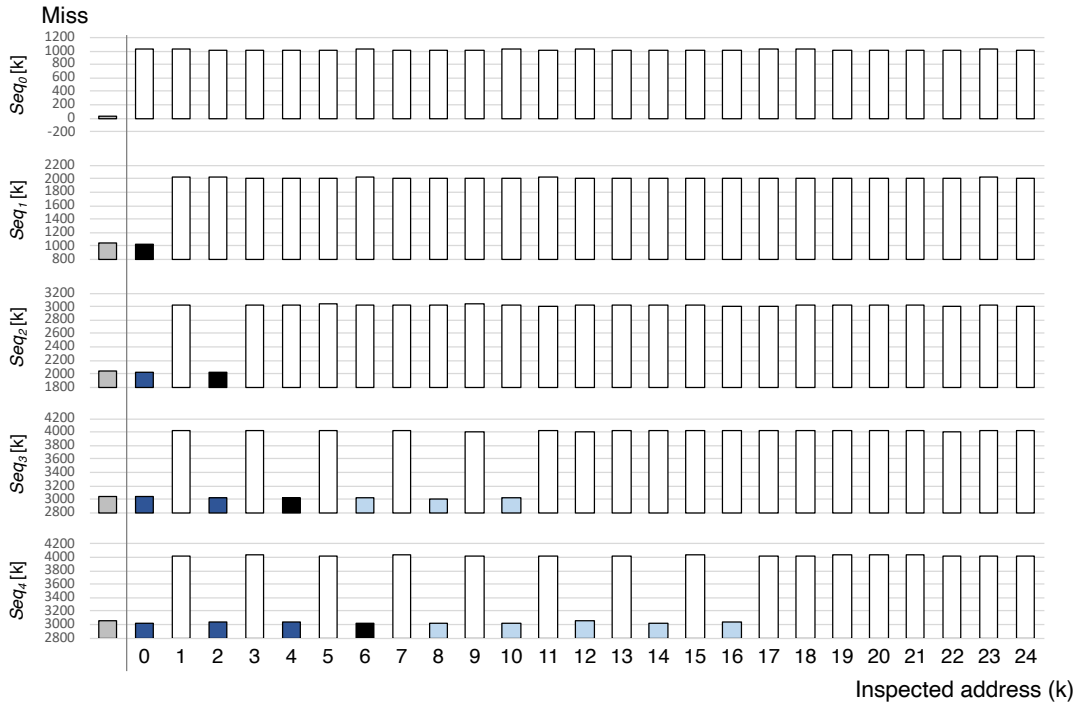

Figure 6.8: *Metadata generated by the sequence runnner, proccessed by the graph generator.*

6.5 Implementation

In this Section, we use Pref-X to reveal the functional behavior of the Cortex-A7 [61] and the Cortex-A53 [148] in-order ARM CPUs. We use two development boards, the Raspberry Pi 2B [153] and the Raspberry Pi 3B+ [147]. We start with the reconstruction phase and illustrate the process with some synthetic sequence examples.

6.5.1 Memory Zone

Before starting the first phase of Pref-X, we must define the memory zone first. This one must be large enough to contain the memory sequences and the generated prefetches we want to expose. However, if the memory zone is too large, some addresses could be evicted from the L1D due to the replacement policy and the size/associativity of the cache.

Thus, we start the process by examining the L1 data cache organization. Both Cortex-A7 and Cortex-A53 L1 data cache implement the same pseudo-random replacement policy. Also, both caches have the same organization, 4way associative with a size of 32KB, which means that the cache is divided into 128 sets containing 4 tags each. Due to the random replacement policy, the only way to avoid evictions is to define the memory zone so that all its cache line addresses use each set of the cache only

once. I.e., we need all the cache line addresses present in the memory zone to have different cache indexes. Figure 6.9 shows how the address bits are mapped in the cache: the six Least-Significant Bits (LSBs) address the byte within the 64B cache line, the next seven bits index the cache set, and the remaining bits compose the tag. The page offset remains the same between the virtual and the physical address space. However, the page number depends on the physical address translation. That way, by limiting the memory zone to one page, we ensure that 64 cache lines are mapped to 64 different indexes.

In order to increase the memory zone, we disable the Address Space Layout Randomization (ASLR) feature of the kernel. This way, the operating system maintains adjacency between pages after the address translation. We can guarantee that two consecutive pages will have a different page number LSB in the physical address space. Consequently, we define the memory zone as two pages with 128 cache lines mapped on 128 different cache indexes. Three pages would be too large to guarantee no evictions and one page too small to expose data prefetcher behavior across multiple pages.

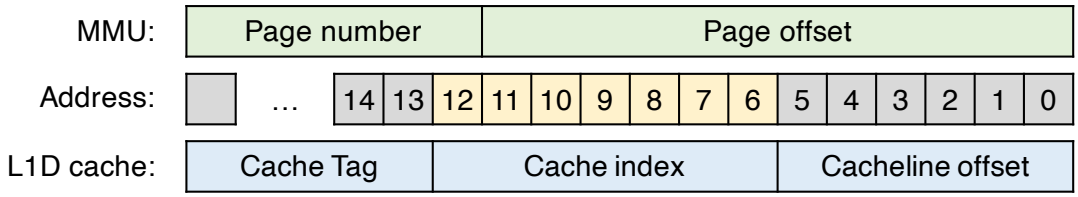

Figure 6.9: *Address mapping L1D of Cortex-A7 and Cortex-A53.*

6.5.2 Reset Method

The reset method used in the step ③ in Subsection 6.4.1 needs to be defined. The goal is to make each iteration of the measure loop independent of the other. For that, we propose to use multiple regions of the address space. We define the memory zone and the memory sequence relatively to a region. For instance, we define the memory zone as page 0 and page 1 of each region. The address 0 of a memory sequence is the first address of the region. In this manner, the absolute physical addresses change at each iteration. Additionally, we can add random accesses between iterations to fill internal prefetcher registers with unrelated addresses before the next iteration. This is only useful if the prefetcher can track a pattern across pages.

Figure 6.10 shows the final pointer chasing. We use 128 different regions of 8 pages. The first two pages are the memory zone. The six others are used to do random accesses. The random sequence is different for each iteration. We start without random accesses and update the number further in the reconstruction phase. For instance, we

observe no difference in adding random accesses with the Cortex-A7. For the Cortex-A53, we add eight random accesses at the end of the reconstruction phase to reduce some noise on corner cases. If there are no random accesses, the region's size can be reduced to the size of the memory zone. Finally, the number of regions needs to be big enough to ensure the whole eviction of the previous accesses before reusing a region. Thus, a number of misses too low may indicate a too low number of regions.

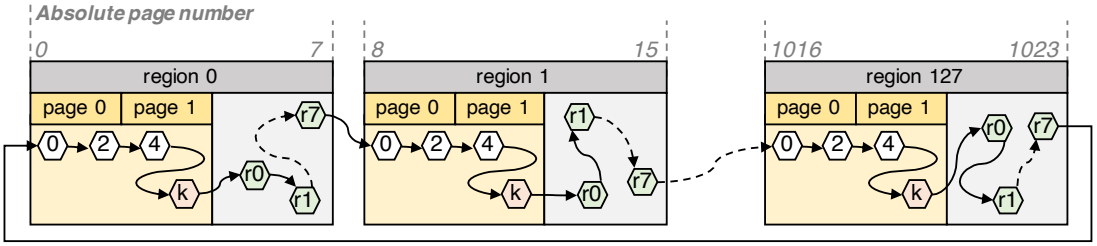

Figure 6.10: *Final pointer chasing implementing reset method.*

6.5.3 Cortex-A7 Reconstruction

The first synthetic sequence we create is a simple pattern composed of a stream with a stride equal to one. We start the sequence at cache line 52 of page 0 and continue it up to cache line 6 of the next page. Figure 6.11 shows the output of the prefetcher inspector. From this experiment, we make four observations. First, three consecutive missing accesses trigger a prefetcher burst, as we can see with the addresses 52, 53, and 54. Second, a prefetcher burst contains three prefetched cache lines, e.g., prefetcher burst of the addresses 55, 56, and 57. Third, to continue a stream prefetching, we need a request to miss the following prefetcher burst cache line. This generates another prefetcher burst of three cache lines, as seen with the addresses 58, 62, and 7. Finally, the fourth observation is that the prefetcher cannot generate prefetches across a page. We believe that comes from the fact that the data prefetcher only has access to the physical addresses and cannot determine the next page.

In order to challenge some more specific features of the Cortex-A7 data prefetcher, we illustrate in Figure 6.12 three other synthetic memory sequences that we execute with the prefetcher inspector. Thus, with Sequence #1, we use the same simple stream of four addresses with a stride of 1, $\langle 0, 1, 2, 3 \rangle$. However, this time we break the stream with a random access to the cache line 12 after the second address of the stream. We observe that this random access is sufficient to break the stream since neither the access to address 2 nor the access to address 3 generates a prefetcher burst.

