UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Corso di Laurea Magistrale in Electronics Engineering

POLITECNICO
MILANO 1863

A Measurement-Based Approach for Interference Assessment on Multi-Cluster Parallel Processors

Advisor:
Prof. William Fornaciari - Polimi
Co-advisor:
Federico Reghenzani - Polimi

External-Advisors:
Dr. Luis J. de la Cruz Llopis - UPC
Dr. Jaume Abella Ferrer - BSC

Master thesis of:
Lorenzo Giuseppe Toscano
Matr. 884758

Academic Year 2018-2019

Acknowledgements

I would like to dedicated this page to thank many people that I met during my Master thesis period that made this experience unforgettable.

Firstly, I want to thank Jaume Abella Ferrer, my advisor at BSC, for the helps that he gave me. He followed me during all my the Master thesis period, giving me the materials that were needed to study for the purpose to achieve. Thanks for the patient that he had to answer all my questions and my doubts that arisen during this experience at BSC, for his kindness and for the time that he dedicated to help me.

I'm grateful to Mikel Fernández, engineer at BSC, for his help and for the tips that he gave me about the several new programming languages that I had to deal with. Thanks for his patient, for the time he spent to explain me clearly new concepts about my work, speeding up all the work that I had to do, and to repeat some of them that I didn't understand the first time.

Particularly thanks go to Dr. Enrico Mezzetti, which help me a lot with many tips about programming languages and technical documents, and to the PhD student Suzana Milutinovic, which helped me to understand some new concepts during my work. I'm grateful to Dr. Francisco J. Cazorla, leader of this group, which introduced me to all the CAOS team members the first day that I arrived, and to Dr. Leonidas Kosmidis for his advises and helps that he gave me during my path to conclude this work. Thanks very much to Matteo Fusi for his help and the contribution that he gave for this Master thesis.

Thanks to the whole CAOS team, which embraces me in their team with great politeness and congeniality, making feel me part of it from the first week that I started to work.

Last thanks are dedicated to my friends and to my family, from my parents to my grandfathers, which believed in me and they supported me each day, cheering me up when problems seemed to be too much difficult to be overcame.

Thank you all,
Lorenzo

Abstract

Nowadays, microcontrollers used in critical real-time embedded systems use mostly one core, but are being replaced with more powerful hardware platforms that implement multicore systems. Among the latter, it is possible to identify in the space domain, for instance, the Cobham Gaisler NGMP developed for the European Space Agency (ESA), which is built with a SPARC quad-core processor that has a two-level cache hierarchy. For what concerns automotive and avionics environments, very flexible platforms like the Zynq UltraScale+ EG one has been regarded as a very powerful platform for these high-performance safety-critical systems. In fact, the aforementioned Zynq board implements two multicore clusters, namely an ARM dual-core Cortex R5 and an ARM quad-core Cortex A53, as well as a GPU and an FPGA. Due to the industrial trend towards the deployment of autonomous driving in the automotive domain and unmanned vehicles in the avionics domain, boards with such multicore systems are very promising.

The use of multicores brings a concern related to contention (interference) in the access to shared hardware resources, which challenges timing verification needed to prove that all critical real-time tasks will execute by their respective deadlines. In particular, Worst-Case Execution Time (WCET) estimates for tasks need to account for the impact in execution time that contention in shared resources may have. While such analysis has been performed on relatively-simple multicores, like the NGMP, it needs to be carried out on the more powerful and complex Zynq UltraScale+ EG platform. In particular, it is required to analyze the different sources of interference for the multicore clusters and how tasks need to be consolidated so that resource sharing is performed efficiently across tasks, thus minimizing the impact on execution

time for the most critical real-time tasks.
In this Master thesis work, the measurement-based methodology developed at Barcelona Supercomputing Center (BSC) to quantify the interference that arises across cores due to contention in shared hardware resources, is ported from the (simple) NGMP platform to each of the computing clusters of the Zynq UltraScale+ EG platform. Such methodology consists in the use of small microbenchmarks that aim at stressing specific shared hardware resources to create very high contention. Hence, this thesis investigates how to produce high contention in the shared hardware resources of the Zynq UltraScale+ EG platform, thus integrating those concepts working on the SPARC V8 instruction set of the NGMP to the ARM v7 and ARM v8 instruction sets of the Zynq platform. This requires porting and adapting microbenchmarks written partly in assembly code, verifying the Performance Monitoring Unit, and analyzing the sources of contention. As final step, guidelines are devised to properly consolidate software to be implemented on the target platform in order to contain as much as possible interference on critical tasks.

Sommario

Oggigiorno, i microcontrollori utilizzati nei sistemi conosciuti come *critical real-time embedded systems* utilizzano principalmente un core, ma tendono sempre di più ad essere sostituiti con piattaforme hardware più potenti che implementano sistemi multicore. Tra questi ultimi, è possibile identificare nel dominio spaziale, per esempio, il NGMP Cobham Gaisler sviluppato per l'European Space Agency (ESA), che è stato sviluppato con un processore quad-core SPARC con una gerarchia di cache a due livelli. Per quanto riguarda l'ambiente automotive e quello avionico, piattaforme molto flessibili come quella denominata Zynq UltraScale + EG sono state considerate come piattaforme molto potenti per questi specifici sistemi *embedded* dal punto di vista della sicurezza ad alte prestazioni. Infatti, la Zynq board menzionata precedentemente implementa due cluster multicore, cioè un ARM dual-core Cortex R5 e un ARM quad-core Cortex A53, oltre a una GPU e un FPGA. A causa della tendenza industriale verso lo sviluppo della guida autonoma nel settore automobilistico e dei veicoli senza conducente nel settore dell'avionica, le piattaforme con tali sistemi multicore sono molto promettenti.
L'uso di multicore pone un problema legato alla contesa, ovvero legato all' interferenza, nell'accesso alle risorse hardware condivise, il quale mette in discussione la verifica dei tempi necessaria per dimostrare che tutte le attività che necessitano di essere calcolate in tempo reale (critical real-time tasks) verranno eseguite rispettando le rispettive scadenze. In particolare, le stime del Worst-Case Execution Time (WCET) per le attività devono tenere conto dell'impatto nei tempi di esecuzione che può avere la contesa nelle risorse condivise. Mentre tale analisi è stata eseguita su multicores relativamente semplici, come il NGMP, essa deve essere eseguita anche sulla più potente

e complessa piattaforma Zynq UltraScale+ EG. In particolare, è necessario analizzare le diverse fonti di interferenza per i cluster multicore e come le attività (tasks) devono essere consolidate in modo che la condivisione delle risorse sia eseguita in modo efficiente tra le attività, riducendo così l'impatto sui tempi di esecuzione per le attività più critiche in tempo reale.
In questo lavoro di tesi di Master, la metodologia basata sulla misurazione sviluppata presso l'azienda Barcelona Supercomputing Center (BSC) per quantificare l'interferenza che si genera tra i core a causa della contesa nelle risorse hardware condivise, viene portata dalla (semplice) piattaforma NGMP a ciascuno dei cluster di calcolo della piattaforma Zynq UltraScale+ EG. Tale metodologia consiste nell'uso di piccoli microbenchmark che mirano a stressare specifiche risorse hardware condivise per creare una controversia molto alta. Quindi, questa tesi indaga su come produrre alta contesa nelle risorse hardware condivise della piattaforma Zynq UltraScale+ EG, integrando così quei concetti che lavorano sul set di istruzioni SPARC V8 dell'NGMP ai set di istruzioni ARM v7 e ARM v8 della piattaforma Zynq. Ciò richiede il *porting* e l'adattamento dei microbenchmark scritti in parte in codice assembly, la verifica dei *Performance Monitoring Counters* e l'analisi delle fonti di conflitto. Come passo finale, sono state ideate delle linee guida per consolidare correttamente il software da implementare sulla piattaforma di destinazione al fine di contenere il più possibile l'interferenza nelle attività critiche.

X

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Nowadays, the demand of high-performance systems is increasing consistently in automotive and avionics domains since industry needs applications able to perform increasingly complex functionalities in real-time. For instance, those functionalities related to autonomous driving in automotive and unmanned vehicles in avionics require capabilities for object detection, trajectory prediction, navigation and routing among others, and those capabilities have strict (real-time) deadlines.
In this chapter, an overall introduction of the motivation and the work done for this Master thesis is given, focusing on the current challenges that this thesis is trying to address. Afterwards, the Master thesis' structure is presented, highlighting in details each chapter content.

1.1 Higher complexities in architectures implementations

The complexity of the architectures of the high-performance systems is increasing since that the programs that have to be performed by such systems are becoming more and more complex as well. This problem is due to technological barriers like [1]:

- Breakdown Dennard scaling, which pushed CPU manufactures to use multi-core processors. In fact, since 2006 MOSFET scaling is not improving the overall circuit power consumption. The number of tran-

sistors is growing up without any issues, but the overall performance is growing slower.

- Single-core performance barrier, due to the impossibility of increasing clock frequency and, consequently, dissipating the additional power.
- Dark silicon, referring to specific circuitry of integrated circuits (ICs) that cannot be turned on due to power design constraints.

Indeed, CPU architectures are more complex, leading to increase the throughput and the number of pipelines, cache and processors to be implemented. For these reasons, real-time systems show issues for what concerns the Worst-Case Execution Time (WCET) analysis, since that the classic one is not suitable for such complex architectures. Many of those systems can be classified as Safety critical systems, meaning that a failure in those systems can cause casualties (or severe injuries), harm the environment or compromise the integrity of the system itself [2]. Timing analysis issues are addressed in details in Section 2.1.

1.2 Complex architectures in safety-critical systems

In such Critical Real-Time Embedded Systems (CRTES), it is critical and mandatory to ensure the satisfaction of real-time constraints. Due to the need of increasing high performance, multi-core processors have been adopted to perform all those critical activities within expected deadlines since they provide sufficient levels of performance. However, several challenges arise on the timing behavior due to the effect of inter-task interference in such systems. In fact, when two or more cores access the same hardware shared resources, contention is experienced, reducing the overall performance of the system.
This contention has a direct effect on the execution time of tasks, possibly increasing both the average execution time and the WCET. Hence, real-time systems running on complex platforms must undergo a validation step to assess to what extent execution time may grow. The timing non-determinism

caused by hardware and software must be systematically assessed and, in particular, an upper-bound to the timing interference in necessary to provide the WCET estimation [3].
So far, such assessment has been performed mostly on relatively simple multicores such as the Infineon AURIX TC27x architecture for the automotive domain or the Cobham Gaisler LEON4 processor for the space domain. However, autonomous navigation in avionics and automotive requires the adoption of further complex platforms with larger core counts and heterogeneous computation resources (e.g. time-predictable cores, high-performance cores, accelerators).

1.2.1 Real-time systems

The board studied in this Master thesis belongs to the class of real-time operating systems (RTOS), which are employed in fields where it is mandatory to know the response time of a specific system. In fact, the system has to be deterministic, namely it is well specified in such system the real-time elaboration of specific processes in both best and worst case [4].
In practice, real-time systems have to guarantee that specific tasks respect deadlines constraints. Therefore, it is important for such systems to know the Task Execution Time (TET), which is a measure of the time required to execute specific real-time tasks. Such tasks can be classified according to their critically:

- soft real-time, when the task does not respect the prefixed deadline, the system is damaged, but it can be repaired.
- hard real-time, which corresponds to the previous statement about soft real-time, but with the important difference that the system cannot be repaired any more.
- "best effort", tasks that do not depend on real-time constraints.

From these definitions, it is possible to understand that real-time systems can be divided in two categories:

- hard systems, which are able to handle both hard and soft real-time tasks.

- soft systems, which can handle only soft real-time tasks.

1.3 Multi-cores: benefits and drawbacks

Before going into deep details of the work developed in this Master thesis, it is important to recall issues that lead to the use of multicore system and the tradeoffs involved in their implementation.
The best choice in terms of complexity and efficiency is to employ cores that are devised exclusively to perform the specific tasks they are intended to execute, but typically real systems end up executing a large variety of tasks. Hence, even if many commercial multicores use general-purpose processing cores, those cores may be specialized to some extent so that some cores prioritize performance over power or vice versa, or limit complexity, etc. Such heterogeneous cores can be also deployed together in the same platform [5] so that end users (or some software layers on their behalf) can offload their applications on those cores that are expected to maximize the metric of interest (e.g. performance, power).
In multicores processors, the resources with high utilization are replicated replicate across cores in order to increase performance, whereas those resources with a typically lower utilization are shared across cores for the sake of efficiency. For instance, the utilization of some large cache memories and memory bandwidth is relatively low for many applications. Therefore, it is common setting up processors with multiple cores that, beyond a given level of the cache hierarchy, share the rest of the hierarchy (e.g. L2 cache and main memory access channels).
Sharing some resources, despite being an efficient solution in terms of resource utilization, creates a new issue: access arbitration due to contention. This is a relevant challenge since multiple cores may attempt to access a given shared resource simultaneously, and arbitration policies must provide balanced choices or, at least, configurable arbitration so that the user can decide the most convenient way to share the resource. Policies like round-robin are often used to grant access to shared resources, so that all cores are granted access to the shared resource periodically. Still, if the amount of requests to access this shared resource is high (at least during some time

periods), requests may get delayed, thus leading to lower performance that would be obtained on a single core architecture.

1.4 Thesis Objectives

In the following subsections, the Master thesis objectives are presented. Firstly, we address goals and experiments performed on the target board with the measurement-based approach used in this work.

1.4.1 Measurement-based approach

In this Master thesis, a measurement-based methodology is proposed to define how much interference can affect execution times in a specific platform, aiming to find the maximum execution time values in different scenarios. The aforementioned methodology has the following objectives:

- Assessment of interference from the quantitative standpoint without employing complex analyses, namely based on the results obtained experimentally by timing measurements.
- Comparison between two different versions of the same processor, in order to assess which one is the best suitable for real-time purposes.
- Estimating the effect on execution time of the contention on the access to shared hardware resources to understand whether it is a suitable platform (and for what type of applications), and how software must be consolidated to make an effective use of the platform.
- Generating as much pressure as possible on specific resources of the platform under study with stressful workloads provided by specific programs that are devised for such purpose. Such workloads are intended to expose how much execution time grows when accessing different shared hardware resources with different types of assembly operations. In order to maximize the stress on these specific resources, different types of operations and parameters tuning are needed to be employed and the details are addressed in Chapter 5.

1.4.2 Zynq platform: targets

The methodology proposed so far is applied to the Zynq UltraScale+ EG platform studied in this thesis, which is a high-performance platform of the interest for several CRTES domains. In the context described in Subsection 1.2, the goal of this Master thesis is to assess whether such platform fits the needs to execute safety-critical real-time software. For this reason, the target of this thesis is to assess how much such inter-task interference (contention) can affect the performance of the Zynq UltraScale+ EG platform, focusing on the memory hierarchy of both Cortex A53 and Cortex R5 processors, which are implemented in the aforementioned multi-core system. Therefore, execution time has been studied and observed under specific experiments, which aim to stress the cache levels and memory implemented in the target processor emulating potential contention scenarios that may arise when consolidating tasks onto this processor.

1.4.3 Designing specialized benchmarks

The proposed benchmarks are perfectly suitable for the approach described so far and for this reason they are the ones used in this work. Specialized micro-benchmarks are designed for the Zynq UltraScale+ EG board to optimize the measurements, as in the case of NGMP Multi-core Processor [6], since that they can be more compact and precise to stress just some specific resources of the target platform w.r.t. the benchmarks of state-of-the-art, which can bring to imprecise or, in the worst case, to unreliable results.
In order to achieve the objectives explained above, some requirements have to be fulfilled by these benchmarks:

- Algorithms have to be written in C and assembly programming languages. This improves the speed and the quality of binary generation, guaranteeing that binaries perform exactly their intended activities, i.e. they stresses each components as expected. Such type of fine control would be very difficult to obtain using high-level programming languages.
- Simplicity and flexibility, i.e. they can easily be implemented in other processor architectures with minimal modifications and, as done in this

thesis, can be ported to multi-core systems with different cache hierarchies with negligible additional effort.

- The amount of memory used by the algorithms must be minimized in order to reduce the space occupied in the different cache levels, leaving the major part of the cache memory free for data. In this way, the contention can be then studied with the data patterns accessed without side effects caused by the code footprint.

1.5 Master thesis structure

The second chapter presents some background needed to understand the key aspects of this thesis. This includes details on how timing validation is performed for CRTES, a brief introduction on cache memories structure and their mechanisms, as well as an overview of the main features of the processor architecture employed by the Zynq UltraScale+ EG platform.
For what concerns the third chapter, it is addressed the state-of-the-art on micro-benchmarks, with particular emphasis on those devised to stress specific timing behavior relevant for timing analysis.

The fourth chapter focuses on describing the methodologies chosen to perform the experiments on the target platform. In particular, this chapter explains the general working principle for which the micro-benchmarks are devised, followed by the detailed description of the ones employed in this work. The concept of Performance Monitoring Counters (PMCs) is introduced, following the example of the ARM architecture and then moving the discussion on the ones relevant for this thesis use-case, i.e. the PCMs defined in the Zynq UltraScale+ EG platform.

In the fifth chapter, the main parts of the developed benchmark codes used for single-core experiments are described in details. How to exploit the PMCs implemented in the aforementioned ARM architectures is presented, together with the full description of their use in the employed algorithms.

The sixth chapter shows the results obtained of all the performed ex-

periments, considering both the single-core and the multi-core scenarios. Subsequently, the chapter describes how to analyze and draw conclusions from the obtained results, using the Zynq use case results as clarifying examples.

Finally, in the last chapter, the conclusions are presented with some proposal for possible future works.

Chapter 2

Background

In this chapter the relevant background theory for this work is presented. We start with Timing Analysis in Section 2.1, which describes generally how WCET of real-time programs can be estimated and problems related with such estimation. Subsequently, typical cache memories employed in modern processors are described, focusing on cache structure and policies used by ARM processors implemented in the target platform. Section 2.3 presents the most important instructions of SPARC instruction set, which are very similar to the ARM ones, concluding with Section 2.4 where Zynq UltraScale+ platform structure is described in detail focusing on the specific processor structures.

2.1 Timing analysis

The estimation of the WCET of real-time programs has been investigated for decades. Two main paradigms can be found in the literature on how to estimate the WCET: Static Timing Analysis (STA) and Measurement-Based Timing Analysis (MBTA). STA relies on the availability of a timing model of the processor on which to calculate the overall timing needed for the whole program to execute each instruction, without requiring a simulation of it. In particular, the timing model of the target processor architecture is built identifying each hardware component, as well as their behavior and relationships regarding to timing behavior. The representation of the source code of the program (in the form of assembly instructions) is analyzed to

model both, the execution path flow and the data flow of the program, so that STA can account for the behavior of the different execution paths and potential data values.
In general, abstract interpretation builds upon unknown information, such as unknown input data values, which may affect memory access patterns and execution paths. This leads to an explosion of potential states that can be reached after the execution of every instruction. Such states are used to understand the behavior of processes in a specific computer system and to identify in which "state" the process is located. The most typical primary process states are described in the following and they are presented in the state diagram (2.1), noting that such processes are "stored" in the main memory.

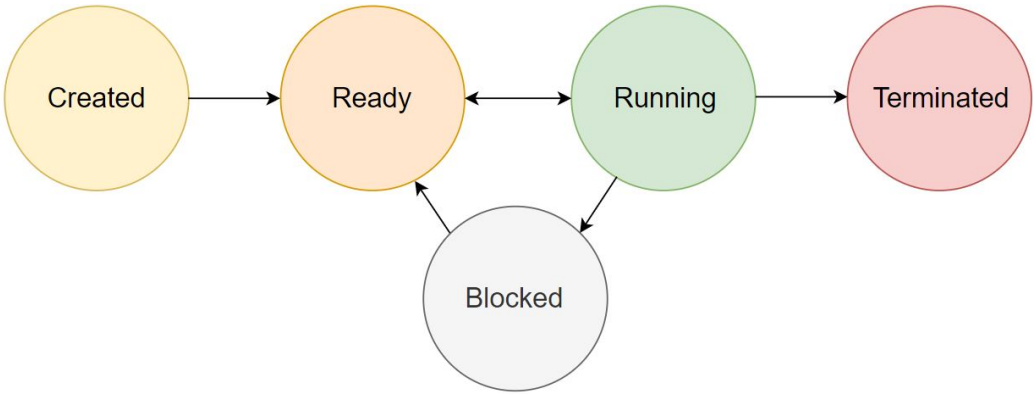

Figure 2.1: State diagram of most typical primary process states. Possible transitions from one state to the other one are pointed out with arrows.

- Created - It states a new process, i.e. when a process is created for the first time and never executed.
- Ready - Right after the "Created" state, there is the "Ready" or "Waiting" one, which indicates that the process has been loaded into main memory. In this condition, there can be many processes and a ready queue is used to make CPU able to execute each of them one at a time.
- Running - It is the running state and it defines when a process is executed by CPU.

- Blocked - When a process is waiting for an event and it requires an external operation to trigger such event, it is "blocked" until an external operation is applied. This is the case when an I/O device is not available, like DVD, HDD and printer.
- Terminated - A process is moved to the "terminated" state if either its execution is completed or it is killed.

STA makes the problem tractable by making "safe" (i.e. pessimistic) assumptions that allow merging different states into few ones that lead to the highest execution times possible. For instance, if the address accessed by a given load instruction cannot be determined, instead of modelling all potential states corresponding to all potential addresses that could be fetched, STA typically assumes that the access is a miss, that no useful data is fetched into cache, and that some cache contents are evicted (either a cache line or a full cache way). Overall, STA trades complexity for pessimism to keep computational cost tractable. A survey on timing analysis, with particular emphasis on STA, can be found in [7].
STA has increasing difficulties with increasingly complex hardware, as analyzed in [8], and simplifying the analysis process by merging states leads to potentially high pessimism. In general, the higher the hardware complexity (e.g. by using cache memories and multicores), the larger the number of potential states and execution time variation across states, and the higher the pessimism to merge states. It is important to point out that processor timing models are typically derived from processor specifications that may have thousands of pages. Manuals with very high number of pages make more difficult to perfectly define correct timing models of a specific processor, possibly jeopardizing their reliability. Moreover, those specifications are often subject to errata, further increasing the uncertainty on the reliability of STA [8].
On the other hand, MBTA is an hybrid approach that combines execution time measurements of the program under analysis on the target hardware platform with static program analysis techniques to estimate the WCET. MBTA approaches have also several sources of uncertainty due to the difficulties to guarantee that the execution conditions, inputs and processor states, are stressing the worst-case scenarios. Only in such conditions the WCET can be observed. For instance, generating inputs that trigger the

highest number of loops iterations, the worst paths in conditional constructs (e.g. if-then-else, switch), the worst memory patterns, etc. is in general out of reach for end users. The facts that measurement collection is affordable and the tightness of WCET estimates makes MBTA attractive for industrial uses [9]. Commonly, the inputs used for functional test of the software, which typically trigger the different operation modes of software, can be reused to obtain execution times relevant for WCET estimation. To increase the reliability of WCET estimates, some approaches can be followed: by adding an engineering factor to the maximum observed execution time (MOET) (e.g. MOET+20%) or by applying more sophisticated logic (e.g. using some static information about path analysis as done by tools like RapiTime [10]). Measurement-based analyses are a hot research topic, because of several challenges that are still open [11] and some methods are controversial [12].
However, while MBTA has been proven to be quite reliable for single-core processors, multicore processors bring new difficulties due to the potential contention that the task under analysis can experience in the access to shared hardware resources. Thus, specific microbenchmarks causing high levels of contention have been considered to obtain execution time measurements relevant for WCET estimation in multicores [6].

2.2 Cache memory

The access to memories is often a bottleneck for all modern processors since they generate latencies in the overall system. To reduce this problem, cache memories are implemented in processors to achieve very low latencies, since they are fast enough to serve data and code at high speed. Unfortunately, cache memories have to be small enough to achieve such speed and their cost is quite high, which brings to many technological issues. For this reason, one of the most important parameters to evaluate the performance of a processor is the memory latency. The introduction of cache memory reduces both power consumption and the number of external memory accesses performed by the system to the main memory, avoiding slow downs in the overall system [13].
Typically, the cache hierarchy implemented in modern processors is composed

of several layers, as represented in Figure (2.2), in order to achieve high performances.

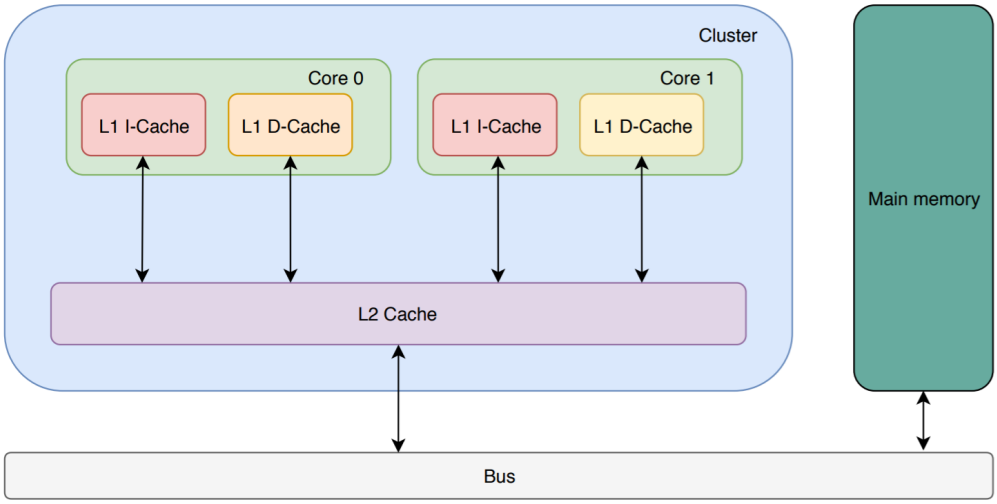

Figure 2.2: Typical cache memory arrangement of modern processors

In such Figure, two cores are represented with two levels of cache, which are the first level (L1) and the second one (L2). Note that L1 cache is divided into Instruction Cache (I-Cache) and Data Cache (D-Cache), namely a modified Harvard architecture within which instruction and data buses are separated in order to reduce interference among them building on the fact that instruction and data access streams are naturally decoupled in program execution. Such buses are internal since they are used for connecting and interacting with internal components and they are not intended to communicate with external components like the main memory. The L2 cache is a resource typically implemented on multi-core, which are structures used to speed up the performance and they group several cores sharing both instruction and data.

When a memory location is accessed for the first time, there is no improvement in terms of access time to the memory. In fact, it is not yet present in the cache hierarchy and must be fetched from the main memory block, and it goes through the interconnection network (e.g. an AMBA AHB processor bus in the case of ARM Cortex-A53) to reach the most internal level cache, typically being stored first in the shared L2 cache and then in the L1 Cache.

Subsequent accesses to the same data memory location are much faster since the data is already available in the internal cache and there is no need to fetch it from the main memory [13].
Depending on the fact that data is found or not in the cache, the two previous cases can be distinguished as follows:

- **Read cache hit**: the data is available in the cache, enabling a rapid access to it.
- **Read cache miss**: the data is not available in the cache and it has to be sought in higher cache levels or directly in the main memory. Then, such data has to be copied in the cache. These steps lead to slow memory accesses, reducing system performance.

Cache hit and cache miss for writing operations change depend on the write policies of the cache implemented in a specific system and they are described in Cache policies section (2.2.2).