With the sequence #2, we want to expose the behavior of the prefetcher when a prefetched address hits in the L1D. For that, we first access the cache line 4. Then, we

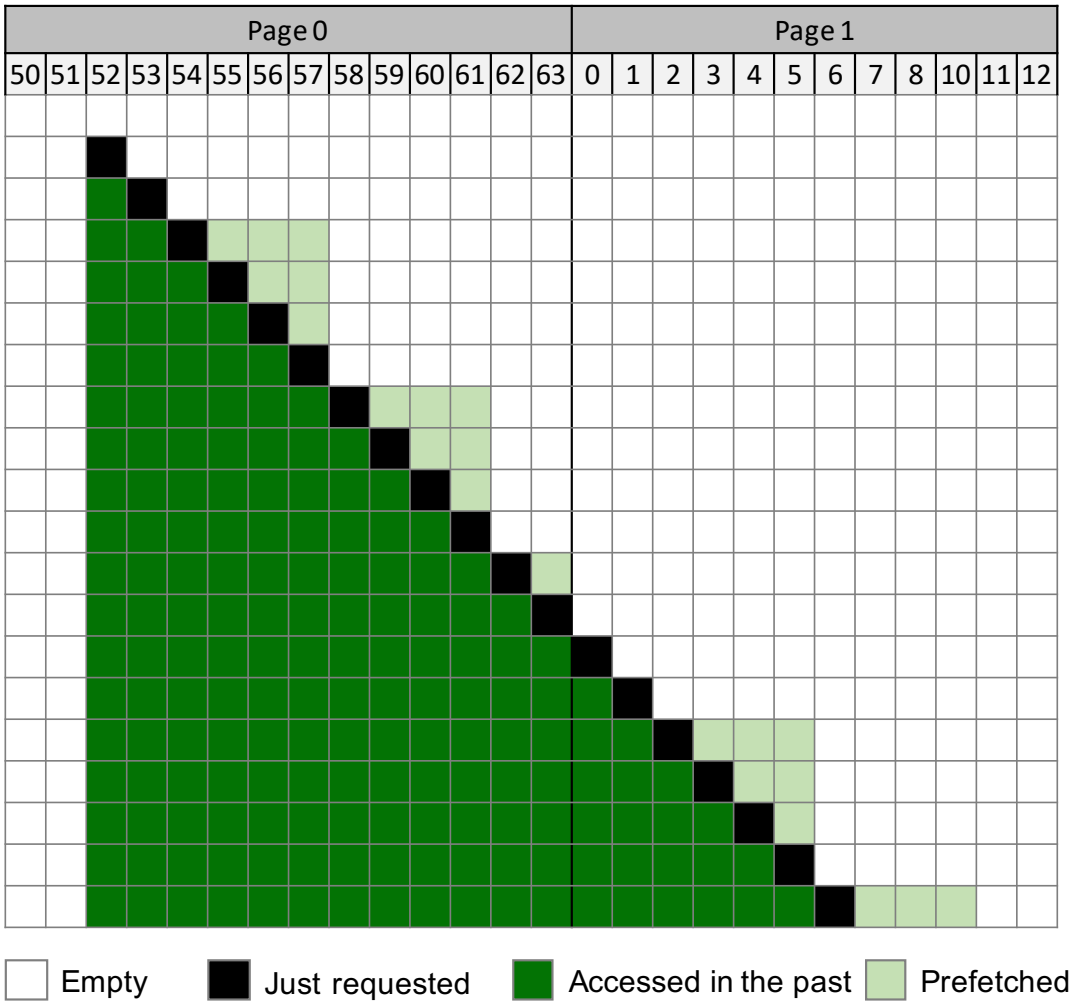

Figure 6.11: *Cortex-A7 synthetic sequence: stream stride 1.*

generate a prefetcher burst by accessing the addresses 0, 1, and 2. We observe that the prefetcher detects the stream and starts data prefetching with cache line 3. However, the burst stops if one address of the prefetcher burst is already present in the L1D. Also, the accesses to the next missing cache lines 5 and 6 cannot restart the data prefetching. Thus, a prefetched cache line that hits in the L1D deactivates the stream.

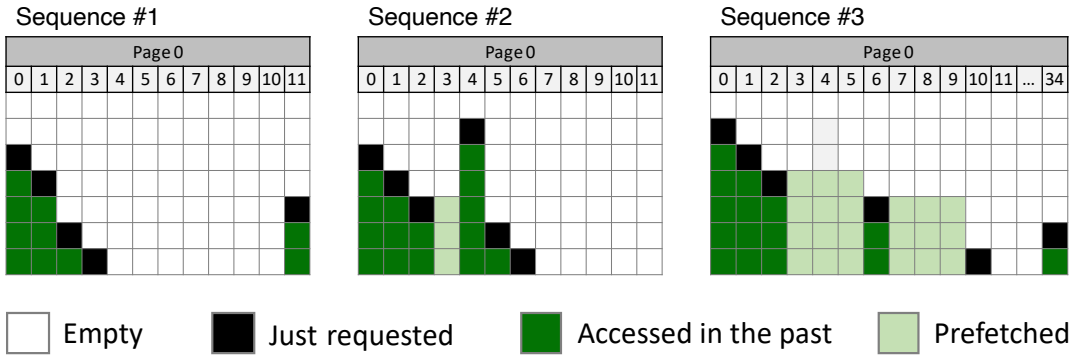

Figure 6.12: *Cortex-A7 synthetic sequences: #1, #2, and #3.*

The previous experiments show that the prefetcher does not monitor the hits in the L1D. To see if we can continue data prefetching without any hits, we design a new synthetic memory sequence with only misses, sequence #3. Hence, we start the sequence with the accesses to the addresses 0, 1, and 2. This generates the data prefetching of the cache lines 3, 4, and 5. Then, instead of continuing the stream and hitting in the prefetched cache lines, we directly access address 6. We observe that it generates a new prefetcher burst. Thus, accessing prefetched cache lines is unnecessary to continue a stream data prefetching.

Additionally, with the sequence #3, we generate random access to address 34 before continuing the stream with address 11. We observe that the access to the missing address 11 does not generate another prefetcher burst. We conclude that if a request to a missing address does not continue a stream, this prevents other access from doing so. This conclusion also shows that the prefetcher can only handle one stream. We verify that creating a synthetic sequence of two interleaved streams. The prefetcher inspector shows that no prefetcher burst is generated (result available on our online repository [16]).

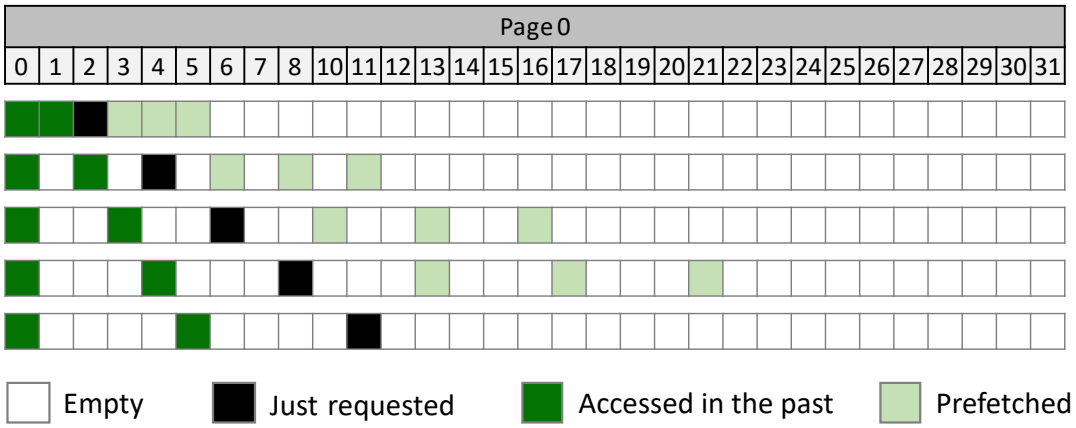

Figure 6.13: *Cortex-A7 synthetic sequence: maximum stride.*

An important feature of the stride prefetcher is the maximum stride value till the stream no longer triggers a prefetcher burst. We use multiple sequences of three addresses with different stride values to identify the maximum value. We start from the minimum value, i.e., stride equal to one, and increase it until the prefetcher no longer generates prefetches. Figure 6.13 shows the results for five sequences with stride values from one up to five. We only show on Figure 6.13 the last row of each sequence graph. We observe that up to four, the stream is detected and triggers a prefetcher burst of three cache lines for all the stride values. Also, we observe that the prefetcher adapts the burst to the stride value to only prefetch cache lines which are part of the stream.

Importantly, we show in this section just a few examples of the complete list of the synthetic sequences we use to create the functional model of the Cortex-A7. Hence, we put on our online repository [16] the complete list of sixteen synthetic memory sequences with their prefetcher inspector outputs.

6.5.4 Cortex-A53 Reconstruction

We implement the reconstruction phase of Pref-X on the Cortex-A53 as we do with the Cortex-A7. For that, we start with the same simple synthetic sequence. We create a stream with a stride equal to one from address 0 to address 23. Figure 6.14 shows the graph prefetcher inspector results. We make four observations.

First, similarly to the Cortex-A7, three missing accesses to a stream trigger a prefetcher burst, as we can see with the addresses 0, 1, and 2. Second, a prefetcher burst has a length of three cache lines, as observed with the seven prefetcher bursts present on the graph. Third, a request that hits a previous prefetched cache line triggers a new prefetcher burst. Thus, in Figure 6.14, only the first burst is generated because of missing accesses. The six others prefetcher bursts are generated after a request hits a previous prefetched cache line. Fourth, all the hitting requests do not trigger a prefetcher burst every time. Gradually, the distance between two prefetcher bursts increases. Thus, addresses 3, 4, and 5 generate one burst each. Then, address 6 does not, and finally, a burst is generated every three hits, e.g., addresses 7, 10, and 13.

Figure 6.15 shows the prefetcher inspector results of three other synthetic memory sequences. The sequence #1 shows the behavior of the prefetcher when a prefetcher burst reaches the end of the page. We observe that a prefetcher burst cannot extend to the next page. Thus, we believe that the prefetcher works with physical addresses and cannot know what the next physical addresses of the next page are. However, we observe that accessing the first cache line of the next page generates a prefetcher burst of three cache lines. The prefetcher identifies this access as the continuity of the stream and restarts the data prefetching on this new page.

With the sequence #2 of Figure 6.15, we show the maximum number of unrelated misses between accesses of the same stream that still triggers a prefetcher burst. Thus, we observe that the maximum number is six missing accesses. We create two other similar sequences but with one more missing access between the accesses of cache lines 1 and 8 and between cache lines 8 and 15. For both sequences, the stream no longer triggers a prefetcher burst. The distance between the accesses is too large.