2.2.1 Cache structure

The traditional types of cache structures are:

- Cache Fully Associative, thanks to which each location in main memory can be stored in any position of the cache. In general, allowing any data to be placed in any cache location requires expensive searches upon an access to determine whether the data is available in cache or not. Hence, this type of caches is expensive and used only for small caches.
- Cache Direct Mapped, which is the opposite of the previous cache structure. In fact, in this case, each location in main memory can be mapped in just one cache entry. Hence, searching for a given data is a cheap process since a single location needs to be checked. Thus, such structure is very convenient for large caches. However, the fact that each data has a predetermined location leads to cache conflicts where few data contend for the same cache entry despite large parts of the cache are empty.

- N-way Set Associative cache is a combination of direct-mapped and fully-associative caches. Each address is placed to a predetermined cache set, as in direct-mapped caches, but in each set there are multiple entries (the same number in each set) and the data can be freely allocated in any line within their set, as in fully-associative caches. Hence, the degree of associativity (number of entries per set) determines the performance and efficiency of these caches. In general, they are the preferred choice for large caches since they allow obtaining most of the benefits of fully-associative caches with costs close to those of direct-mapped ones.

The N-way set associative cache is the structure mainly implemented in almost all ARM processors [13]. In fact, in this thesis all the cache levels of the considered processors are N-way set associative caches. For this reason, the following paragraph describes in details such cache structure.

N-way set associative cache

A N-way set associative cache structure is conceptually arranged into S sets (rows) and N columns (ways), as shown in Figure (2.3). Each cell represents a cache line. Each location in main memory can be mapped to one and only one set, but its content can be placed in any of the cache lines (ways) in that set. Therefore, the lookup of a specific data is made in a group of N cache lines (those within the corresponding set).
Cache lines have a specific size (in bytes). In general, for the sake of implementation efficiency, all parameters are powers-of-two, and the size of the cache is determined as the product of the number of ways (N), the number of sets (S) and the cache line size (B). For instance, a 4-way cache with 128 sets and 64-byte cache lines is a 32 kB cache.

2.2.2 Cache policies

The replacement of cache lines is governed by several policies. This Section provides insights on those relevant to this thesis: replacement policies and write policies.

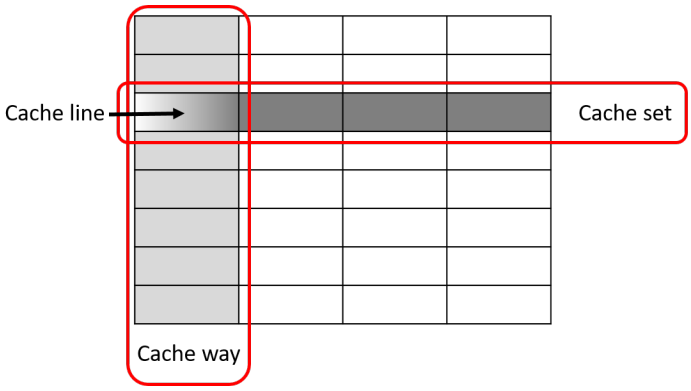

Figure 2.3: structure of 4-way set associative cache

Focusing on the replacement policies, there are two quite popular approaches employed in many systems:

- Least-Recently-Used (LRU), which aims to replace the data of a cache line that is the least recently used out of all those in the set. This policy leverages the time-locality, i.e. the fact that there is a high change to access the same location in a short time-span.

- (Pseudo-)Random, which ensures that on a miss event, a cache way in the corresponding set is randomly evicted to make room for the new cache line [14]. Since this policy is used in some platform considered in this thesis, we analyze it later in detail.

Regarding the write policies:

- Write-back (WB), which updates the main memory just when a cache line is evicted in the cache [13]. Thus, on a write operation, if the data is already present in the cache, it is only updated in the cache and not in memory unit. This policy is quite convenient in terms of performance, since many memory accesses are avoided, but it is more complex to implement since delayed memory writes need to be managed.

- Write-through (WT), which aims to update both the cache and the main memory of the system upon a write operation [13]. Vice versa to the previous case, this policy is less efficient but very easy to implement.

Pseudo-Random Replacement policy

If in a N-way set associative cache a Random Replacement policy is implemented, the probability that a specific cache line will be evicted is equal to $\frac{1}{N}$ for each set [14]. Hence, cache hits or misses are, in theory, truly probabilistic within the cache set. It has been shown in [14] that the hit probability (P_{hit}) for a specific access, for instance A_j, in an access sequence to its cache set like $< A_i, B_{i+1}, ..., B_{j-1}, A_j >$ is obtained using the following equation:

$$P_{hit_{A_j}} = \left(\frac{N-1}{N}\right)^{\sum_{k=i+1}^{j-1} P_{miss_{B_k}}} \tag{2.1}$$

where A_i and A_j are accesses to the same cache line, B_k corresponds to the accesses that are performed to cache lines different from the one where is present A and $P_{miss} = 1 - P_{hit}$ is the probability to have a miss for any access. $P_{hit_{A_j}}$ is the probability that A is survived after all evictions caused by B_k, while the probability that A is survived when one random eviction occurs is equal to $\frac{N-1}{N}$. Therefore, increasing the number of evictions in the cache set, increases the probability to evict A as well [14].
While this is the theoretical behavior of random replacement policies, actual implementations in processors may use poor pseudo-random number generators that do not produce fully random replacements. As shown later in the evaluation section, experimental evidences show that the random replacement policies implemented in the target platform may not be effectively random.

2.3 SPARC instruction set

In order to correctly devise the microbenchmarks for the ARM processor architectures, we initially studied the microbenchmarks employed for the Next Generation MicroProcessor (NGMP). This architecture implements a SPARC V8 quad-core processor, which was developed by Cobham Gaisler for the future European Space Agency (ESA) missions [6]. For this reason, we analyzed some instructions provided by the SPARC V8 instruction set in detail [15]. In this section, we present the most important ones of the instruction set, in particular the memory write and read operations, since they

are specular to the ones in the ARM instruction set that produce analogous access patterns:

- `ld` stands for *load*, which has the following syntax:

$$\texttt{ld}\ [r_s],\ r_d \tag{2.2}$$

  where r_s is the *source* register and r_d is the *destination* one.
  This is a memory read operation and it fetches from the main memory the data that is stored in the memory address specified in the register r_s. Afterwards, the fetched content is saved in the register r_d.

- `st` stands for *store*, which has the following syntax:

$$\texttt{st}\ r_s,\ [r_d + \texttt{offset}] \tag{2.3}$$

  where the same notation used for (2.2) is employed.
  The store instruction is a memory write operation, which has the goal to deliver into a specific memory address defined in the register r_d of the main memory the value of the register r_s. Note that it is possible to specify an *offset* value, which is a constant number added to the variable register value. This feature is very useful for programs that perform many store operations in sequential memory addresses.

2.4 Zynq Ultrascale+

Development of specific microbenchmarks is the focus of this Master thesis to help WCET estimation of the target hardware platform that brings increasing difficulties due to the use of multiple and heterogeneous core clusters. However, such platform offers high computation power, which is of high interest for many industries such as those in the avionics and railway domains among others.
The Xilinx® UltraScale multiprocessor system-on-chip (MPSoC) implements in the same device both a processing system (PS) and user-programmable logic (PL).
For what concerns the PS, it features three main processing units:

- Cortex-A53 application processing unit (APU)
- Cortex-R5 real-time processing unit (RPU)
- Mali-400 graphics processing unit (GPU)

In the target platform structure represented in Figure (2.4) the first two individual blocks are presented, which are the ones of interest for the work in this Master thesis. In particular, the Zynq UltraScale+ board includes two clusters of processors that feature two different architectures: the ARM v8 architecture-based 64-bit for the APU and the ARM v7 architecture-based 32-bit for the RPU. Instead, the GPU is not subject of this work.

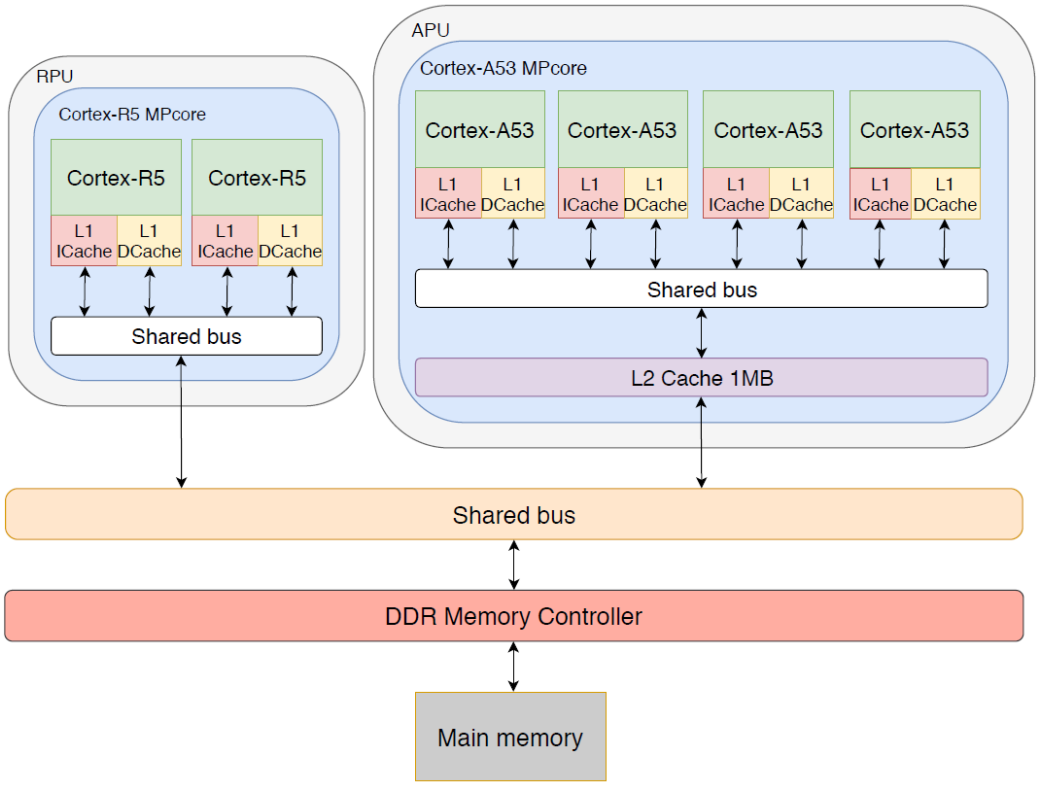

Figure 2.4: Zynq UltraScale+ platform structure: APU and RPU embedded blocks with their relative processors.

In the following subsections, Cortex-A53 and Cortex-R5 cache features are described in detail.

2.4.1 ARM Cortex-A53 Processor

The APU of this platform consists of four Cortex-A53 MPCore processor cores and a L2 Cache, which is a shared resource among these four processor cores.
The Cortex-A53 processor has a modified Harvard architecture with internal different buses both for instructions and data. For this reason, the private Level-1 (L1) cache is divided in Instruction cache (I-cache) and Data cache (D-cache), both implemented as Set associative caches. Instead, the external bus is still a Von Neumann architecture, which delivers both instruction and data.
Subsequently, we describe the main features of the APU:

- ARM v8-A architecture instruction set.
  - Possibility to choose either A64 instruction set in 64-bit mode or A32/T32 instruction set in 32-bit mode.
- I-Cache and D-Cache are separated.
- Cache size of both L1 caches is 32 KB.
- Cache line size is fixed to 16 words, which corresponds to 64 bytes, both for L1 I-Cache and L1 D-Cache. Hence, each of those caches has 512 cache lines of 64 bytes each.
- L1 I-Cache is implemented as a 2-way Set associative cache, whereas the L1 D-Cache is implemented as 4-way Set associative cache. Hence, the L1 I-cache has 256 sets with 2 cache lines each, whereas the D-cache has 128 sets with 4 cache lines each.
- Level-2 (L2) Cache size is equal to 1 MB.
- The replacement policy implemented for L1 caches is the Pseudo-random one, while for L2 cache we do not have any information from the processor specifications.
- For what concerns the cache update policies, the L1 Data and L2 caches use write-back policy. Since the I-cache does not modify the code stored, it does not need any write policy.

2.4.2 ARM Cortex-R5 Processor

The RPU is a cluster that includes a dual-core Cortex-R5 for real-time processing. Also in this case the internal structure has a modified Harvard structure and N-way set associative caches.
The main features of this processor are:

- ARM v7-R architecture instruction set.
  - The available instruction set is A32/T32.
- Instruction and data caches are separated thanks to the implemented Harvard architecture. The access to the main memory is instead performed through an external bus employed for both instruction and data.
- Cache size of both L1 caches corresponds to 32 KB.
- Cache line size is fixed to 32 bytes, which corresponds to 8 words of 4 bytes each, both for instruction and data caches. Hence, each cache has 1,024 cache lines.
- L1 Instruction and Data caches are 4-way Set associative. Hence, they have 256 sets with 4 cache lines each.
- Level-2 (L2) cache is not present in this cluster.
- Caches of the Cortex-R5 cores implement Pseudo-random replacement policy, which is the same implemented in Cortex-A53.
- The write-back policy is implemented in the Cortex-R5 L1 Data cache.

Chapter 3

State of the art

As explained before, some industries rely on MBTA for WCET estimation, and some approaches based on the use of microbenchmarks to model multicore contention have been found appropriate. Hence, in this section we review some of the main works in the area of microbenchmark development to induce high stress conditions in multicores. In particular, approaches generating stressful scenarios consider not only performance stressful conditions, but also power and temperature conditions as a means to assess relevant non-functional metrics of processors and applications. Due to their importance, these last banchmarks are discussed briefly in the last section of this chapter even if they are not in line with this Master thesis.

3.1 Performance Stressing Benchmarks

In the context of critical real-time systems, and with particular emphasis on commercial off-the-shelf (COTS) multicore processors, software testing has been largely exploited to test functional and non-functional properties of software. In particular, those tests are run during the analysis phase of the system, in early design stages, when many applications are still under development. In the case of WCET estimation, the objective is obtaining WCET estimates during the single task testing (i.e. when the task under analysis has been implemented), without the need of waiting for other tasks (e.g. tasks that will run concurrently) to also be implemented. This allows assessing whether execution time budgets allocated to tasks suffice to run

them and, if this is not the case, address this issue as soon as possible. Detecting timing violations during late design stages incurs high costs and may impact time-to-market.

Therefore, since tasks running concurrently are unknown during WCET estimation, assumptions need to be made on the contention those tasks can generate. Usually, this has been accounted for using simple programs, called microbenchmarks, that place specific amounts of contention on specific shared hardware resources. For instance, one may develop a microbenchmark by reading constantly from memory to generate high contention in the access to memory to measure how sensitive the task under analysis is to such contention.

Several strategies to create contention relevant for WCET estimation exist. Some authors created those types of microbenchmarks to study the impact of contention on high-performance Intel and AMD processors [16]. While those processors are generally regarded as inappropriate for critical real-time embedded systems due to the large number of hard-to-control sources of execution time variation, the strategy followed to develop microbenchmarks has been later reused to evaluate more appropriate processors. In particular, authors developed microbenchmarks with simple loops sufficiently small to fit in L1 Instruction caches, but sufficiently large so that the overhead to increase the loop counter and jump was negligible. Then, those loops contain a sequence of instructions of the same type accessing a specific shared resource (e.g. the second level, L2, cache) with the aim of creating the highest contention possible.

A similar strategy was applied on the Cobham Gaisler LEON4 processor [6], whose target is the Space domain. Experiments performed with microbenchmarks revealed that an early design of the LEON4 allowed to cause 20x slowdowns on a 4-core multicore. Microbenchmarks showed that, by using a non-split bus, the worst contention impact occurred when the task under analysis was performing sustained L2 hits whereas all other contenders were performing L2 misses. Upon an access to L2, the non-split bus gets locked by the accessing task and it is not released until the transaction completes. Hence, each (short) L2 hit of the task under analysis may have to wait for 3 memory accesses (L2 misses) caused by each of the 3 contender tasks. This behavior turned to be particularly exacerbated for store operations whose

latency for L2 hits is very low, whereas sustained store L2 misses caused 2 memory accesses each: one to evict a dirty line modified by previous stores and another to fetch the line accessed by the store itself.

While maximum stress contention scenarios are relevant for WCET estimation, they may be overly pessimistic for some platforms and applications. Hence, some authors extended microbenchmarks for the LEON4 platform to consider specific amounts of contention [17]. Authors build upon the concept of partial time composability instead of full time composability, meaning that the obtained contention bounds are only valid under specific amounts of contention. In particular, authors show how to account for specific access counts to each shared hardware resource so that the obtained contention bounds are valid as long as contenders do not exceed those access count bounds. This approach is particularly useful when some information about contenders is available, so that specific access count bounds can be set with the aim of upper bounding real access counts but without having to account for the maximum number of accesses possible. This approach builds on the use of maximum stress microbenchmarks to derive per-resource latencies, which are later used to statically model the maximum contention possible under specific loads. Also, this work devises microbenchmarks performing specific access counts as a mean to verify that the statically estimated contention bounds match the worst-case observed values.

A similar approach has been followed for the Infineon AURIX TC27x processor family [18]. Due to the particular characteristics of this platform to count events, an Integer Linear Programming (ILP) model has been developed to obtain upper and lower bounds to the values of PMCs counting the stall cycles. On the other hand, microbenchmark technology to measure maximum latencies and to create contention scenarios is the same of the previous work [17].

Such strategy has also been considered for the Qualcomm SnapDragon 810 processor within the framework of the H2020 SAFURE project [19]. Such processor has been regarded as appropriate for the telecommunications domain and used in many embedded systems such as the Sony Xperia smartphone. While the strategy followed for this platform has been analogous to that for the LEON4 and AURIX processors, results showed that the architectural documentation is incomplete and inaccurate for this platform [20].

In particular, the prefetcher could not be disabled and events monitored by PMCs were insufficient to estimate contention with meaningful accuracy. Thus, this platform has been regarded as inappropriate unless an improvement of available documentation so that the platform can be mastered to a sufficient extent.

Other authors attempted to model the contention in the interconnection network of the NXP P4080 processor – relevant for avionics and railway domains – by developing similar microbenchmarks [21, 22]. Their work revealed that contention is not linear with the number of cores, thus exposing that, while this 8-core architecture may seem to be symmetric, it is not. In particular, experiments revealed that contention caused by some cores was higher than that caused by others, thus exposing the fact that the interconnection network organizes cores into two different clusters, and contention between cores of the same cluster may be higher than across clusters. Further analysis of the NXP P4080 has been carried out with microbenchmarks assessing other types of execution time interference across cores, and revealing that, for instance, some asymmetric behavior is caused by cache coherence protocol, even in case tasks do not share any data [23].

Other approaches have focused on performing some forms of stochastic analysis of contention with the aim of identifying typical timing behavior under high contention on shared hardware resources, but without explicitly considering the worst case, thus providing a family of testing techniques building on the correlation of PMCs [24]. Those approaches build also upon the use of microbenchmarks to expose dependencies across events which, ultimately, requires the creation of some microbenchmarks producing high contention to reveal dependencies in the access to shared hardware resources. A similar approach building on similar types of microbenchmarks have been devised with the aim of applying statistical techniques such as principal component analysis to predict the worst contention bounds of critical real-time tasks on multicores [25].

Finally, some authors have attempted to model contention at late design stages by running simultaneously tasks that may contend against each other, modifying their time alignment (i.e. their relative starting time) to account for the worst – yet realistic – contention that specific tasks can cause on each other [26].

3.2 Power and Thermal Stressing Benchmarks

Stressing benchmarks have been used in other contexts with the aim of predicting other non-functional metrics such as power and temperature. Next we provide few illustrative examples of those applications of microbenchmarks.
Thermal analysis by means of software-based solutions has been mostly considered for post-silicon validation of processors with the aim of identifying the Thermal Design Power (TDP), which is the maximum sustained temperature that a processor can produce. A proper identification of the TDP is key for chip manufacturers to size the cooling solution needed to keep the processor under specific temperatures. Triggering the TDP typically requires the development of the so-called power virus programs, which create sustained high-power activities [27, 28]. For instance, floating point operations have been shown to consume high energy, while allowing virtually executing one such operation per cycle per core. Hence, microbenchmarks building on those types of operations are often used to trigger specific high-temperature scenarios.
In essence, benchmarks intended to trigger high execution times (due to contention) or high temperature have similar structures (loops with specific patterns that repeat many times) varying the type of operation that is executed sustainedly depending on whether the objective is to create high contention or high temperature.
So far, we presented the benchmarks given by state-of-the-art, which are widely developed for many platforms, but they are not suitable to study aspects of the platform that we want to address. For this reason, we proceed presenting and explaining in details the development a new set of benchmarks together with a new measurement-based approach to study with more accuracy features of the target multicore system that cannot be addressed with the ones described in this chapter.

Chapter 4

Methodology

In this chapter, the methodology used for the experiments performed is explained in detail, explaining why and how such experiments are performed in order to achieve the expected objectives. In particular, different microbenchmarks and functions devised both in C and assembly programming languages are described in details, focusing on their features and their ability to detect the sources of unpredictability.
The proposed methodology consists of designing specific microbenchmarks to collect empirical evidence directly measuring the considered metrics on the target platform. Those microbenchmarks are executed in two scenarios: (1) in isolation and (2) together with co-running microbenchmarks (either identical or different) onto other cores. The results of these experiments are collected reading the PMCs available in the platform under study. In order to increase the results reliability, the following steps have been followed:

- *Interference-free run*: These tests are performed with all the cores (except the one where the microbenchmark runs) in power-down mode, in order to avoid any type of inter-core interference. In this way, it is possible to verify that both the microbenchmark and the target main core are properly working. Moreover, it is possible to verify if the features represented in the corresponding manuals are correct or not. The rationale behind these experiments is that microbenchmarks have a known behavior on the specific hardware: approximate values for some metrics such as executed instructions, cache accesses, memory accesses, etc. are known a priori and can be used as baseline values for

subsequent analyses.

- *Study of PMCs behavior*, understanding if they work properly, i.e. if the results obtained with measurement-based approach are the ones expected from the PMCs features reported in manuals. When the measurements performed show that PMCs are behaving differently as expected by the reading of manuals, PMCs are not working properly. It is carried out also the evaluation of how much noise and other type of disturbances can affect the behavior of PMCs. In this way, it is possible to assess that such counters are reliable enough to be used for achieving the goal of providing an useful and representative metric. This is an important step since it is not unusual having documentation where PMC description is scarce, so that their definition is ambiguous. In fact, manuals do not give always all PMCs information and many times they provide *"IMPLEMENTATION DEFINED control"* [29], compromising the complete study of the PMCs profile. A possible example is when PMCs have an unexpected poor accuracy that is shown by the results obtained using the measurement-based approach while instead they should be accurate. Also, since PMCs are not part of the functional behavior of the system, they are often less debugged that other parts of the processor and may have unexpected behavior.

- *Collecting PMCs data when all cores are active*, thus verifying that all PMCs can be interfaced properly even when all cores in both clusters are running their own private workloads. This is important to verify that events corresponding to shared hardware resources can be counted on a per-core basis, thus avoiding interference on PMC values themselves.

- *Run several experiments with all the cores in running mode, using the same microbenchmarks devised before.* Each experiment is distinguished among the other ones since that different microbenchmarks are executed in parallel. Therefore, there are processor cores behaving as "contenders" and one processor core that is the "task under-analysis". Contenders are the processor cores that can generate inter-task interference due to the contentions that arise in the hardware shared resources

with the main core under observation. These are the most relevant experiments since they represent the worst-case scenario that a given critical real-time task may experience in the system during operation.

- *Collecting data of the PMCs results when all the processors are executing simultaneously*, comparing them with the ones obtained for single cores. While the previous set of experiments exposes the actual behavior in terms of execution time, this set of experiments shows why execution time in parallel operation differs from that in single core operation. Thus, from the results of this set of experiments it is possible to detect what type and degree of interference occurs in each shared hardware resource. Such information is crucial to understand what the most convenient way to consolidate tasks is. In fact, based on the comparison between PMCs results obtained for all cores and for single cores, it is possible to define guidelines on which type of instruction is better to use among the other ones to get the lowest achievable execution time of specific tasks. Following these guidelines, software can be consolidated with the best performance achievable, by distinguishing in each scenario the instructions that guarantee the lowest number of contentions in shared hardware resources.

4.1 Microbenchmarks

As explained before, microbenchmarks can be used to evaluate performance of multi-core architectures. Reasons of such statement are confirmed from the fact that they are developed in C/C++ programming language, which is simpler than assembly and than high-level programming languages that can stress more the resources. Also, industry preference for quantitative evidence on the target platform is also a plus for this approach since that microbenchmarks are suitable to make measurements and collecting quantitative results. The size of each microbenchmark (in terms of code footprint) should be small enough to fit in the instruction cache, in order to avoid the shared resource interference caused by the instruction fetches from the higher cache levels or the main memory. The experiments are mainly focused on generating different loads in the memory hierarchy by controlling the amount and rate of

data transfers.
Algorithm 1 represents the conceptual schematic employed for the microbenchmarks proposed in this Master thesis. The parameters `A` and `B` represent the input arguments of the microbenchmark, where `A` depends on the type of operations performed, while `B` is the number of iterations of the main loop to be executed. Afterwards, benchmark-specific memory allocation and initialization procedures take place. In some experiments, memory initialization requires setting specific contents in memory so that the main loop of the microbenchmark maximizes the number of accesses per cycle. To this aim, it is employed the *pointer chasing* technique, which is addressed in details in the next chapter.
Finally, the loop body represented in such algorithm is mostly written in machine-specific assembly code (ARMv8 and ARMv7) in order to directly control the memory-related instructions without the unwanted compiler optimizations and side-effects. The type and number of instructions in each microbenchmark depend on the specific interference goal we want to achieve, i.e. on the location and magnitude of effects on shared resources. Among them, it is possible to distinguish two classes used in this work: the memory instructions that read from the main memory or cache (loads) and the ones that write to the main memory or caches (stores).
As shown in the next chapter, for implementation simplicity and flexibility purposes, the initialization/allocation phase and the main loop can be decoupled in different functions increasing the re-usability of the code across different microbenchmark types.

4.2 Performance Monitoring Counters

The quantitative evaluation of the results obtained with the microbenchmarks described in the previous section (4.1) is performed through the use of the *Performance Monitoring Units* (PMUs) existing in each core. Each unit includes a set of *Performance Monitoring Counters* (PMCs).
The PMCs available in both the Cortex-A53 and Cortex-R5 processor cores, are useful to verify that the experimental results correspond to the expected ones and to analyze the magnitude of interference in shared hardware

Algorithm 1 General structure of the implemented microbenchmarks

```
void Name_Microbenchmark (A, B)
{
 Start memory allocation procedures
   (...)
 End of such operations

 Start assembly code
    Start Loop of B iterations
       Memory Read/Write Instruction (A -> Register/Main Memory)
       Memory Read/Write Instruction (A -> Register/Main Memory)
            (...)
       Memory Read/Write Instruction (A -> Register/Main Memory)
    End Loop of B iterations
 End assembly code
}
```

resources.
Such PMCs can be set to measure different events, so it is important to properly configure them to count the events of interest. Then, PMCs must be enabled or cleared right before the execution of the microbenchmark and disabled right after in order to monitor only the activity of the single microbenchmark. Once this is guaranteed, the values obtained for a given event type with PMCs can be compared against the expected values. As first step, experiments were performed in order to check if the PMCs of such processor works properly, i.e. comparing the experimental results obtained by PMCs counting for one event with the expected ones. When the two kind of results are matched, the PMCs counting for such event is reliable, so that knowledge can be built on top of its results when collecting data for single core and multi-core experiments. Proceeding with this approach, it is possible to define the reliability of the PMCs counting for all the implemented events. After that PMCs are confirmed to properly work, a C programming language interface is built in order to access the PMCs values from the microbenchmarks code. The PMCs interfacing has been integrated in the function where the microbenchmark was defined in order to improve the reliability of the measurements of each event counted. In particular, PMCs interface is nested in each microbenchmark and it is divided in two main

blocks: the first one placed right before the core of the specific microbenchmark, i.e. the beginning of assembly code, and the other one right after the end of iterations made by such assembly code.
Several PMU, or PMC, events are available for these architectures and in the following paragraphs we list the most important ones that have been used for experimental evaluation. Firstly, the common events are described, followed by the Cortex-A53 and Cortex-R5 specific events.