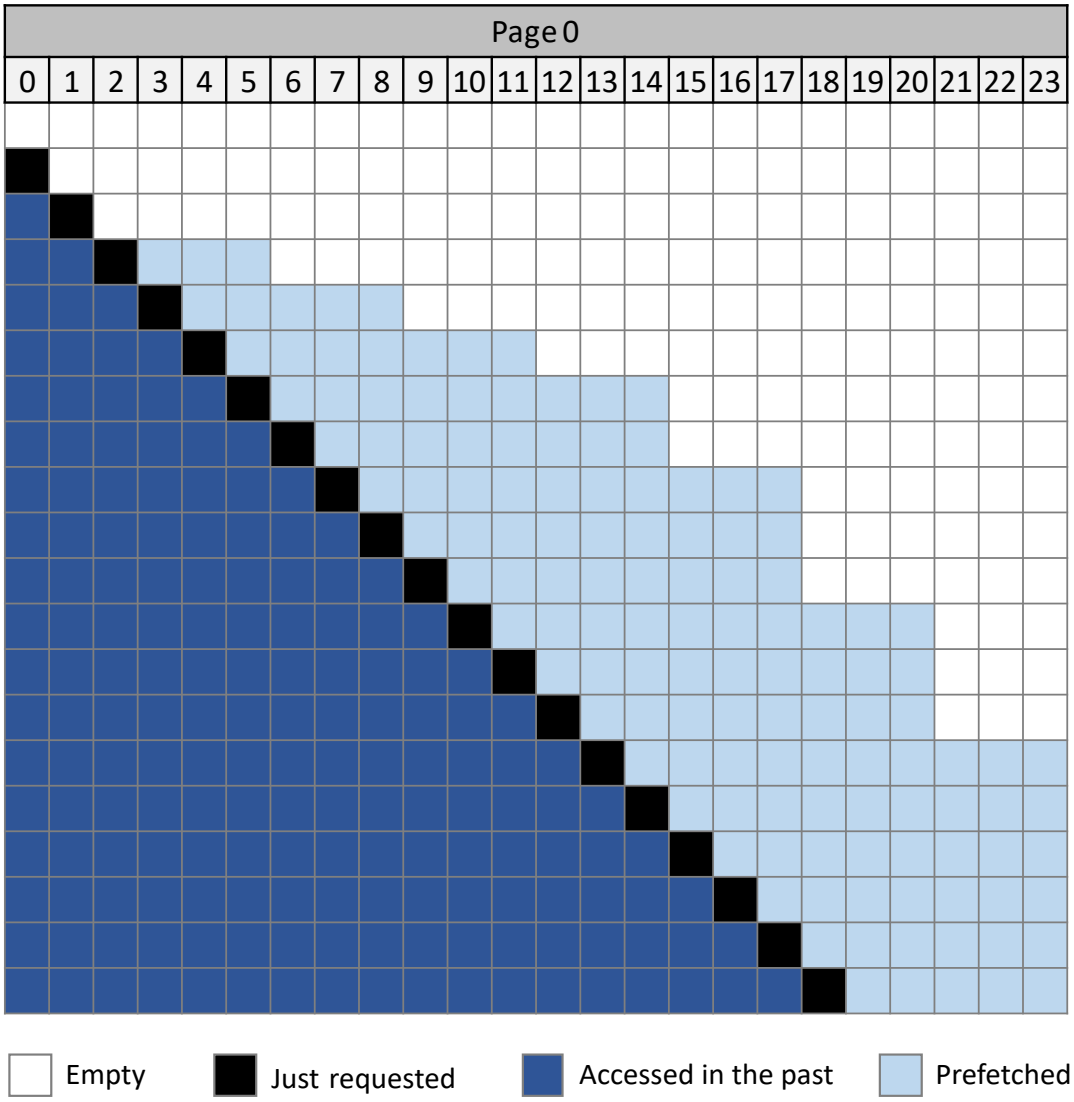

Figure 6.14: *Cortex-A53 synthetic sequence: stream stride 1.*

Another feature we expose on the sequence #2 is the behavior of the prefetcher if we access the missing cache line that follows a prefetcher burst. Hence, the end of the sequence is a request to address 6 just after the prefetched cache lines 3, 4, and 5. We observe that contrary to the Cortex-A7 prefetcher, this access triggers a one-cache line prefetch, the address 7.

With the sequence #3 of Figure 6.15, we observe that the prefetched cache lines that hit in the L1D do not stop the data prefetching. We start the sequence with two accesses to addresses 4 and 8. Then, we generate a first prefetcher burst that includes address 4 with three consecutive misses to the addresses 0, 1, and 2. We observe that as address 4 is already present in the L1D, the burst is extended to address 6 to conserve the same number of prefetched cache lines. The second burst that hits address 3 generates the same behavior. The prefetcher burst jumps the already present address 8.

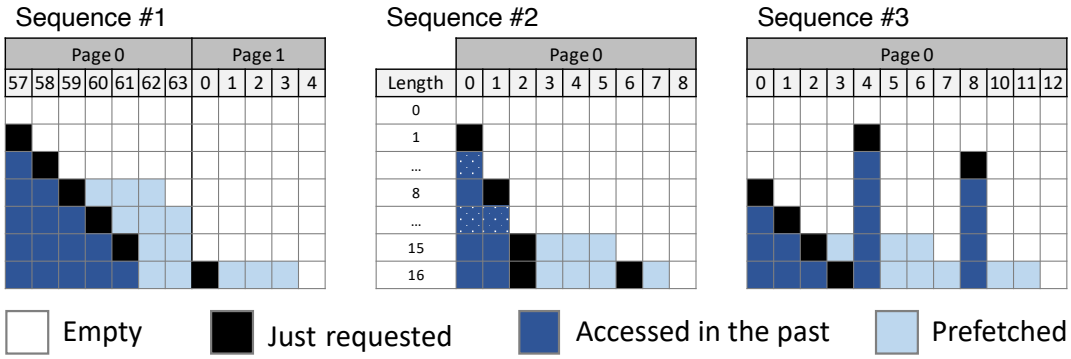

Figure 6.15: *Cortex-A53 synthetic sequence: #1, #2, and #3.*

As we see with the sequence #2 of Figure 6.15, the accesses of a stream can be distanced from other accesses. That way, we build the sequence of Figure 6.16. The sequence contains three interleaved streams with the same stride equal to 1. We observe with the accesses to addresses 2 and 34 of page 0 that the first and second streams are detected and trigger two prefetcher bursts to the corresponding stream in parallel. The third stream does not trigger any bursts. Also, in parallel, the hitting accesses to addresses 3 and 35 of the first and second streams generate new prefetcher bursts. Then, the data prefetching of the first and second streams stops, and the third stream triggers prefetcher bursts. At the end of the sequence, the first and second streams access the missing address following the last prefetched cache line, i.e., the addresses 9 and 41. We observe that only the missing access of the second stream (41) generates a one-cache line prefetch. That may indicate that the prefetcher had to stop tracking the first stream to start tracking the third one. Based on those observations, we conclude that the prefetcher can track up to two streams in parallel. We create more sequences to challenge this draw insight, including from two to four interleaved streams. We also change the stride value of the streams to see if we observe the same behavior. Those synthetic sequences are available on our online repository [16].

6.5.5 Functional Models

Once we run enough synthetic sequences, we create the Cortex-A7 and Cortex-A53 data prefetcher functional models. We implement them as Python state machines that react to incoming memory requests. We also create a functional Python model of the Cortex-A7 and Cortex-A53 L1 data cache. Both L1D caches are 4way associative and have the same 32KB size. Furthermore, they both use a Last Recent Used (LRU) replacement policy. Both functional models are available on our online repository [16]. We summarize the main features of the models in Table 6.1.

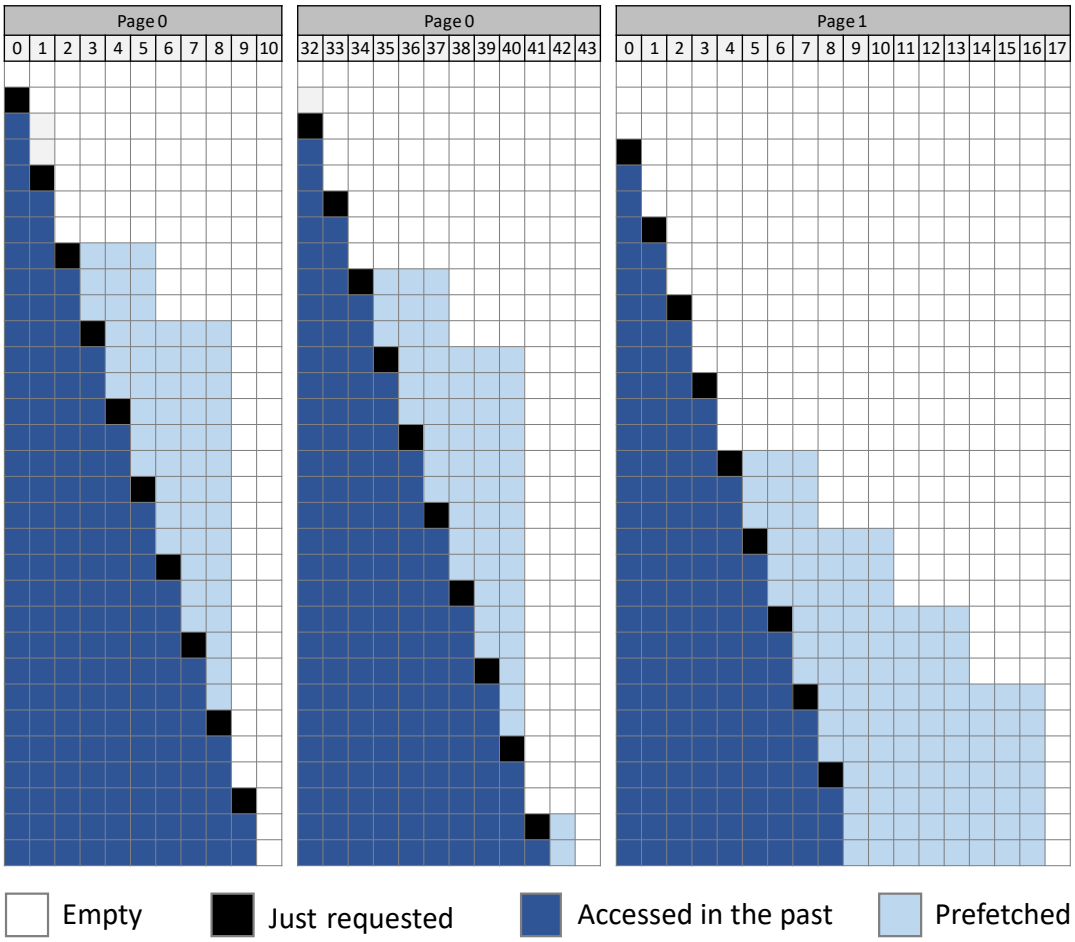

Figure 6.16: *Cortex-A53 synthetic sequence: three interleaved streams.*

To finish the first phase of Pref-X we execute all the synthetic sequences on both functional models. For each sequence, we verify that the functional model behaves the same way as the real reference data prefetcher. When the results are precisely the same, we end the first phase of Pref-X.