4.2.1 Common events

1. L1D_CACHE - L1 Data cache access, which counts how many times the L1 Data cache is accessed by any operation. Read and write accesses are not distinguished.

2. L1I_CACHE - L1 Instruction cache access, which counts instruction memory accesses to both the L1 Instruction cache and L1 Instruction memory structures.

3. L1D_CACHE_REFILL - L1 Data cache refill, which corresponds to the number of read and write misses that occur in the L1 Data cache. In this case, the PMC counts each access to L1 cache causing a refill of a cache line brought from the upper level (either main memory or another cache level).

4. L1I_CACHE_REFILL - L1 Instruction cache refill, which corresponds to the number of read misses that occur in the L1 Instruction cache. Note that the L1 I-cache is read-only, because the instructions are in general only read from the main memory.

5. CPU_CYCLES - This counter counts the number of processor cycles. To compute the execution time, we need to use the operating frequency of the processor since it may operate at a different frequency than other components (e.g. main memory). Combining this counter with the previous ones, it is possible to figure out the frequency of the occurrence of a specific event (e.g. number of L1 Data cache misses per cycle).

4.2.2 Architecture-specific events

Since the cluster Cortex-A53 architecture differs from the Cortex-R5 one, some events are specific of core type. For instance, the Cortex-A53 cluster has an L2 cache, whereas the Cortex-R5 cluster has not.

Cortex-A53

1. APU_L2D_CACHE_REFILL - L2 Data cache refill, which counts the number of accesses to the L2 cache causing a refill of a L2 cache line, regardless of whether they also cause a refill of the L1 Instruction cache, the L1 Data cache or none of them. This means that events 4 and 3 may overlap with this event if there is a miss in L2.

2. APU_L1D_CACHE_WB - L1 Data cache Write-Back, which corresponds to the number of performed write-backs from L1 Data cache to higher memory levels like L2 cache or the main memory. In other words, it counts how many times a modified line in L1 Data cache is evicted.

3. APU_L2D_CACHE_WB - L2 Data cache Write-Back, which corresponds to the number of write-back of data performed from L2 Data cache to main memory.

4. APU_L2D_CACHE - L2 Data cache access counter considers all accesses to a cache line of the L2 Data cache caused by read and write operations. Therefore, it includes the number of refills of both L1 Instruction and Data caches and the number of write-backs of data performed from L1 Data cache. This means that this event is the sum of all the accesses performed to a cache line of the L2 cache performed by L1 caches, i.e. the aforementioned events 4, 3 and 2 are included in this event.

5. APU_MEM_ACCESS - The memory accesses counter considers all accesses to L1 Data cache, L2 Data cache and main memory caused by read and write operations. Differently from *APU_L2D_CACHE* counter, it does not include the number of:

- Refilling of any cache.
- Write-backs of data performed from any cache.
- Instruction memory accesses.

6. APU_EXT_MEM_REQUESTS - The external memory request counter is defined for the APU of Cortex-A53 and it increments for each memory access request to the main memory that is "external" to such unit, thus including L2 cache misses and L2 cache write-back operations.

Cortex-R5

1. RPU_DCACHE_WB - L1 Data cache Write-Back counter for the two processors in the RPU cluster. It increases when one write-back of data is performed from L1 Data cache to higher memory levels.

2. RPU_EXT_MEM_REQUESTS - This PMC is incremented for each access request from L1 Data cache to an external memory like the main memory. It is analogous to APU_EXT_MEM_REQUESTS of the Cortex-A53 one.

4.3 Tools

In order to send commands to the embedded system, to check relative performances and to collect results, specific tools have to be employed. For instance, it is important to check constantly the correct execution of microbenchmarks looking in a text window where code is running, solving easily problems that can arise in such context. In particular, two tools employed in this Master thesis are presented: the interface through which user can communicate with the target platform and the server provided by BSC research center, which enables to make a lot of operations like compilation of the devised codes, check of their correct operation in real-time and further more. Such tools are:

- A debugger interface directly connected to the Zynq UltraScale+ EG platform, which is called Xilinx System Debugger. It is exploited to perform operations like reading the registers of each processor core in real-time or resetting the target processor when errors occur.

- BSC server with the GNU Collection Compiler (GCC), which is a cross-compiler toolchain suitable and widely used for most embedded systems. It mainly compiles C and C++ programming languages and it has the ARM backend support. Thanks to extended `asm` commands, we move from assembly code to C labels, facilitating the whole syntax. Note that GNU command language is Bash (Bourne-again shell), which is useful also to read and execute commands from a file, named as shell script. Bash is the most used language in this context since it enables to make a lot of operations with very few code lines compared to other scripting tools.

Chapter 5

Implementation

This chapter explains in details how the microbenchmarks described theoretically in Section 4.1 are implemented in practice through the use of both C and assembly programming languages. Firstly, the general structure of the main function in common with all benchmarks is described in Section 5.1, while the descriptions and features of the microbenchmarks subject of this study are listed in Section 5.2.

5.1 Main function

The experiments were performed using the main function represented in Algorithm 2. Note that the overall execution time is spent in the *Array initialization* function (executed just once) and by the microbenchmark chosen to be run. The latter is nested in the `do/while` loop and the `status` variable defines how many times the specific microbenchmark has to be performed. Such variable is fixed to 0 value because in this way the microbenchmark is "stuck" in an infinite loop, i.e. it can be executed an infinite number of times, stressing a specific resource of the target processor sustainedly as desired, and accounting for an execution time arbitrarily larger than the initialization part. Since that for the experiments of this work it is needed to perform the same microbenchmark (theoretically) infinite times, there is no need to change the `status` variable of the code and the user can decide to stop the microbenchmark whenever an error is showing up or when the expected results are obtained. Thanks to this approach, the code is made more

flexible and easier to be used during the Master thesis work.
The function `Disable_Prefetch()` is made to disable the data prefetcher implemented in Cortex-A53 and Cortex-R5 cores, which changes their timing behavior in an unpredictable way. However, this problem arises for some specific experiments only and it is addressed in Section 5.2.3, including details of the previous described function.
Note that it was needed to define some C pre-processor directives in order to make the code more flexible. In fact, as shown in the main function, depending on whether the *macro-name* `Init` or `Same_Set` are defined or not, it is possible to initialize the array in different ways. Other pre-processor directives were used, which we decided to do not include in Algorithm 2 to keep the pseudocode easy to read.
The `mem_init` function initializes differently the array that is used by microbenchmarks depending on the type of experiment. In particular, when `B` is defined, the corresponding `array_size` is changed depending both on the number of replacements chosen by the user and on the stride from one cache way to the next one, i.e. `OWS`, that changes from one processor architecture to the other one.

5.2 Experiments

The metrics used in experiments performed to stress the cache hierarchy of the two different processor clusters are the following ones:

1. L1 Data cache and L2 cache read misses and hits accessing different sets.

2. L1 Data cache and L2 cache read misses and hits forcing the processor to access always the same set.

3. L1 Data cache and L2 cache store misses and hits.

In the following sections the microbenchmarks used for such experiments written in C and assembly programming languages are presented in details. In particular, Section 5.2.1 introduces read microbenchmarks, Section 5.2.2

Algorithm 2 Main function

```
{
  int status = 0;
  Disable_Prefetch();

  #if defined Init     // Array initialization function...
    int **array = mem_init(array_size, stride);
  #elif defined Same_Set      //...to access the same set.
    int replacements = X;     // Nº of replacements;
    int **array = mem_init(OWS * replacements, OWS);
  #endif

  do
  {
    status = Microbenchmark();  // Desired microbenchmark
  }
  while (status==0);

  return 0;
}
```

introduces write microbenchmarks, Section 5.2.3 describes how prefetch interference is avoided, and Section 5.2.4 describes how PMCs have been interfaced.

5.2.1 Cache read operations: Load instructions

The microbenchmark devised for cache read operations is used for both L1 and L2 Data cache. In fact, setting properly both the size of the whole array and the stride for each array element, it is possible to impose cache read hits or misses in L1 or L2 Data cache. Such stride corresponds to the distance between beginnings of successive array elements in memory and depending on its size we can distinguish two general cases: if the stride size is exactly the same of one cache line, the next array element is located in the next cache line (cache miss), while if it is smaller than one cache line, the next array element is located in the same cache line (cache hit). The efficiency achieved with one stride instead of another one depends on the target architecture, i.e. on caching, access patterns, etc. and it can be

attributed to the principle of locality [30]. Since updating the stride to access the following element requires at least a non-load operation, this could lead to a load frequency lower than the maximum possible. To address this concern, it is needed to implement the *pointer chasing* technique (Algorithm 3). Such pointer chasing is part of the initialization process and, hence, it needs to be performed before the actual microbenchmark code (e.g. Algorithms 4 and 5) is executed. Next, we introduce how pointer chasing works and how it allows read microbenchmarks to execute mostly only load operations.

Array initialization using pointer chasing

The pointer chasing approach aims at placing in each array element the memory address of the next element to be accessed, avoiding its computation during the execution of the microbenchmark. Memory addresses are computed before using assembly code, thus reducing the number of non-portable operations written in assembly code. Moreover, no algebraic computations are needed to access the next array element. This improves the reliability of the results obtained with the measurement-based methodology used in this work, speeding up the execution of the microbenchmark itself. As shown in the Array initialization code (Algorithm 3), the `mem_init` function receives two inputs: the size of the array (array_size) and the distance between two consecutive array elements (stride). Afterwards, their initial value is divided by the size that the pointer to an integer variable occupies in the memory of the processor, which changes from architecture to architecture. This step is needed since each array element occupies 8 bytes in memory but the actual size of the pointer may vary across architectures, being either 4 or 8 bytes. Hence, we must make sure that we access appropriate addresses and the specified number of times. If this step was not done, there would be the risk that, after a number of accesses, memory accesses could occur beyond the boundaries of the array, thus leading to potential memory violations and to undesired timing behavior for the microbenchmark.
Afterwards, we allocate the required amount of memory for the microbenchmark with the `malloc` function given by C programming language. Later, the real array initialization takes place, which is mainly achieved using a "for" loop with a simple conditional inside, in order to capture the case of the last

element of list. Therefore, the first array element stores the memory address of the next array element that is placed in `cnt+stride`, which corresponds to the stride updated on each iteration. The stride is chosen in order to not violate the word-alignment, meaning that it has to be a multiple of 4 bytes. Note that `cnt` variable increases in each iteration by an amount equal to the stride defined initially. This means that just some array elements are accessed by the microbenchmark, while the other ones remain unset. For instance, with a stride of 16-bytes, the first element of the array contains the address of the array element 16 bytes after, which in turn contains the address of the array element 16 further bytes away, and so on and so forth. Elements in-between those ones are neither set nor used.

Algorithm 3 Array initialization using pointer chasing

```
int **mem_init(unsigned long int array_size, int stride)
{
  array_size = array_size / sizeof(int*);
  stride = stride / sizeof(int*);
  int**array = (int**) malloc(sizeof(int*)*array_size);

  unsigned long int cnt;
  for(cnt=0; cnt < array_size; cnt+=stride)
  {
    if(cnt < array_size - stride)
    {
      array[cnt] = (int*) &array[cnt+stride];
      //Each array element points to the address of the next one
    }
    else
    {
      array[cnt] = (int*) array;
      //The last accessed array element points to the first one.
    }
  }

  return array;
}
```

The microbenchmark algorithm uses this initialized array in order to perform cache and memory read operations.

Microbenchmark based on Load instructions

The code aimed to perform memory read operations (Algorithm 4) is based on load instructions, which are defined in the processor architecture and they have to be written for this reason in assembly code to ensure that the compiler does not alter the access pattern. Note that this corresponds to the first experiment (1) performed for the two different clusters.
Firstly, two pointer variables are defined to reference to the same address of the first element of the initialized array previously described (Algorithm 3). Afterwards, the first load instruction (`LDR`) is performed, which fetches the content of the first array element from the main memory and puts it in a register of the processor.
Since the content of the first array element corresponds to the address of the next array element to be accessed by this microbenchmark, the second load instruction fetches the content of the element whose address has just been fetched from main memory, putting it in another register. The same reasoning holds for the other 126 load instructions, where data loaded from memory is always the next address to be accessed, thus not needing to compute any address during the execution of the microbenchmark.
After these 128 load instructions are executed, a compare instruction (`CMP`) checks if the last memory address stored in the register accessed by the last load instruction corresponds to the one of the first array element. Depending on whether all array elements were accessed or not, and whether all iterations have been exhausted, either we iterate in the loop performing further load instructions or the loop finishes. Such control is performed with the `BNE` and `SUBS` instructions at the end of the loop.
Note that this microbenchmark is used for causing either all cache hits or all cache misses. In fact, such events occur depending on the size of the whole array to be stored in the target data cache and the particular stride used. Therefore, if the whole array fits completely in the target data cache, cache hits occur, otherwise cache misses are experienced as long as the stride is equal or larger than cache line size (under Least Recently Used, LRU, replacement policy). Other cases are not relevant for contention evaluation since we look for pure-hit or pure-miss cases in each cache memory, so mixed behavior is not interesting.