6.5.6 Verification

Once we finish the reconstruction phase, we start the verification phase of both functional models. For that, we first generate realistic memory sequences that we execute and simulate on the real reference commercial CPUs.

We use the gem5 simulator and the SPEC CPU2006 suite to generate realistic sequences. We use the HPI gem5 model and the SimPoint methodology [108] with only 1 million instructions per slice and K set to 30. For each simpoint, we extract from the simulation a memory trace. We filter this trace to keep the accesses to L1 and then split it into segments of 1 thousand lines. We generate realistic memory sequences

Table 6.1: *Comparison of the A7 and the A53 stride prefetchers.*

Parameter	Cortex-A7	Cortex-A53
Initial trigger cond.	3 misses	3 misses
Trigger input	L1 misses	L1 misses + hit on prefetch
Burst length	3	3 / 1
Max. stride length	4	4
Max. dist. in requests	1	7
Cond. to continue	Miss after prefetched	Hit on prefetch: burst of 3 / miss after prefetch: burst of 1
Burst hitting in L1	Stop burst	Keep burst skipping lines in L1
Tracking across pages	No	Yes
Max. num. of streams	1	2
Max. inter stream dist.	-	8: from 3rd miss to any prefetch

from those segments. For that, we remove redundant accesses to the same cache line, e.g., multiple accesses to different words of the same cache line. In order to reduce the memory footprint, we compact the page mapping by removing unused intermediate pages. Finally, we remove the sequences with less than ten accesses or more than one hundred pages as those sequences would generate very low data prefetching activity. Overall, we obtain around 149 thousand realistic memory sequences, including over 16 million memory accesses.

Once the realistic sequences are generated, we execute them on both the commercial CPUs and the functional models. For that, we use a simpler version of the prefetcher inspector. This version only executes the memory sequences on the CPU and measures the number of prefetches generated. We compare the number of prefetches generated on the commercial CPU and the corresponding function model. When a significant difference is detected, we divide the sequence, which may contain several hundred accesses and a dozen pages, into smaller portions that we analyze with the prefetcher inspector. We use the results from the prefetcher inspector to fine-tune the functional model.

As we see during the reconstruction phase, the Cortex-A7 data prefetcher is not as complex as the Cortex-A53 one. Thus, we mainly iterate the verification phase with the Cortex-A53 functional model. We use four sequences that we split into 24 realistic sequences. All those realistic sequences and prefetcher inspector results are available on our online repository [16].

6.6 Evaluation

In this section, we evaluate the functional models of the Cortex-A7 and Cortex-A53 that we create in the previous Section 6.5. We use the memory sequences generated with gem5 and the SimPoint methodology with different metrics to show the accuracy of our framework.

One the execution and simulation of memory sequences, Figure 6.17 shows the prefetcher intensity, which is the ratio between the number of prefetches and memory accesses. We sum the number of prefetches and memory accesses for all the sequences belonging to the same benchmark. Then, we process the ratio for each benchmark and the average of all. We observe that the Cortex-A53 has an average prefetcher intensity close to 40%, which is three times higher than the Cortex-A7. Also, one benchmark, 462.libquantum, has a prefetcher intensity higher than 100% with the Cortex-A53. Thus, more than half of the data movement is due to the data prefetcher with these memory sequences. Consequently, Figure 6.17 motives the need for accurate data prefetcher models in architecture simulators.

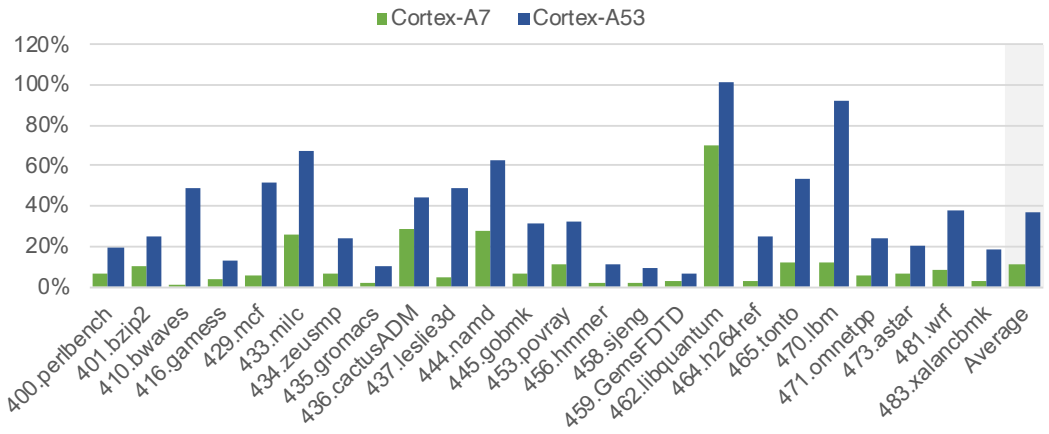

Figure 6.17: *Prefetcher Intensity of SPEC CPU2006 benchmarks executed on the Cortex-A7 and Cortex-A53, and simulated on their corresponding functional models.*

Figure 6.18 shows the modeling error for each benchmark and the average of all of them. We define the modeling error as the absolute difference in prefetches between the real execution and the functional model, normalized with the results from the real execution. We observe that the average and maximum modeling error for the Cortex-A7 are very low, 0.2% and 0.8%, respectively. For the Cortex-A53, the average and maximum modeling error for the Cortex-A7 are very low, 3.2% and 5.9%, respectively. The modeling error difference between both CPUs comes from the more complex data prefetching processes implemented in the Cortex-A53. In addition, the higher prefetcher intensity of the Cortex-A53 causes small deviations in behavior to

generate high modeling errors. However, we believe the modeling error to be low enough to validate the functional models created with Pref-X.

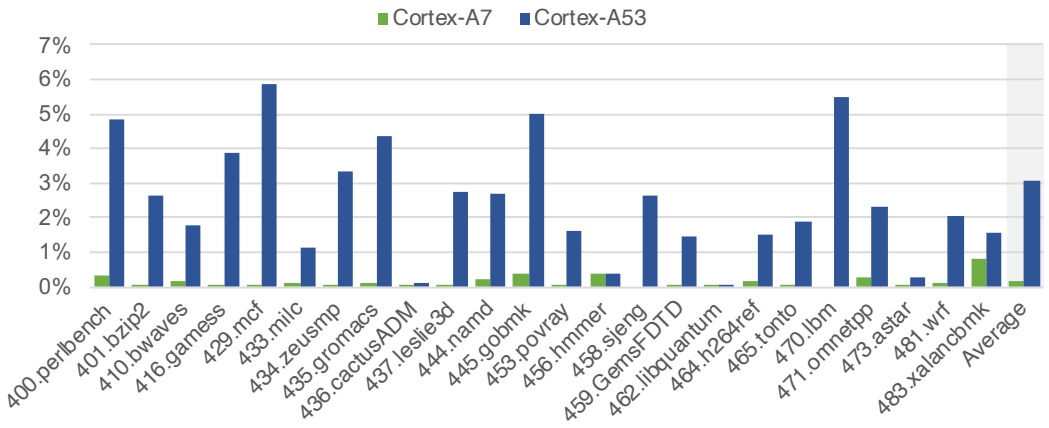

Figure 6.18: *Normalized modeling error of SPEC CPU2006 benchmarks executed on the Cortex-A7 and Cortex-A53, and simulated on their corresponding functional models.*

Furthermore, we show in Figure 6.19 the distribution of the memory sequences depending on the number of prefetches for the Cortex-A7 and the Cortex-A53. Each sequence is assigned to a bin, defined by a minimum and a maximum number of prefetches. We calculate the average modeling error for each bin and plot it on Figure 6.19. From the results, we make two observations. First, most sequences generate less than ten prefetches for the Cortex-A7 and less than twenty for the Cortex-A53. Second, the modeling error remains quite constant and low across all bins, meaning that the accuracy of our functional models does not depend on the prefetcher intensity.

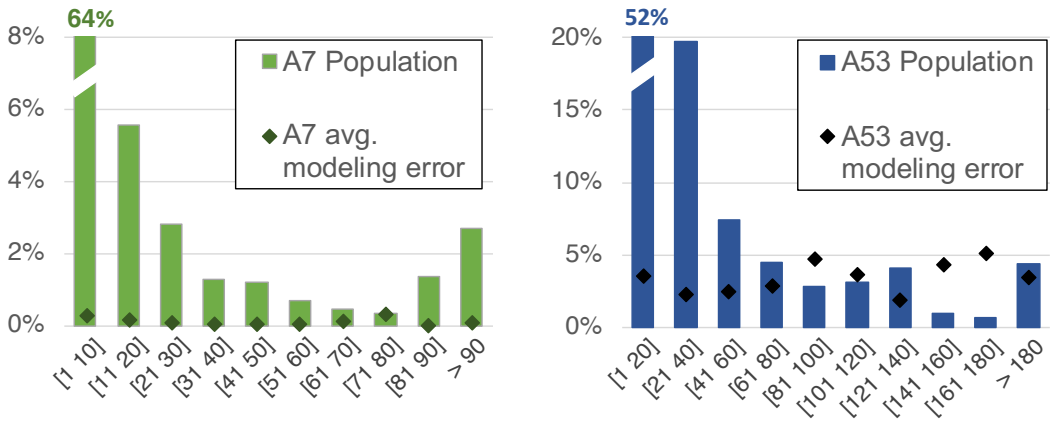

Figure 6.19: *Memory sequences distribution depending on the number of prefetches generated on the Cortex-A7 and Cortex-A53.*

6.7 Summary

To conclude, we propose in this chapter Pref-X, a framework to analyze functional characteristics of data prefetching in commercial in-order cores. We first illustrate the insights we use to build the framework based on the methodology introduced in Chapter 4. Basically, we stress the data prefetcher using specific memory request sequences. Then, we inspect the addresses present in the cache to detect data prefetching. We enumerate the possible scenarios that can arise from executing the sequence and inspecting the cache content. Finally, depending on the scenario monitored, we can conclude whether either an address has been prefetched or not.

We introduce the two phases of the methodology, i.e., the reconstruction and the verification phases. The first phase allows to create a functional model of the target architecture data prefetcher. For that, we describe the prefetcher inspector, a sub-tool of Pref-X. The prefetcher inspector is a set of microbenchmarks and scripts that take memory sequences as input and execute them on the target architecture to expose data prefetching activity.

Once we reconstruct the functional model with the first phase, we start verifying it with the second one. For that, we use realistic benchmarks and extract from them realistic memory sequences. We execute those sequences on the target architecture using a simpler version of the prefetcher inspector and the functional model. We compare the results to detect differences. Then, we correct the functional model in function of the new insights we get from the verification results.

Finally, we implement Pref-X on two ARM in-order cores, the Cortex-A7, and the Cortex-A53. To evaluate the two data prefetcher functional models, we use benchmarks from the SPEC CPU2006 suite. We use the gem5 simulator to extract around 149 thousand memory sequences that we execute on both the functional models and the target architectures. Thus, we show a functional accuracy of 99.8% and 96.8% for, respectively the Cortex-A7 and the Cortex-A53.

VII

Conclusion

In summary, this thesis tackles the calibration problem of computer architecture simulation models. We focus on the on-chip memory subsystem and its interaction with the main working memory, which significantly impacts modern architecture performance and energy efficiency. In particular, this thesis aims to provide more accurate computer architecture simulations by reducing parametrization errors in the memory system of existing simulation models.

In this chapter, we summarize our key findings and propose some recommendations for future work. Finally, we provide concluding remarks.

7.1 Summary of Key Findings

This thesis contributes to the reduction of parametrization errors in computer architecture simulators with three main contributions that are summarized next.

7.1.1 Memory Timing Calibration

The first main contribution of the thesis is a systematic methodology to calibrate the memory system of architecture simulations against a real target commercial architecture. Thus, the methodology provides a complete workflow through two phases: Parameter Identification and Parameter Discovery. The first phase describes the generic memory system parameters, which are summarized in a template applicable to different simulators. Then, the second phase details the modular design of handcrafted microbenchmarks that we use to extract missing technical information necessary for simulator parameter calibration. The design includes multiple microbenchmark features, i.e., data pinning, memory requests dependency, and noise minimization, which are instantiated accordingly to the target architecture. We apply our general methodology to two different use cases, which correspond to our next two main contributions.

7.1.2 Memory Levels Instance

The second main contribution is the instantiation of the methodology to the timing calibration of a simulator's memory system. We implement the instantiated methodology on the gem5 simulator and one ARM Cortex-A53 present in the MediaTek Helio X20 SoC as the target architecture. We show that the designed microbenchmarks allow the extraction of technical information:

- Cache levels: size, parallelism, replacement policy, and access time.
- TLB: micro-TLB penalty, micro-TLB entries, main-TLB penalty, and main-TLB entries
- Main memory: access time, conflict penalty, and address mapping

To evaluate the methodology, we compare the simulation of the SPEC CPU2006 suite on three gem5 configurations against execution on the MediaTek Helio X20. The three gem5 configurations are the default gem5 baseline, i.e., the HPI model, the calibrated gem5 configuration, and the calibrated gem5 configuration extended, which includes Ramulator to simulate the main memory. The results show that the calibrated gem5 configuration (extended) has an average error of 17.6% (16.5%) and a maximum error of 37.1 % (36.2%). Reducing the average simulation error by 39.6% (43.3%) and the maximum error by 61.5% (62.5%) compared to the default configuration. Thus, the evaluation shows that the methodology achieves better simulation accuracy, offering sufficient baseline models for accurate design explorations.

Additionally, we compare the execution time of both calibrated configurations. Thus, we observe that to reduce the average and maximum errors by respectively 6.2% and 2.5%, the extended configuration increases the simulation time by 60%, from 128h to 205h. Thus, it illustrates that the modeling accuracy needs to be properly adapted considering the application to optimize the simulation time.

This work resulted in two contributions:

- International Conference: Quentin Huppert, Timon Evenblij, Manu Perumkunnil, Francky Catthoor, Lionel Torres, and David Novo. "Memory Hierarchy Calibration Based on Real Hardware In-order Cores for Accurate Simulation," in Proceedings of DATE, 2021.
- National Symposium: Quentin Huppert, Lionel Torres, and David Novo. "Memory Hierarchy Calibration Based on Real Hardware In-order Cores for Accurate Simulation," Poster in GDR SoC2 Colloque, 2021.

7.1.3 Pref-X Instance

The third main contribution is an instantiation of the methodology on the data prefetching engine: a key memory system component that was disregarded in our previous study. The instantiation results in a framework, Pref-X [16], to analyze the

functional characteristics of data prefetching in commercial in-order cores. We implement Pref-X on two ARM in-order cores, the Cortex-A7, and the Cortex-A53, and their respective functional models. We open-source the tool and the results of both implementations on our online repository [16].

We evaluate Pref-X using memory traces extracted from the SPEC CPU2006 suite. We first execute the traces on both target CPUs and show that the data prefetcher has a significant role in the data movement as the average prefetcher intensity of the Cortex-A7 and Cortex-A53 is respectively 11.1% and 36.6%. However, depending on the benchmark and the data prefetcher, the prefetcher intensity value can change significantly. Thus, the prefetcher intensity for the Cortex-A7 is between 1.2% and 70.0%, and between 6.8% and 101.6% for the Cortex-A53. Noticeably, more than 100% means that more than half of the data movement is due to the data prefetcher.

Then, we execute the SPEC CPU2006 memory traces on both functional models and compare the results with the results from the target CPUs. The results show that Pref-X provides functional accuracy of 99.8% and 96.8% for the Cortex-A7 and the Cortex-A53, respectively. Also, we observe that for all the benchmarks, the minimum accuracy is 99.1% for the Cortex-A7 and 94.1% for the Cortex-A53. It shows that the accuracy is relatively constant over the benchmarks. The memory traces used for the evaluation are available on our online repository [16].

Additionally, we bin the memory traces depending on the number of prefetches they generate on the target CPUs. The population of the binning shows that many memory traces do not generate high data prefetching activity. However, the average accuracy remains low and constant over the bins, i.e., from a low to high data prefetcher activity. Hence, the results show that Pref-X allows good data prefetching functional modeling stride prefetchers, even with high data prefetching intensity.

This work resulted in two contributions:

- International Conference: Quentin Huppert, Francky Catthoor, Lionel Torres, and David Novo. "Pref-X: a framework to reveal data prefetching in commercial in-order cores," in Proceedings of DAC, 2022.
- Code Release: Quentin Huppert. "Pref-X." `https://gite.lirmm.fr/adac/pref-x`

We also plan to submit an extension of our DAC paper, including the analysis of the more modern Cortex-A55 data prefetcher, to the IEEE Transactions on Computers before the end of the year.

7.2 Recommendations for Future Work

We illustrate in this thesis two instances of the calibration methodology that we propose. The evaluation of those methodology instances show conclusive results. Hence, we believe that the number of instances of the methodology should be increased. Thus, in the rest of the section we introduce different instances of the methodology that we identify as future work.

7.2.1 Timing Calibration Coverage

We believe that the methodology's instantiation we propose for the memory system's reactive components could be extended for the timing aspects of different parameters or components. Thus, to illustrate possible extension directions, we introduce different examples for each element we have just enumerated.

Components parameters. During the instantiation, we extract multiple parameter values from the target architectures. However, the proposed microbenchmarks do not extract all the key parameter values introduced in the methodology. For instance, the bandwidth is a crucial characteristic of the interconnect, especially with multicore simulations. Thus, a dedicated microbenchmark design should be detailed to calibrate this parameter which is often not public. In the same, if the associativity of the caches is not disclosed, microbenchmarks should extract it. Consequently, multiple parameters, such as the main memory organization, or the DRAM controller policies, should still be part of microbenchmark designs.

Additional memory system. The methodology instantiation covers multiple components of the reactive memory system. However, some of them are still not covered by it. Thus, the instantiation could further include the storage memory level in the calibration process. However, adding storage memory to the simulation implies simulating an OS, as it manages the complete data movement between the main and the storage memory. Another component that is not covered by the instantiation is the cache coherence protocol. Multicore architectures can implement many different protocols. The calibration of this component would be necessary for future explorations. For instance, to explore the utilization of NVM within the memory system, it could be necessary to evaluate the impact of the coherent cache protocol against the NVM write penalty.

7.2.2 Data Prefetching Analysis

The instantiation of the methodology on the data prefetcher engine results with Pref-X framework, which suffers from two limitations. First, the framework is limited to analyzing in-order core data prefetchers. Second, the framework proposes functional modeling of the data prefetcher, which does not include all the characteristics, such as the timing characteristics.

Coverage. Pref-X implements the prefetcher inspector to expose data prefetching activity. The prefetcher inspector microbenchmark is based on three important steps. First, it executes the memory sequence. Then, it inspects one address. Finally, it resets both the cache and the prefetcher for the next iterations, i.e., to make each iteration independent from the other. The execution of the memory sequence and the address inspection can easily be done with other target architectures. However, the final reset step is more tricky as we can not directly flush data prefetcher metadata. For instance, we propose with the Cortex-A7 and Cortex-A53 to use several relative memory zones like that we do not use the same addresses over the iterations. However, this reset method could be usable with all the data prefetcher. Some data prefetchers may identify the same pattern over different physical addresses and trigger prefetches. Consequently, a significant part of the Pref-X coverage depends on the possibility of resetting or not the state of the data prefetcher. Accordingly, future work would focus on evaluating the reset method on other architectures to extend Pref-X on them.