For instance, with this approach we can generate a microbenchmark hitting in L1 (array size not exceeding L1), missing in L1 and hitting in L2 (array size larger than L1 but not exceeding L2 size), and missing in both L1 and L2 (array size larger than L2). In all cases, we use a stride matching a cache line size to ensure that accesses occur in different cache lines so that cache line locality does not interfere with the experiment.

Algorithm 4 Microbenchmark based on Load instructions

```
int Loads(int** array, int iterations)
{
  int** q = array;
  int** r = array;
  //Array provided by the Pointer Chasing algorithm.

  //Start assembly code
  __asm__ __volatile__
  (
  ".data_cache_label_L1:" "\n\t"
  "LDR %0, [%1]"  "\n\t"
  "LDR %1, [%0]"  "\n\t"
                ...
              ...
  "LDR %0, [%1]"  "\n\t"
  "LDR %1, [%0]"  "\n\t"  //Total of 128 load instructions
  ""  "\n\t"
  "CMP %2, %1"  "\n\t"
  "BNE .data_cache_label_L1" "\n\t"
  "SUBS %3, %3, \#1"  "\n\t"
  "BNE .data_cache_label_L1"  "\n\t"
  ".label_L1_exit:"      "\n\t"
  :
  : "r"(q), "r"(array), "r"(r), "r"(iterations)
  );
  //End assembly code

  return 0;
}
```

Microbenchmark for accessing the same set

The microbenchmark methodology has been assessed in the past on caches implementing LRU replacement [6], but not specifically on caches using pseudo-random replacement. To assess the impact of using such a replacement policy of both the ARM Cortex-A53 and the ARM Cortex-R5 processors, we needed to perform a second experiment (2) and to devise another microbenchmark aiming to access the same set of cache lines in the L1 Data cache. For this reason, in the Array initialization code (Algorithm 3), the stride and the size have changed properly with respect to the ones employed in (Algorithm 4). In particular, the stride (`OWS`) is changed to match the size of one way of the L1 Data cache, which is computed as follows:

$$\mathrm{OWS} = \frac{\mathrm{DC\_size}}{\mathrm{W}} \tag{5.1}$$

where `DC_size` is the size of the data cache and `W` is the number of ways featuring the target cache. Note that the size of the cache and the size of the stride are expressed in the same units (i.e. either bytes or array elements). Therefore, the stride is set in such a way that the next array element accessed is at 1-way distance in the array, so that it is mapped exactly in the same L1 cache set. The corresponding code is shown in Algorithm 5.
It can be noted that such microbenchmark is very similar to the one based on load instructions (Algorithm 4). The main difference is the number of load instructions employed. In fact, the number of ways in which the two levels data caches (L1 and L2) are divided for both processor clusters is lower than the 128 load instructions used in the load instructions microbenchmark (Algorithm 4). Moreover, by using a variable number of load instructions, the microbenchmark is made more flexible from a programming point of view enabling a finer-grain control on the number of different cache lines competing for the space in a cache set. In fact, the size of the whole array is set to be:

$$\mathrm{DC\_size} = \mathrm{OWS} \cdot \mathrm{R} \tag{5.2}$$

where `R` is the number of different cache lines (number of replacements)

competing for the space in a cache set. Hence, depending on the number of replacements chosen, the size of the array changes. According to Equation 5.2, the number of replacements corresponds to the total number of unique addresses loaded in the loop. Therefore, the real number of replacements performed in the same set of L1 (or L2) Data cache has to be computed after all those addresses have been accessed for the first time so that a steady state is achieved.

Algorithm 5 Microbenchmark to access the same set

```
int LDR_L1waysN(int** BDA, const int iterations)
{
  int** p = BDA;
  int** q = BDA;

  //Start assembly code
  __asm__ __volatile__
  (
  ".l1waysnloop_begin:" "\n\t"
  "LDR %1, [%0]"    "\n\t"
  "LDR %0, [%1]"    "\n\t"
  ""   "\n\t"
  "CMP %2, %0"   "\n\t"
  "BNE .l1waysnloop_begin" "\n\t"
  "SUBS %3, %3, #1"          "\n\t"
  "BNE .l1waysnloop_begin" "\n\t"
  :
  : "r"(p), "r"(BDA), "r"(q), "r"(iterations)
  );
  //End assembly code

  return 0;
}
```

5.2.2 Cache write operations: Store instructions

For what concerns the cache and memory write operations, a microbenchmark different from the ones addressed in Section 5.2.1 has to be developed. In fact, it is not needed anymore to create an initialized array variable, meaning that the *pointer chasing* technique (Algorithm 3) is not employed for the third topology of experiments (3) shown in Section 5.2. The reason is that write operations do not fetch data from the memory addresses computed with the *pointer chasing* technique. In fact, contents are sent to memory by store instructions and each address to be accessed need to be computed. This implies using non-store instructions to set the address to be accessed by the following store instructions.
In the following paragraphs, microbenchmarks that stress data write features in the L1 and L2 Data cache are presented.

Microbenchmark based on Store instructions - Hits

The algorithm addressed in this section (Algorithm 6) employs store instructions to perform memory write operations and its goal is to cause hits in the target cache of the Cortex A53 processor cluster. Such instructions, as for the load ones, are pre-defined in the processor architecture and assembly language is needed to use them, avoiding the compiler to interfere with the desired microbenchmark behavior.
Firstly, memory allocation is performed with the `malloc` function so that stores can be performed on this memory structure. The address of the allocated memory is stored in the `st_array` integer pointer variable within which the data will be stored. Afterwards, the `st_pointer` integer pointer variable is used to access the allocated memory structure, so it is defined to point to the memory address previously assigned to the `st_array` first. Then, the `useless_data` (DEADBEEF) is the data to be stored in the allocated memory region. Note that this data corresponds to a size of 32-bit, namely one word size. This is in line with the fact that the stride between each store instruction has to be a multiple of 4 bytes to avoid unaligned accesses.
The main core of this microbenchmark consists of a for cycle including a total of 32 store instructions. Therefore, in each iteration, 32 memory write operations are carried out, meaning that the word “DEADBEEF” is written

32 times. Depending on the `iterations` variable value, it performs a higher or lower number of store instructions and such value must be chosen so that the amount of data accessed (and how many times it is accessed) produces the desired behavior, which in this case is a sequence of store hits (except for the first access to each cache line).
The microbenchmark based on store hits represented in this section uses a stride of 64 bytes between each element, meaning that the `useless_data` is stored to memory addresses with 64 bytes of distance each other. Such stride is used for Cortex-A53 processor and it is chosen in such a way that each store instruction accesses the next cache line (64-byte size for Cortex-A53 cores).
Each store instruction has the same memory address as the initial argument between square brackets plus an offset (stride) of 64 additional bytes with respect to the previous store. In order to allow that consecutive instructions are stores without needing instructions to update the stride, strides need to be written manually directly in the assembly code.
When the 32 store instructions are performed, an `ADD` instruction is used to update the total stride for the next iteration. In particular, such stride is 2048 bytes, thus equal to the 64 bytes per store multiplied by 32 store operations. In this way, in the next iteration, the store instructions are performed on the next available memory address of the `st_array` variable.
Before the next iteration starts, it is checked whether the `st_pointer` variable references to a valid range of memory addresses, namely to the ones assigned to the array `st_array` by not exceeding the array size. Such array is sized to ensure that store instructions cause cache hits. In fact, in this work, 24 kB was defined as upper limit since the L1 cache size is equal to 32 kB. Therefore, after the first 12 iterations, which correspond to 24 kB, cache hits are always experienced due to the fact all the memory addresses are stored in the L1 Data cache. Note that we could use up to 32 kB of data, but since some temporal variables are stored in cache, this would create some conflicts and consequently, some misses.
If the `st_pointer` references a memory location over the array bounds, it is set to the initial `st_array` memory address.
At the end of the microbenchmark, the function `free` is employed to clear the memory allocated to the `st_array` variable.

The description of Algorithm 6 also holds for the one used for the Cortex-R5 processor cores (Algorithm 7). It is possible to observe that the main differences between the two microbenchmarks are:

- Stride value, which is of 32 bytes in Cortex R5 since cache lines in those cores are 32-byte instead of 64-byte long.
- The total number of instructions per iteration set to 64 instead of 32 so that the amount of data accessed per iteration remains constant, although this constraint is not needed in practice and strides could be adapted accordingly.

Algorithm 6 Microbenchmark for store hits - Cortex A53

```
int store_hit(unsigned long int size, const int iterations)
{
  size = size / sizeof(int);
  int *st_array = (int*) malloc (size*sizeof(int));
  int *st_pointer = st_array;
  int useless_data = 0xdeadbeef;

  int j;
  for (j = 0; j < iterations; j++)
  {

    //Start assembly code
    __asm__ __volatile__
    (
    "STR %1, [%2, #64]" "\n\t"
    "STR %1, [%2, #128]"  "\n\t"
            ...
            ...
    "STR %1, [%2, #2048]" "\n\t"
    //Total of 32 stores instructions
    "ADD %0, %2, #2048"   "\n\t"
    : "=r"(st_pointer)
    : "r"(useless_data), "r"(st_pointer)
    );
    //End assembly code

    if ((int)st_pointer+(32*64) > (int)st_array + (24*1024))
      st_pointer = st_array;
  }

  free(st_array);
  return 0;
}
```

Microbenchmark based on Store instructions - Misses

The microbenchmark used to cause cache misses in Cortex A53 processor is represented below (Algorithm 8) and it is very similar to the one previously described (Algorithm 6), both in terms of concept and of implementation.

Algorithm 7 Microbenchmark for store hits - Cortex R5

```
int store_hit(unsigned long int size, const int iterations)
{
  size = size / sizeof(int);
  int *st_array = (int*) malloc (size*sizeof(int));
  int *st_pointer = st_array;
  int useless_data = 0xdeadbeef;

  int j;
  for (j = 0; j < iterations; j++)
  {

    //Start assembly code
    __asm__ __volatile__
    (
    "STR %1, [%2, #32]" "\n\t"
    "STR %1, [%2, #64]" "\n\t"
          ...
          ...
    "STR %1, [%2, #2048]" "\n\t"
    //Total of 64 stores instructions
    "ADD %0, %2, #2048"    "\n\t"
    : "=r"(st_pointer)
    : "r"(useless_data), "r"(st_pointer)
    );
    //End assembly code

    if ((int)st_pointer+(32*64) > (int)st_array + (24*1024))
      st_pointer = st_array;
  }

  free(st_array);
  return 0;
}
```

The main difference is the upper limit of the data array traversed. In fact, if accesses must miss in L1, but not in L2, then the size of the array traversed should be larger than L1 (32 kB) but smaller than L2 (1 MB). For instance, we could use an array of 64 kB. Conversely, if accesses must miss both in L1 and L2, we have to change the upper limit of the data array traversed to `DCL2_SIZE*2` (so twice the size of the L2 cache). Therefore, each store instruction has to cause a cache miss, since each cache line is written once (at the beginning of every cache line) in the target data cache and, since the number of lines accessed is larger than the number of cache lines available (in each cache set), those cache lines cannot fit in the L1 Data cache (and L2 cache).
An analogous microbenchmark is employed also for Cortex R5 processor (Algorithm 9) and the same observations made about Algorithm 6 are valid also for this one. As before, the stride value is 32 bytes instead of 64 bytes. Also note that the upper limit of the array traversed is, in this case, `DC_SIZE*2`, where `DC_SIZE` corresponds to 32 kB, namely the L1 Data cache size, since the Cortex-R5 processor cluster does not have L2 cache. Therefore, systematic cache misses are experienced since the amount of data accessed exceeds L1 cache space available.

Algorithm 8 Microbenchmark for store misses - Cortex A53

```
int store_miss(unsigned long int size, const int iterations)
{
  size = size / sizeof(int*);
  int *st_array = (int*) malloc (size*2*sizeof(int*));
  int *st_pointer = st_array;
  int useless_data = 0xdeadbeef;

  int j;
  for (j = 0; j < iterations; j++)
  {

    //Start assembly code
    __asm__ __volatile__
    (
    "STR %1, [%2, #64]" "\n\t"
         ...
         ...
    "STR %1, [%2, #128]"  "\n\t"
    "STR %1, [%2, #2048]" "\n\t"
    //Total of 32 stores instructions
    "ADD %0, %2, #2048"   "\n\t"
    : "=r"(st_pointer)
    : "r"(useless_data), "r"(st_pointer)
    );
    //End assembly code

    if ((int)st_pointer+(32*64) > (int)st_array + (DCL2_SIZE*2)
      st_pointer = st_array;
  }

  free(st_array);
  return 0;
}
```

5.2.3 Data prefetcher

The ARM v8 and v7 architectures have a Data prefetcher which changes considerably the behavior of the processor under study when the microbench-

Algorithm 9 Microbenchmark for store misses - Cortex R5

```
int store_miss(unsigned long int size, const int iterations)
{
  size = size / sizeof(int*);
  int *st_array = (int*) malloc (size*2*sizeof(int*));
  int *st_pointer = st_array;
  int useless_data = 0xdeadbeef;

  int j;
  for (j = 0; j < iterations; j++)
  {

    //Start assembly code
    __asm__ __volatile__
    (
    "STR %1, [%2, #32]" "\n\t"
    "STR %1, [%2, #64]" "\n\t"
          ...
          ...
    "STR %1, [%2, #2048]" "\n\t"
    //Total of 64 stores instructions
    "ADD %0, %2, #2048"    "\n\t
    : "=r"(st_pointer)
    : "r"(useless_data), "r"(st_pointer)
    );
    //End assembly code

    if ((int)st_pointer+(32*64) > (int)st_array + (DC_SIZE*2)
      st_pointer = st_array;
  }

  free(st_array);
  return 0;
}
```

marks are performed. In fact, it can anticipate the fetch of a cache line in the memory cache when a cache miss occurs. In general, it is hard to determine how many L1 and L2 cache and memory accesses are performed by the prefetcher, thus creating both arbitrary interference on other cores and making the task execution time unpredictable. Therefore, it is mandatory to

disable it during analysis to get realistic worst-case data. In order to disable such prefetcher in both processor cluster architectures, two functions were implemented: one to be employed for the Cortex A53 processor cores (Algorithm 10) and the other one to be used for the Cortex R5 processor cores (Algorithm 11), which are written in both C and assembly programming languages.
In Cortex A53, the prefetcher is disabled by resetting the relative bit in the Auxiliary Control Register. Instead, in Cortex R5, a couple of bits has to be set in the Auxiliary Control Register as specified by the manual [31].
Note that the Data prefetcher of both Cortex A53 and Cortex R5 are enabled again at the end of the PMC function that reads the PMCs described in Section (4.2), in order to restore the initial state of the platform. In the system during operation, prefetchers would be disabled in all cores before starting the execution of critical real-time tasks in any core. Then, whenever those tasks would finish, prefetchers could be set back to maximize average performance of non-critical software.