Timing metrics. The data prefetcher is part of the complete memory system. Similar to the other components, it operates concerning timing constraints. In order to focus on the functional behavior of the data prefetcher, we add many NOPs operations in the microbenchmark measure loop operations to evict any timing interferences. For instance, the number of outstanding requests the prefetcher can generate in parallel does not impact the functional behavior. In this way, we believe that the instantiation of the methodology could be extended to generate a complete data prefetcher model. Further, the extended instantiation would be used to calibrate more architecture simulators using flawed data prefetcher baseline models.

7.2.3 Further Instantiations

In this thesis, we instantiate our methodology on typical CPU memory systems with a traditional cache hierarchy accessed from the main SDRAM-based working memory. As we develop this methodology to be systematic, we propose instantiating it through

other kinds of memory system organizations as future work. The principal limitation of the methodology comes from hardware monitoring. However, performance counters are available in many other kinds of architecture. Thus, the methodology could be instantiated on GPU or CNN accelerator memory systems. Moreover, new organizations of the memory hierarchy would also be included. For instance, scratchpad implementation [154], or the RISCV MemPool [155] with its shared-L1.

Another way to instantiate the methodology would be to have new hardware measurements available, such as energy consumption. Thus with an energy measurement, the methodology could be instantiated to calibrate power architecture simulators. Hence, exploring new hardware measurements should also be part of future work.

7.2.4 Beyond the Memory System

Finally, with the methodology, we focus on a critical component of modern multicore architectures, the memory system. However, we believe this methodology to be used as a reference to other calibration methodologies. Thus, future methodologies could target other elements of the architecture, such as the different functional units of the core pipelines. Moreover, modern multicore architectures implement many different process units like GPUs or accelerators. Those elements can significantly impact global system performance and energy efficiency. In this way, dedicated methodologies can be part of future work.

7.3 Concluding Remarks

In this thesis, we explore memory system modeling and its calibration against real state-of-the-art architectures. We illustrate the challenges of calibrating computer architecture simulators and propose a systematic methodology to calibrate their memory system. The methodology details and demonstrates the design of handcrafted microbenchmarks that use hardware monitoring to reveal hidden technical information needed for adequate calibration. We describe two instantiations of our methodology on two different use cases. First, we propose a memory system timing calibration to extract non-public timing information from commercial architectures. Second, we propose a framework to analyze in-order core data prefetching and derive a complete functional model of commercial data prefetchers. We hope our methodology and use-case instantiations will help the community achieve more accurate simulations, which

are crucial for exploring new multicore architectures. Also, we hope this thesis will encourage similar work to improve the accuracy of computer architecture simulation further.

Bibliography

[1] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of ASPLOS*, 2018. 3, 30

[2] Jalil Boukhobza, Stéphane Rubini, Chen Renhail, and Zili Shaor. Emerging NVM: A survey on architectural integration and research challenges. *TODAES*, 23(2):1–32, 2017. 3, 30, 46

[3] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *Proceedings of ISPASS*, pages 256–267, 2013. 3, 30

[4] Clinton W. Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *Proceedings of HPCA*, pages 50–61, 2011. 3, 26

[5] Sophiane Senni, Lionel Torres, Gilles Sassatelli, Abdoulaye Gamatie, and Bruno Mussard. Emerging non-volatile memory technologies exploration flow for processor architecture. In *Proceedings of VLSI*, pages 460–460, 2015. 3, 26

[6] K. Skadron, M. Martonosi, D.I. August, M.D. Hill, D.J. Lilja, and V.S. Pai. Challenges in computer architecture evaluation. *Computer*, 36(8):30–36, 2003. 3, 20, 26

[7] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *IEEE Access*, 7:78120–78145, 2019. 3, 20, 26, 27

[8] Ayaz Akram and Lina Sawalha. ×86 computer architecture simulators: A comparative study. In *Proceedings of ICCD*, pages 638–645, 2016. 3, 20, 22, 27

[9] Richard A Uhlig and Trevor N Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997. 3, 20

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011. 3, 5, 21, 22, 25, 26, 27, 30, 31

[11] Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. Architectural simulators considered harmful. *IEEE Micro*, 35(6):4–12, 2015. 3, 20, 26

[12] Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5. In *Proceedings of SAMOS)*, pages 266–273, 2014. 4, 21

[13] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Sassatelli Gilles. Accuracy evaluation of gem5 simulator system. In *Proceedings of ReCoSoC*, 2012. 4, 21

[14] Anthony Gutierrez, Joseph Pusdesris, Ronald G Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D Emmons, Mitchell Hayenga, and Nigel Paver. Sources of error in full-system simulation. In *Proceedings of ISPASS*, 2014. 4, 21, 27

[15] Mediatek x20 development board hardware user manual. https://www.96boards.org/documentation/consumer/mediatekx20/hardware-docs/hardware-user-manual.md.html. [September-22]. 5, 10, 46, 51

[16] Pref-X GitLab repository. https://gite.lirmm.fr/adac/PhD_Template. 5, 89, 90, 92, 94, 101, 102

[17] Onur Mutlu. Memory scaling: A systems architecture perspective. In *Proceedings in IMW*, pages 21–25, 2013. 8

[18] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. A survey of different approaches for overcoming the processor-memory bottleneck. *International Journal of Computer Science and Information Technology*, 9(2):151–163, 2017. 8

[19] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. A review of near-memory computing architectures: Opportunities and challenges. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 608–617, 2018. 8

[20] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010. 8, 9

[21] Shimeng Yu and Pai-Yu Chen. Emerging memory technologies: Recent trends and prospects. *IEEE Solid-State Circuits Magazine*, 8(2):43–56, 2016. 8

[22] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *Proceedings of IEDM*, pages 1.1.1–1.1.8, 2016. 8

[23] P Greenhalgh. big.LITTLE technology: The future of mobile. *Arm Limited, White Paper*, page 12, 2013. 10, 21

[24] ARM Cortex-A series programmer's guide for armv8-a. https://developer.arm.com/documentation/den0024/a/Caches/Cache-terminology/Cache-tags-and-Physical-Addresses. [Sep-22]. 13, 123

[25] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A performance comparison of contemporary dram architectures. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 222–233, 1999. 14

[26] Kiyoo Itoh, Yoshinobu Nakagome, Shin'ichiro Kimura, and Takao Watanabe. Limitations and challenges of multigigabit dram chip design. *IEEE Journal of Solid-State Circuits*, 32(5):624–634, 1997. 14

[27] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2017. 14

[28] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *2012 symposium on VLSI technology (VLSIT)*, pages 87–88. IEEE, 2012. 14

[29] Vivek Seshadri and Onur Mutlu. In-dram bulk bitwise execution engine. *arXiv preprint arXiv:1905.09822*, 2019. 14

[30] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2016. 14, 17, 21, 23, 25, 27, 30, 31, 62

[31] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 18, 1990. 15

[32] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G Abraham. Effective stream-based and execution-based data prefetching. In *Proceedings of ICS*, 2004. 15

[33] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of ICS*, 1991. 15

[34] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of ASPLOS*, 2020. 15

[35] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *Proceedings of MICRO*, 2021. 15

[36] Arm Cortex-A53 MPCore processor technical reference manual/cpu auxiliary control register. https://developer.arm.com/documentation/ddi0500/j/System-Control/AArch32-register-descriptions/CPU-Auxiliary-Control-Register?lang=en. [September-22]. 16

[37] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006. 16

[38] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, nov 2005. 16

[39] Ayaz Akram and Lina Sawalha. A study of performance and power consumption differences among different isas. In *Proceedings of DSD*, pages 628–632, 2019. 16

[40] Ashkan Tousi and Chuan Zhu. Arm research starter kit: System modeling using gem5, 2017. 17

[41] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of SC*, pages 1–12, 2011. 17, 22, 23, 26, 27, 30

[42] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: A full system simulator for multicore x86 cpus. In *Proceedings of DAC*, pages 1050–1055, 2011. 17, 22, 23, 27, 30

[43] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news*, 41(3):475–486, 2013. 17, 22, 27, 30

[44] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Proceedings of ISPASS*, pages 33–42. IEEE, 2009. 17, 23

[45] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro Lopez. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In *Proceedings of SBAC-PAD*, pages 62–68, 2007. 17, 22, 27, 30

[46] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002. 17, 25

[47] Matt Poremba and Yuan Xie. Nvmain: An architectural-level main memory simulator for emerging non-volatile memories. In *2012 IEEE Computer Society Annual Symposium on VLSI*, pages 392–397, 2012. 17, 23, 25, 30

[48] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. Dramsim: a memory system simulator. *ACM SIGARCH Computer Architecture News*, 33(4):100–107, 2005. 17, 23, 25

[49] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE computer architecture letters*, 10(1):16–19, 2011. 17, 23

[50] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: a cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020. 17, 23

[51] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth Pugsley, Aniruddha Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep*, pages 1–24, 2012. 17, 23, 27

[52] Joshua J. Yi, Lieven Eeckhout, David J. Lilja, Brad Calder, Lizy K. John, and James E. Smith. The future of simulation: a field of dreams. *Computer*, 39(11):22–29, 2006. 20

[53] R. Desikan, D. Burger, and S.W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of ISCA*, pages 266–277, 2001. 21

[54] D.G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of MICRO*, pages 43–54, 2004. 21

[55] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11(3), 2014. 21, 26

[56] Marco Antonio Zanata Alves, Carlos Villavieja, Matthias Diener, Francis Birck Moreira, and Philippe Olivier Alexandre Navaux. Sinuca: A validated micro-architecture simulator. In *Proceedings of PACT*, pages 605–610, 2015. 21, 27

[57] Standard Performance Evaluation Corporation. Spec cpu 2006. https://www.spec.org/cpu2006/. [Accessed: Sep-22]. 21, 22, 47

[58] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008. 21

[59] Arm. Cortex-A8 technical reference manual. https://developer.arm.com/documentation/ddi0344/k/. [Accessed: Sep-22]. 21

[60] Arm. Cortex-A9 technical reference manual. https://developer.arm.com/documentation/ddi0388/latest. [Accessed: Sep-22]. 21

[61] Arm. Cortex-A7 MPCore technical reference manual. https://developer.arm.com/documentation/ddi0464/f/. [Accessed: Sep-22]. 21, 85

[62] Arm. Arm Cortex-A15 MPCore processor technical reference manual. https://developer.arm.com/documentation/ddi0438/i/. [Accessed: Sep-22]. 21

[63] Anastasiia Butko, Florent Bruguier, Abdoulaye Gamatié, Gilles Sassatelli, David Novo, Lionel Torres, and Michel Robert. Full-system simulation of big. little multicore architecture for performance and energy exploration. *mcsoc*, pages 201–208, 2016. 21

[64] Ayaz Akram and Lina Sawalha. Validation of the gem5 simulator for x86 architectures. In *Proceedings of PMBS*, 2019. 21

[65] linux. sperf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. [Accessed: Sep-22]. 21

[66] Matt T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of ISPASS*, pages 23–34, 2007. 22, 23, 27

[67] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of WWC*, pages 3–14, 2001. 22

[68] Haihang You Dan Terpstra, Heike Jagode and Jack Dongarra. Collecting performance data with PAPI-C. *Tools for High Performance Computing*, pages 157–173, 2010. 22, 38

[69] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems. In *Proceedings of ICS*, pages 37–46, 2013. 23

[70] J. W. Sheaffer, D. Luebke, and K. Skadron. A flexible simulation framework for graphics architectures. In *Proceedings of the SCOPES*. Association for Computing Machinery, 2004. 23

[71] Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015. 23

[72] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of ISCA*, pages 280–289, 2010. 23

[73] Jieun Lim, Nagesh B Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. Power modeling for gpu architectures using mcpat. *ACM TODAES*, 19(3):1–24, 2014. 23

[74] Karthik Ramani, Ali Ibrahim, and Dan Shimizu. Powerred: A flexible modeling framework for power efficiency exploration in gpus. In *Proceedings of the Workshop on General Purpose Processing on GPUs, GPGPU*, volume 7. Citeseer, 2007. 23

[75] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. *ACM SIGARCH Computer Architecture News*, 41(3):487–498, 2013. 23

[76] V.M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and Espasa E. Attila: a cycle-level execution-driven simulator for modern gpu architectures. In *Proceedings of ISPASS*, pages 231–241, 2006. 23

[77] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009*

IEEE International Symposium on Performance Analysis of Systems and Software, pages 163–174, 2009. 23

[78] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *Proceedings of USENIX ATC*, 2008. 23

[79] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of ICS*, pages 96–107, 2011. 23

[80] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. Vssim: Virtual machine based ssd simulator. In *Proceedings of MSST*, pages 1–14, 2013. 23

[81] Myoungsoo Jung, Jie Zhang, Ahmed Abulila, Miryeong Kwon, Narges Shahidi, John Shalf, Nam Sung Kim, and Mahmut Kandemir. Simplessd: Modeling solid state drives for holistic system simulation. *IEEE Computer Architecture Letters*, 17(1):37–41, 2017. 23

[82] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQsim: a framework for enabling realistic studies of modern multi-queue SSD devices. *FAST 18*, pages 49–65, 2018. 23, 25, 30, 62

[83] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):1–25, 2017. 23

[84] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. A survey of cache simulators. *ACM Computing Surveys (CSUR)*, 53(1):1–32, 2020. 23

[85] Smruti R Sarangi, Rajshekar Kalayappan, Prathmesh Kallurkar, Seep Goel, and Eldhose Peter. Tejas: A java based versatile micro-architectural simulator. In *Proceedings of PATMOS*, pages 47–54. IEEE, 2015. 23

[86] Ravi Iyer. On modeling and analyzing cache hierarchies using casper. In *Proceedings of MASCOTS*, pages 182–187. IEEE, 2003. 23

[87] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008. 23

[88] Hyokeun Lee, Hyungsuk Kim, Seokbo Shim, Seungyong Lee, Dosun Hong, Hyuk-Jae Lee, and Hyun Kim. Pcmcsim: An accurate phase-change memory controller simulator and its performance analysis. In *Proceedings of ISPASS*, pages 300–310. IEEE, 2022. 23

[89] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *Proceedings of SIMUL*, pages 125–131. IEEE, 2009. 23

[90] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *Proceedings of ISCA*, pages 473–486, 2020. 23

[91] Tien-Ju Yang, Yu-Hsin Chen, Joel Emer, and Vivienne Sze. A method to estimate the energy consumption of deep neural networks. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 1916–1920, 2017. 23

[92] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019. 23

[93] Michael K. Papamichael, James C. Hoe, and Onur Mutlu. Fist: A fast, lightweight, fpga-friendly packet latency estimator for noc modeling in full-system simulations. In *Proceedings of the Fifth ACM/IEEE International Symposium*, pages 137–144, 2011. 23, 25

[94] Vincenzo Catania, Andrea Mineo, Salvatore Monteleone, Maurizio Palesi, and Davide Patti. Noxim: An open, extensible and cycle-accurate network on chip simulator. In *Proceedings of ASAP*, pages 162–163, 2015. 23

[95] Nan Jiang, Daniel U Becker, George Michelogiannakis, James Balfour, Brian Towles, David E Shaw, John Kim, and William J Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Proceedings of ISPASS*, pages 86–96. IEEE, 2013. 23

[96] Oumaima Matoussi. Noc performance model for efficient network latency estimation. In *Proceediongs of DATE*, pages 994–999. IEEE, 2021. 23, 25

[97] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. Gvsoc: A highly configurable, fast and accurate full-platform simulator for risc-v based iot processors. In *Proceedings of ICCD*, pages 409–416, 2021. 24

[98] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of MICRO*, page 469–480, 2009. 24

[99] William Lloyd Bircher and Lizy K John. Complete system power estimation using processor performance events. *IEEE Transactions on Computers*, 61(4):563–577, 2011. 24

[100] Nicolas Fournel, Antoine Fraboulet, and Paul Feautrier. esimu: a fast and accurate energy consumption simulator for real embedded system. In *Proceedings of VLSI*, pages 1–6, 2007. 24

[101] Wooseok Lee, Youngchun Kim, Jee Ho Ryoo, Dam Sunwoo, Andreas Gerstlauer, and Lizy K. John. Powertrain: A learning-based calibration of mcpat power models. In *Proceedings of ISLPED*, pages 189–194, 2015. 24

[102] Matthew Walker, Sascha Bischoff, Stephan Diestelhorst, Geoff Merrett, and Bashir Al-Hashimi. Hardware-validated cpu performance and energy modelling. In *Proceedings of ISPASS*, pages 44–53, 2018. 24

[103] Mihai Pricopi, Thannirmalai Somu Muthukaruppan, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Power-performance modeling on asymmetric multi-cores. In *Proceedings of CASES*, pages 1–10, 2013. 24

[104] Quentin Forcioli, Jean-Luc Danger, Clémentine Maurice, Lilian Bossuet, Florent Bruguier, Maria Mushtaq, David Novo, Loïc France, Pascal Benoit, Sylvain Guilley, et al. Virtual platform to analyze the security of a system on chip at microarchitectural level. In *Proceedings of EuroSPW*, pages 96–102. IEEE, 2021. 24

[105] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, 1987. 24

[106] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, 2010. 24

[107] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of ISCA*, pages 84–95, 2003. 24

[108] Timothy Sherwood, Erez Perelman, Hamerly Greg, and Brad Calder. Automatically characterizing large scale program behavior. *Proceedings of ASPLOS*, 2002. 24, 28, 68, 93

[109] Radhika Jagtap, Stephan Diestelhorst, Andreas Hansson, Matthias Jung, and Norbert When. Exploring system performance using elastic traces: Fast, accurate and portable. In *Proceedings of SAMOS*, pages 96–105, 2016. 25, 26

[110] Alejandro Nocua, Florent Bruguier, Gilles Sassatelli, and Abdoulaye Gamatie. Elasticsimmate: A fast and accurate gem5 trace-driven simulator for multicore systems. In *Proceedings of ReCoSoC*, pages 1–8. IEEE, 2017. 25

[111] Anastasiia Butko, Rafael Garibotti, Luciano Ost, Vianney Lapotre, Abdoulaye Gamatie, Gilles Sassatelli, and Chris Adeniyi-Jones. A trace-driven approach for fast and accurate simulation of manycore architectures. In *Proceedings of ASP-DAC*, pages 707–712. IEEE, 2015. 25

[112] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of ISCA*, pages 29–42. IEEE, 2018. 25

[113] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Proceedings of MICRO*, pages 249–261, 2007. 25

[114] Eric S Chung, Michael K Papamichael, Eriko Nurvitadhi, James C Hoe, Ken Mai, and Babak Falsafi. Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(2):1–32, 2009. 25

[115] Otavio A de Lima, Weslley N Costa, Virginie Fresse, and Frédéric Rousseau. A survey of noc evaluation platforms on fpgas. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 221–224. IEEE, 2016. 25

[116] Danyao Wang, Charles Lo, Jasmina Vasiljevic, Natalie Enright Jerger, and J. Gregory Steffan. Dart: A programmable architecture for noc simulation on fpgas. *IEEE Transactions on Computers*, 63(3):664–678, 2014. 25

[117] Hari Angepat, Derek Chiou, Eric S Chung, and James C Hoe. Fpga-accelerated simulation of computer systems. *Synthesis Lectures on Computer Architecture*, 9(2):1–80, 2014. 25

[118] Dave Whipp. Levels of abstraction in asic modeling. http://dave.whipp.name/dv/abstractions.html. [Accessed: Sep-22]. 25

[119] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of HPCA*, pages 1–12, 2010. 26

[120] Bradley Wang, Ayaz Akram, and Jason Lowe-Power. Flexcpu: A configurable out-of-order cpu abstraction. In *Proceedings of ISPASS*, pages 147–148. IEEE, 2019. 26

[121] Gabriel H. Loh and Mark D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of MICRO*, pages 454–464, 2011. 26

[122] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *Proceedings of HPCA*, pages 283–295, 2015. 26

[123] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of MICRO*, pages 273–287, 2017. 26

[124] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *Proceedings of HPCA*, pages 476–488, 2015. 26