Algorithm 10 Disabling Data Prefetcher - A53

```
#define mask_DPreFetch 0xA000
//Bit mask for Disabling data prefetching

void Disable_Prefetch()
{
  uint64_t r = 0;
  uint64_t mask = ~mask_DPreFetch;

  __asm__ __volatile__
  (
  "MRS %0, S3_1_C15_C2_0" : "=r"(r)
  );
  // Read EL1 CPU Auxiliary Control Register

  r &= mask;

  __asm__ __volatile__
  (
  "MSR S3_1_C15_C2_0, %0" : :"r"(r)
  );
  // Write EL1 CPU Auxiliary Control Register
}
```

Algorithm 11 Disabling Data Prefetcher - R5

```
#define Mask_R5_Dis_prefetch 0x3000
//Bit mask for Disabling data prefetching

void Disable_Prefetch()
{
  unsigned int read = 0;

  __asm__ __volatile__
  (
  "MRC p15, 0, %0, c1, c0, 1" : "=r"(read)
  );
  // Read ACTLR

  read |= Mask_R5_Dis_prefetch;

  __asm__ __volatile__
  (
  "MCR p15, 0, %0, c1, c0, 1" : :"r"(read)
  );
  // Write ACTRL
}
```

5.2.4 Events counting: PMCs

To monitor cache hits or misses of the target cache, we used the Performance Monitoring Counters defined in the specific processor architecture.
The C function defined to perform events counting is shown in Algorithm 12, which has as input arguments the initialized array with Pointer Chasing technique (`p`) and the number of times that the microbenchmark has to be performed (`nruns`) that is chosen by the user.
Firstly, some arrays have to be defined in such function:

- In `PMCs[n_events]`each array element corresponds to one PMC and the array is initialized with the identifiers of the events to monitor.
- `PMCs_start[n_events]` contains the value of each PMC at the time of start counting the corresponding events defined in `PMCs[n_events]`.

- `PMCs_stop[n_events]`, instead, contains the value of each PMC after the execution of the microbenchmark for the corresponding events defined in `PMCs[n_events]`.

Note that `t_PMC_data` and `t_cnt_values` are defined in the overall code as respectively `unsigned int` and `uint64_t` types.
After these first steps, the prefetcher is disabled, and invalidation of the instruction cache and flush of the data cache are performed thanks to the `Xil_ICacheInvalidate` and `Xil_DCacheFlush` functions defined in the *Xilinx* environment. Then, the PMCs are enabled to start counting and their initial values are saved in `PMCs_start[]`, and immediately after the desired microbenchmark is executed. Upon the completion of the microbenchmark, the PMCs are disabled and their values retrieved to `PMCs_stop[]`. Then, the `for` loop iterates across the different PMCs monitored printing their identifiers and the number of events occurred during the microbenchmark execution, which is equal to the difference between `PMCs_stop[j]` and `PMCs_start[j]`. Finally, flush and invalidation of the relative caches are performed again and the Data prefetcher can be re-enabled.
It is important to note that data and instructions are processed continuously in the processor and they can affect the PMCs counting, meaning that the number of events counted can vary due to effects not strictly related to the execution of the microbenchmark. Therefore, the best solution to mitigate measurement noise as much as possible is to enable PMCs right before the microbenchmark loop (inside the microbenchmark) and disable them right after. For the sake of illustration, however, we wrote PMC enabling and disabling outside the microbenchmark, as this facilitates understanding the operation of the whole process. In practice, those operations occur inside the microbenchmarks themselves.

Algorithm 12 Performance Monitoring Counter function

```
int PMC(int** p, int nruns)
{
  int i, j, status;

  t_PMC_data PMCs[n_events] = {A,B,...};
  t_cnt_values PMCs_start[n_events];
  t_cnt_values PMCs_stop[n_events];
  uint32_t len = n_events;

  Disable_Prefetch();

  Xil_ICacheInvalidate();
  Xil_DCacheFlush();

  start_PMCs(PMCs_start[]);

  status = Microbenchmark(p, nruns);
  //Target microbenchmark to be monitored

  stop_PMCs(PMCs_stop[]);

  for (j=0; j<len; j++)
  {
    //Reading PMCs
    printf ("#PMCs: %x\n", PMCs[j]);
    printf (" %lld \t # low\n", PMCs_stop[j] - PMCs_start[j]);
  }

  Xil_ICacheInvalidate();
  Xil_DCacheFlush();

  Enable_Prefetch();

  return status;
}
```

Chapter 6

Results

In this chapter, the experimental results obtained with the microbenchmarks described in Section 5.2 are shown. The measurements are taken using all the PMCs described in 4.2 that are implemented in the target platform of this Master thesis.
Firstly, results for the execution on isolation are reported. Based on approximations of the maximum number of measurements obtained in each experiment and with the proper observations and interpretations of such results, we can understand how the main core of the two cluster works in isolation, highlighting worst-cases of memory latencies and comparing them with best ones. Afterwards, results with contenders are discussed, along with the research findings, where measurements are taken like in the experiments in isolation case, but in different scenarios, i.e. when all cores are running the same or different codes. In this case, the goal is to understand how the cores of both clusters work together in order to define guidelines on how the platform can experience real-time performance to be used in critical real-time embedded systems.

6.1 Experiments in Isolation

The first set of experiments focused on executing the benchmark tasks in isolation. Given that four Cortex-A53 cores are included in the APU of the Zynq platform and two Cortex-R5 cores are included instead in the

RPU, having the clusters different characteristics, we need to assess their performance and cache features separately. Therefore, in the following sections, experimental results for one core in each cluster is presented and discussed. For the sake of convenience, we refer to the core running the task under analysis as the "main core", although all cores in each cluster are identical.

6.1.1 Cortex-A53 laboratory results

In this section, the results obtained for the main core of the Cortex-A53 processor are analyzed in detail, since it is mandatory to assess the correctness of the microbenchmarks implemented, their performance on just one core and whether actual processor behavior matches specifications.

Level-1 data cache: accessing different sets

The first microbenchmark executed in this experimental evaluation is the one that accesses different sets of the target caches (Algorithm 4). Changing the array size while keeping the stride value constant, we can decide if only read cache hits are experienced or also read cache misses, as explained in previous chapter. In fact, choosing an array size larger than the one of L1 Data cache, both read cache hits and misses are experienced, otherwise just read cache hits.

L1 read cache hits. To obtain only L1 Data cache hits accessing different sets, it is needed to set an array size lower than the L1 Data cache size of 32 kB and a distance between each array element equal to one cache line size. In this way, each array element is stored in one different cache line of L1 Data cache and the accessing to an array element corresponds to the accessing one cache line. We decided to set a fixed array size of 24 kB and a stride of 64 bytes, getting the results shown in Table 6.1.
The number of the L1D_CACHE events, which are part of the *PMC events* discussed in Section 4.2, is higher than the one expected since that the number of accesses counted should be equal to the number of array elements accessed, i.e. 384 cache lines:

$$n\_array = \frac{Array\ size}{stride} = \frac{24 \cdot 1024\ Bytes}{64\ Bytes} = 384 \tag{6.1}$$

where `n_array` corresponds to the number of array elements, i.e. to the expected number of L1 Data cache accesses.
For this reason, the code was performed without any load instructions and it turned out that accesses to L1 Data cache occur due to other instructions needed for the implementation of the microbenchmark, which justifies this small discrepancy.
Looking at the APU_L2D_CACHE event, the expected results match quite well the ones obtained in laboratory since that no access to L2 cache is required and it is just the sum of two PMCs already addressed in Section 4.2, namely L1D_CACHE_REFILL and L1I_CACHE_REFILL.
In fact, the first one is equal to 384, which is expected from (6.1). Note that L1I_CACHE_REFILL is always equal to 11 for all the iterations performed due to the instructions needed for the implementation of the microbenchmark. Moreover, in this case, the APU_L2D_CACHE_REFILL event matches the previous one named APU_L2D_CACHE. It is expected since no dirty data is evicted from any cache memory, i.e. PMCs of write-backs (APU_L1D_CACHE_WB and APU_L2D_CACHE_WB) do not increment. From the description of PMCs behavior presented in Section 4.2, this means that APU_L2D_CACHE_REFILL event increments exactly as the APU_L2D_CACHE one.
For what concerns the remaining events named APU_EXT_MEM_REQUESTS and APU_MEM_ACCESS, they also match the expected results. In fact, the first one gives a result that is the sum of the L1D_CACHE_REFILL and L1I_CACHE_REFILL events, which is expected since that external memory requests cause cache line refills of both L1 Data and Instruction caches. The second one, instead, counts not only the cache line refills of the first level of cache, but also other instructions that are needed for the implementation and execution of the microbenchmark employed.
When more iterations are performed of the same microbenchmark, measurements are taken always making PMCs start to count when the first iteration

starts and making them stop to count after all iterations are over. Based on this observation, it is possible to note that the results obtained for some PMCs in the case of more iterations are almost equal to the ones obtained with just one iteration. The L1D_CACHE_REFILL counter, for instance, is expected to not increment after the first iteration, while the L1D_CACHE one increments since that the array entirely fit L1 Data cache, no read cache misses occur after the first iteration and accesses to L1 Data cache are performed experiencing read cache hits for all next iterations. For such counters, which are related to data cache accesses and cache line refills of the two cache levels implemented in Cortex-A53, it is expected that the number of events counted does not change. In fact, the size of the initialized array completely fits in the L1 Data cache and no other cache line refills or external memory requests are performed.
Counting of memory and L1 Data cache accesses, instead, increases as expected because the higher number of iterations, the higher the number of accesses to the first level of cache.
In terms of CPU_CYCLES, we observe that they are around 50,000 for just one iteration, thus more than 100 cycles per memory instruction (L1D_CACHE). This is expected because with just one iteration all accesses are cache misses. When increasing iterations to 10, then execution time is around 60,000 cycles, i.e. around 50,000 in the first iteration accounting for the first 384 cache misses and around 10,000 in the next 9 iterations. As expected, very low number of cycles is needed for each memory instruction after the first iteration since that in the next 9 iterations just cache hits are experienced. Thanks to these observations, we can conclude that around 3 cycles per memory instruction are performed, which is much lower than the previous 100 ones:

$$C\_hit = \frac{C_{10\ iterations} - C_{1\ iterations}}{D_{10\ iterations} - D_{1\ iteration}} = \frac{60703 - 50728}{3853 - 397} = \frac{9975}{3456} \approx 3 \qquad (6.2)$$

where `C_hit` corresponds to the number of cycles per read cache hit, `C` to the laboratory results of the counter CPU_CYCLES and `D` to the laboratory results of the counter L1D_CACHE.
From (6.2) we figure out that the denominator corresponds to the number of

L1 Data cache accesses in the next 9 iterations, i.e. around 3,500 cache hits (3,500 additional L1D_CACHE accesses), while the numerator represents the number of cycles experienced during the next 9 iterations, i.e. around 10,000. As we increase the number of iterations, the same (6.2) holds changing just the numbers of `C` and `D` of 10 iterations with the other ones, i.e. 100 and 1000, and the number of cycles per read cache hit is still approximately equal to 3.
Overall, we can conclude that, as expected, this microbenchmark causes all-misses during the first iteration and all-hits during the following ones. The number of cycles per hit is much lower than the one per miss as expected. Such code works properly and the platform did not show up any anomalies, i.e. any defects.