[125] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi. Prediction hybrid cache: An energy-efficient stt-ram cache architecture. *IEEE Transactions on Computers*, 65(3):940–951, 2015. 26

[126] Zhe Wang, Daniel A Jiménez, Cong Xu, Guangyu Sun, and Yuan Xie. Adaptive placement and migration policy for an stt-ram-based hybrid cache. In *Proceedings of HPCA*, pages 13–24. IEEE, 2014. 26

[127] Hamed Farbeh, Amir Mahdi Hosseini Monazzah, Ensieh Aliagha, and Elham Cheshmikhani. A-cache: Alternating cache allocation to conduct higher endurance in nvm-based caches. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 66(7):1237–1241, 2018. 26

[128] Jia Zhan, Onur Kayıran, Gabriel H Loh, Chita R Das, and Yuan Xie. Oscar: Orchestrating stt-ram cache traffic for heterogeneous cpu-gpu architectures. In *Proceedings of MICRO*, pages 1–13. IEEE, 2016. 26

[129] Amir Mahdi Hosseini Monazzah, Hamed Farbeh, and Seyed Ghassem Miremadi. Ler: Least-error-rate replacement algorithm for emerging stt-ram caches. *IEEE Transactions on Device and Materials Reliability*, 16(2):220–226, 2016. 26

[130] Ashish Ranjan, Swagath Venkataramani, Zoha Pajouhi, Rangharajan Venkatesan, Kaushik Roy, and Anand Raghunathan. Staxcache: An approximate, energy efficient stt-mram cache. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 356–361. IEEE, 2017. 26

[131] Gabriel Rodríguez, Juan Touriño, and Mahmut T Kandemir. Volatile stt-ram scratchpad design and data allocation for low energy. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, 2014. 26

[132] Jue Wang, Xiangyu Dong, and Yuan Xie. Oap: An obstruction-aware cache management policy for stt-ram last-level caches. In *Proceedings of DATE*, pages 847–852. IEEE, 2013. 26

[133] Mohammad Reza Jokar, Mohammad Arjomand, and Hamid Sarbazi-Azad. Sequoia: A high-endurance nvm-based cache architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(3):954–967, 2015. 26

[134] Xunchao Chen, Navid Khoshavi, Jian Zhou, Dan Huang, Ronald F DeMara, Jun Wang, Wujie Wen, and Yiran Chen. Aos: Adaptive overwrite scheme for energy-efficient mlc stt-ram cache. In *Proceedings of DAC*, pages 1–6. IEEE, 2016. 26

[135] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of ISCA*, page 24–33, 2009. 26

[136] Rajesh Kumar, Suchita Pati, and Kanishka Lahiri. Darts: Performance-counter driven sampling using binary translators. In *Proceedings of ISPASS*, pages 131–132, 2017. 27

[137] Ramon Bertran, Marc Gonzalez, Xavier Martorell, Nacho Navarro, and Eduard Ayguade. A systematic methodology to generate decomposable and responsive power models for cmps. *IEEE Transactions on Computers*, 62(7):1289–1302, 2013. 27

[138] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. Reverse engineering the stream prefetcher for profit. In *Proceedings of EuroSPW*, pages 682–687. IEEE, 2020. 27

[139] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. A formal security analysis of even-odd sequential prefetching in profiled cache-timing attacks. In *Proceedings of HASP*, pages 1–8, 2016. 27

[140] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of CCS*, pages 131–145, 2018. 27

[141] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *Proceedings of USENIX ATC*, 1996. 27

[142] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/. [Accessed: Sep-22]. 27

[143] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of ISCA*, 2009. 30

[144] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, 2010. 31

[145] An Chen. A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 125:25–38, 2016. 31, 46

[146] Ping Chi, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H Kang, and Yuan Xie. Architecture design with stt-ram: Opportunities and challenges. In *Proceedings of ASP-DAC*, pages 109–114. IEEE, 2016. 31

[147] Raspberry Pi. Raspberry pi 3 model b+. https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/. [Accessed: Sep-22]. 31, 33, 85

[148] Arm. Arm Cortex-A53 MPCore processor technical reference manual. https://developer.arm.com/documentation/ddi0500/latest/. [Accessed: Sep-22]. 34, 53, 59, 85

[149] Mediatek. MT6797 LTE-A smartphone application processor functional specification for development board. https://www.96boards.org/documentation/consumer/mediatekx20/additional-docs/docs/MT6797_Functional_Specification_V1_0.pdf. [Accessed: Sep-22]. 47, 51, 53, 62

[150] Creating disk images for full system mode. https://www.gem5.org/documentation/general_docs/fullsystem/disks. [September-22]. 59

[151] Timon Evenblij, Manu Perumkunnil, Francky Catthoor, Sushil Sakhare, Peter Debacker, Gouri Kar, Arnaud Furnemont, Nicolas Bueno, José Ignacio Gómez-Pérez, and Christian Tenllado. A comparative analysis on the impact of bank contention in STT-MRAM and SRAM based LLCs. In *Proceedings of ICCD*, 2019. 68

[152] Arm Cortex-A53 MPCore processor technical reference manual/data prefetching and monitoring. https://developer.arm.com/documentation/ddi0500/j/Level-1-Memory-System/Data-prefetching/Data-prefetching-and-monitoring?lang=en. [September-22]. 76

[153] Raspberry Pi. Raspberry pi 2 model b. https://www.raspberrypi.com/products/raspberry-pi-2-model-b//. [Accessed: Sep-22]. 85

[154] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of CODES*, pages 73–78, 2002. 105

[155] Matheus Cavalcante, Samuel Riedel, Antonio Pullini, and Luca Benini. Mempool: A shared-l1 memory many-core cluster with a low-latency interconnect. In *Proceedings of DATE*, pages 701–706, 2021. 105

Appendix

Cache Flushing

We illustrate in Figure 1 the inline assembly function we use to flush an address. This function depends on the ISA. Here, we use ARMv8-A ISA [24].

```
#   define flush(p) {\
asm volatile("dc civac, %0" : : "r"(p) : "memory"); }
```

Figure 1: *Flush function for ARMv8-A CPUs.*

LOOP Monitoring

Figure 2 illustrates the use of the PAPI library to monitor C-code measure loop and read the results from the performance counters.

```
/* Performance Counters Values */
long long values[6] = {0, 0, 0, 0, 0, 0};

/* clock variables */
clock_t start, end;
float cpu_cyc;

/* Start counting events in the Event Set */
if (PAPI_start(EventSet) != PAPI_OK)
    printf("Error: PAPI_start \n");

start = clock();
/* Reset the counting events in the Event Set */
if (PAPI_reset(EventSet) != PAPI_OK)
    printf("Error: PAPI_reset\n");

/********** Measure LOOP **********/
for(long long int loop=0; loop < LOOP; loop++){  // Outer Loop
    //
    //  Measured operations
    //
}

/* Read the counting events in the Event Set */
if (PAPI_read(EventSet, values) != PAPI_OK)
    printf("Error: PAPI_read\n");

/* Read time from time-linux library */
end = clock();
cpu_avg_cyc = (float)(end- start) * 1.391;

printf("L1D_A,L1D_R,L2D_A,L2D_R,L1D_TLB,PREF,CYC\n");
printf("%lld,%lld,%lld,%lld,%lld,%lld,%f\n",
        values[0], values[1], values[2],
        values[3], values[4], values[5],
        cpu_cyc);

PAPI_shutdown();
```

Figure 2: *Measure loop monitoring with PAPI.*

PAPI implementation

Figure 3 and Figure 4 illustrates the implementation of the PAPI library with a C-code design. The events are those introduces in Section 4.4.

```
void papi_init(){

    /* PAPI variables */
    int retval = 0;
    int code[6];
    int EventSet = PAPI_NULL;

    /* Setup PAPI library and begin collecting data from the
   counters */
    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT){
        printf("PAPI library init error! %d\n", retval);
    }

    /* PAPI Events */
    retval = PAPI_event_name_to_code("L1D_CACHE_ACCESS", &code[0]);
    if(retval != PAPI_OK) printf("Error: PAPI_event 0\n");
    retval = PAPI_event_name_to_code("L1D_CACHE_REFILL", &code[1]);
    if(retval != PAPI_OK) printf("Error: PAPI_event 1\n");
    retval = PAPI_event_name_to_code("L2D_CACHE_ACCESS", &code[2]);
    if(retval != PAPI_OK) printf("Error: PAPI_event 2\n");
    retval = PAPI_event_name_to_code("L2D_CACHE_REFILL", &code[3]);
    if(retval != PAPI_OK){printf("Error: PAPI_event 3\n");
    retval = PAPI_event_name_to_code("L1D_TLB_REFILL", &code[4]);
    if(retval != PAPI_OK) printf("Error: PAPI_event 4\n");
    retval = PAPI_event_name_to_code("L1D_CACHE_REFILL_PREFETCH",\
                                        &code[5]);
    if(retval != PAPI_OK) printf("Error: PAPI_event 5\n");
```

Figure 3: *Microbenchmark C code designed to extract memory level signatures*

```
    /* Create the Event Set */
    retval = PAPI_create_eventset(&EventSet);
    if (retval != PAPI_OK)
        printf("Error: PAPI_create_eventset (%d)\n", retval);

    /* Add Total Instructions Executed to our Event Set */
    if (PAPI_add_event(EventSet, code[0]) != PAPI_OK)
        printf("Error: PAPI_add_event 0\n");
    if (PAPI_add_event(EventSet, code[1]) != PAPI_OK)
        printf("Error: PAPI_add_event 1\n");
    if (PAPI_add_event(EventSet, code[2]) != PAPI_OK)
        printf("Error: PAPI_add_event 2\n");
    if (PAPI_add_event(EventSet, code[3]) != PAPI_OK)
        printf("Error: PAPI_add_event 3\n");
    if (PAPI_add_event(EventSet, code[4]) != PAPI_OK)
        printf("Error: PAPI_add_event 4\n");
    if (PAPI_add_event(EventSet, code[5]) != PAPI_OK)
        printf("Error: PAPI_add_event 5\n");
}
```

Figure 4: *Microbenchmark C code designed to extract memory level signatures*