L1 read cache misses. For what concerns L1 read cache misses, we had to employ an array size larger than the size of L1 Data cache. In this way, not all the array elements can fit the L1 Data cache and the ones that do not fit replace the ones already present in such cache. Since that L1 Data cache of Cortex-A53 is equal to 32 kB, an array size of 40 kB is a reasonable choice. Therefore, selecting this array size with 64 bytes of stride, results in Table 6.2 are obtained.
In the first iteration, similar results are obtained for events like L1D_CACHE and APU_MEM_ACCESS compared with the ones represented in Table 6.1. The only relevant difference is that, since a larger array is traversed (40 kB instead of 24 kB), the number of accesses, and so the values of these event counters, increase proportionally.
It is also noted that the APU_L1D_CACHE_WB counter is not zero. This is an unexpected result since array data is only read and never modified, so cache lines evicted from the L1 Data cache are clean. Thus, we would expect this counter to be zero, but it is not. In fact, it corresponds exactly to the number of array elements that do not fit the L1 Data cache. The total number of array elements is 640 using the expression 6.1, but the L1 Data cache is 32 kB, meaning that just 512 array elements can be stored without evictions. Therefore, making the difference between these two last numbers, 128 is the number of array elements that are evicted from L1 to higher memory levels with 1 iteration, and the value read for this counter is 134, so with only a

L1 (read) Hits - A53_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	397	384
	APU_L2D_CACHE	395	384+11
	L1D_CACHE_REFILL	384	384
	APU_L2D_CACHE_REFILL	395	384+11
	CPU_CYCLES	50728	-
	APU_MEM_ACCESS	408	>384
	APU_EXT_MEM_REQUESTS	394	384+11
10	L1D_CACHE	3853	3840
	APU_L2D_CACHE	394	384+10
	L1D_CACHE_REFILL	384	384
	APU_L2D_CACHE_REFILL	394	384+10
	CPU_CYCLES	60703	-
	APU_MEM_ACCESS	3864	>3840
	APU_EXT_MEM_REQUESTS	394	384+10
100	L1D_CACHE	38413	38400
	APU_L2D_CACHE	394	384+10
	L1D_CACHE_REFILL	384	384
	APU_L2D_CACHE_REFILL	394	384+10
	CPU_CYCLES	164439	-
	APU_MEM_ACCESS	38424	>38400
	APU_EXT_MEM_REQUESTS	394	384+10
1000	L1D_CACHE	384013	384000
	APU_L2D_CACHE	394	384+10
	L1D_CACHE_REFILL	384	384
	APU_L2D_CACHE_REFILL	394	384+10
	CPU_CYCLES	1202360	-
	APU_MEM_ACCESS	384024	>384000
	APU_EXT_MEM_REQUESTS	394	384+10

Table 6.1: L1 read cache hits accessing different sets - A53_0

negligible discrepancy due to other instructions in the microbenchmark. As we increase the number of iterations, it holds that APU_L1D_CACHE_WB is roughly equal to L1D_CACHE_REFILL minus 512. Overall, a relevant conclusion of this analysis is as follows: *APU_L1D_CACHE_WB does not count only dirty L1 Data evictions, but* ***all*** *evictions (dirty and clean), which does not match the specifications.*
Note that the counted number of L2 Data cache accesses is quite large and this is due to the fact that such counter includes both L1 data misses (L1D_CACHE_REFILL) and L1 Data cache write-backs (APU_L1D_CACHE_WB), as explained in Section 4.2. Moreover, the number of events counted for L1I_CACHE_REFILL PMC is always equal to 10 for any number of iterations selected.
Note that the number of L1 Data cache misses (L1D_CACHE_REFILL) does not match the number of L1 Data cache accesses (L1D_CACHE). In fact, the expected number of misses for this microbenchmark is shown in the following expression:

$$N^\circ\ misses_{total\ iterations} = N^\circ\ misses_{1\ iteration} \cdot N^\circ\ iterations \tag{6.3}$$

Such relation has been proven to hold in other processors where the cache replacement policy is LRU [6]. However, since the L1 Data cache replacement policy implemented is Pseudo-Random in our case, many cache hits are experienced. In our microbenchmark, by traversing an array of 40 kB, we place 5 different cache lines (namely A, B, C, D and E) in each 4-way set. Under LRU, we fetch A, B, C, D first, then E evicts A, A evicts B, B evicts C and so on and so forth so that each access evicts the cache line needed next, thus creating an all-misses pattern. In the case of pseudo-random replacement, once A, B, C, D are fetched, E may or may not evict A, so that some times it leads to a miss and some others to a hit. For instance, if E evicts C, A hits, B hits and C misses, thus leading to another eviction, which may or may not evict D. Overall, a relevant fraction of hits is expected, which matches with the results obtained. In particular, as we increase the number of iterations, the miss rate decreases, reaching 40% for the highest number of iterations instead of the 100% desired. This is easily deducted by experimental results

obtained from L1D_CACHE and L1D_CACHE_REFILL counters, computing the ratio between the number of L1 Data cache accesses performed and the number of read cache misses experienced:

$$M\_R = \frac{A}{M} \cdot 100 \tag{6.4}$$

where `M_R` corresponds to miss ratio expressed in percentage, `A` to the L1 Data cache accesses performed and `M` to the read cache misses experienced.
Therefore, a different experiment is needed to cause an all-misses behavior, which allows us to maximize contention in the desired shared hardware resources.

Level-1 data cache: accessing the same set

The microbenchmark devised to cause all-misses in the L1 Data cache with the implemented Pseudo-Random replacement policy is the one aimed to force many accesses to occur on the same set of cache lines (Algorithm 5). Such algorithm forces all cache lines accessed to be placed in the same set so that few different cache lines are enough to exceed the space in a cache set. While we cannot enforce all accesses to be misses due to the Pseudo-Random replacement policy, where the probability of survival of a cache line is never zero, i.e. miss rate of 100%, we can approach asymptotically such case by increasing the number of cache lines largely above the cache set space.
Therefore, employing the One-Way stride value (5.1) to ensure to access the same cache set and adjusting the array size with respect to the number of replacements to be performed as presented in Algorithm 2, the results in Table (6.3) were obtained. In such experiment, 1000 iterations are performed of the same microbenchmark for all "replacements" values, which is the number of cache lines fetched in excess of the cache set space. For instance, 4 replacements means that we access 8 different cache lines in the same cache set for a 4-way cache, thus making sure that at least 4 replacements occur in such cache set.
As shown in the Table (6.3), the laboratory results of L1D_CACHE match always the expected ones since that each load instruction has to cause one access to the L1 Data cache. Note that they are slightly bigger than ex-

L1 (read) Misses - A53_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	653	640
	APU_L2D_CACHE	784	640+10+134
	L1D_CACHE_REFILL	640	640
	APU_L2D_CACHE_REFILL	650	640+10
	CPU_CYCLES	83902	-
	APU_MEM_ACCESS	664	>640
	APU_L1D_CACHE_WB	134	134
10	L1D_CACHE	6413	6400
	APU_L2D_CACHE	5440	>2968+10+2461
	L1D_CACHE_REFILL	2968	6400
	APU_L2D_CACHE_REFILL	650	640+10
	CPU_CYCLES	130327	-
	APU_MEM_ACCESS	6424	>6400
	APU_L1D_CACHE_WB	2461	5894
100	L1D_CACHE	64013	64000
	APU_L2D_CACHE	51471	>25984+10+25477
	L1D_CACHE_REFILL	25984	64000
	APU_L2D_CACHE_REFILL	651	640+10
	CPU_CYCLES	602222	-
	APU_MEM_ACCESS	64024	>64000
	APU_L1D_CACHE_WB	25477	63494
1000	L1D_CACHE	640013	640000
	APU_L2D_CACHE	512463	>256480+10+255973
	L1D_CACHE_REFILL	256480	640000
	APU_L2D_CACHE_REFILL	650	640+10
	CPU_CYCLES	5335666	-
	APU_MEM_ACCESS	640024	>640000
	APU_L1D_CACHE_WB	255973	639494

Table 6.2: L1 read cache misses accessing different sets - A53_0

pected and the reason is the same addressed in the previous experiments where different cache sets are accessed. Indeed, other L1 Data cache accesses are performed before the load instructions take place, precisely 14 accesses, and they are instructions needed for the implementation of this specific microbenchmark.
From the laboratory results of the L1D_CACHE_REFILL counter, instead, it is possible to figure out that the cache miss rate increases with higher number of replacements performed in the same set of cache lines. In fact, increasing the number of replacements, the probability of cache misses increases as well while the one of cache hits decreases (2.1). In the most extreme case, with 116 replacements the miss rate is around 99%, since 120 different cache lines contend for 4 physical cache lines in the set. The measured read cache misses are always lower than the expected values computed with (6.3), which corresponds to the case of 100% miss rate due to LRU cache policy, and they asymptotically approach the ideal value obtained with (6.3).

L1 Misses: accessing same set - A53_0			
Replacements	PMC events	Lab. results	Expected results
4	L1D_CACHE	8014	8000+14
	L1D_CACHE_REFILL	6122	<8000
	CPU_CYCLES	222453	-
12	L1D_CACHE	16014	16000+14
	L1D_CACHE_REFILL	14501	<16000
	CPU_CYCLES	538315	-
20	L1D_CACHE	24014	24000+14
	L1D_CACHE_REFILL	22502	<24000
	CPU_CYCLES	2172028	-
28	L1D_CACHE	32014	32000+14
	L1D_CACHE_REFILL	30545	<32000
	CPU_CYCLES	4567597	-
36	L1D_CACHE	40014	40000+14
	L1D_CACHE_REFILL	38546	<40000
	CPU_CYCLES	6560729	-
76	L1D_CACHE	80014	80000+14
	L1D_CACHE_REFILL	78547	<80000
	CPU_CYCLES	13714240	-
116	L1D_CACHE	120014	120000+14
	L1D_CACHE_REFILL	118546	<120000
	CPU_CYCLES	20732514	-

Table 6.3: L1 read cache misses accessing the same set - A53_0

Level-1 data cache: store instructions

Another type of microbenchmark was devised to perform writes to the L1 data cache as described in detail in Section (5.2.2), i.e. the one based on store instructions. Note that two microbenchmarks are employed for Cortex-A53 core: one ensuring that writing operations occur only in the cache lines of L1 Data cache, causing only write cache hits, and the other one performing writing operations that exceed L1 Data cache size, causing write cache misses.

L1 write cache hits. Using the microbenchmark to perform memory write operations aimed to cause cache hits (Algorithm 6), the results shown in Table (6.4) are obtained. Note that the stride value between each memory addresses is set to be equal to one cache line size (64 bytes) so that one store instruction corresponds to write data to one cache line, i.e. one cache line access. The total number of store instructions to be performed in each iteration is 32 and it is defined arbitrarily, ensuring to do not overcome the L1 Data cache size and to do not use many store instructions that may increase the code size.
Firstly, with just one iteration, the L1D_CACHE counter generates a number that is higher than the expected value of 32. In fact, it was experimentally tested that there are 20 accesses to L1 Data cache without considering the 32 store instructions of the microbenchmark implemented, being 6 of them within the main loop. Then, it is possible to conclude that the laboratory results match the expected ones.
For what concerns the APU_L2D_CACHE counter, it can be observed that it includes also the sum of the L1I_CACHE_REFILL and L1D_CACHE_REFILL counters represented in the same table. This is expected since all data fit L1 Data cache and no write backs are experienced, i.e. no counters of write-backs increment, meaning that the counter under observation works as explained in 4.2. Note that the L1I_CACHE_REFILL counter remains constant to 7 for all iterations and it corresponds only to the operations needed to implement the microbenchmark.
Similar observations can be made regarding the APU_L2D_CACHE_REFILL and the APU_EXT_MEM_REQUESTS counters. The first one is behaving exactly as the APU_L2D_CACHE counter since no write backs are experienced, but it shows a small discrepancy with respect to the other counter. Such error is quite acceptable and expected due to noise. The APU_EXT_MEM_REQUESTS counter matches almost the expected number of accesses performed to the second level cache and this corresponds to access a memory that is external to L1 Data cache 4.2. The reasons for the error shown by this PMC are equal to the ones explained for the APU_L2D_CACHE_REFILL counter.
Note that the number of cache misses (L1D_CACHE_REFILL) is 32 for one iteration, thus the 32 misses due to the 32 store instructions, which fetch 2

kB of data. When we increase iterations to 10, there are 320 misses (20 kB of data). For 100 iterations we have 383 misses (≈24 kB of data). In this latter case, we fetch the whole array several times, since the whole array fits L1 data cache and no cache lines have to evicted, so we experience the maximum number of misses possible. Finally, for 1,000 iterations the number of misses remains constant (383), so the effect on execution time, compared to the case with 100 iterations, is the increment caused by the store hits in the L1 Data cache. Therefore, CPU_CYCLE counter measured 71671 cycles, i.e. there is an increase in execution time of around 55,000 cycles with respect to the case of 100 iterations (16828 cycles) caused by the additional 900 iterations with 32 store operations each. From these observations, we can compute approximately the number of cycles per store hit operation:

$$C\_st\_hit = \frac{C_{1000\ iterations} - C_{100\ iterations}}{D_{1000\ iterations} - D_{100\ iteration}} \approx \frac{55000}{34500} \approx 2 \qquad (6.5)$$

where `C_st_hit` corresponds to the number of cycles per store hit operation, `C` to the laboratory results of the counter CPU_CYCLES and `D` to the laboratory results of the counter L1D_CACHE.
Note also that, by having 6 "unwanted" memory accesses within the loop, the L1D_CACHE counter matches approximately 32+6 accesses per iteration. Those 6 accesses correspond to the benchmark code that checks the condition to update the array pointer and few other variable accesses. They could likely become register accesses with some compiler optimizations. However, compilation was performed without optimizations to prevent the compiler from altering the assembly code placed to create specific cache behavior.

L1 write cache misses. To cause L1 cache misses on memory write operations, we need to use the microbenchmark performing store instructions that exceed L1 Data cache size (Algorithm 8). The stride value is set to 64 bytes, namely one cache line size, exactly for the same reasons explained in the L1 write cache hits experiment (6.1.1) and using a total of 32 store instructions for just one iteration, the results shown in Table (6.5) are obtained.
Considering the case when the microbenchmark is executed just once (iteration), the laboratory results of the L1D_CACHE counter are analogous to

L1 (write) Hits - A53_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	52	32+6+14
	APU_L2D_CACHE	38	32+6
	L1D_CACHE_REFILL	32	32
	L1I_CACHE_REFILL	7	∼7
	APU_L2D_CACHE_REFILL	37	∼38
	CPU_CYCLES	1172	-
	APU_EXT_MEM_REQUESTS	38	∼37
10	L1D_CACHE	394	320+60+14
	APU_L2D_CACHE	390	320+6
	L1I_CACHE_REFILL	7	∼7
	L1D_CACHE_REFILL	320	320
	APU_L2D_CACHE_REFILL	326	∼326
	CPU_CYCLES	9672	-
	APU_EXT_MEM_REQUESTS	326	∼326
100	L1D_CACHE	3830	3200+600+14
	APU_L2D_CACHE	390	383+7
	L1I_CACHE_REFILL	7	∼7
	L1D_CACHE_REFILL	383	384
	APU_L2D_CACHE_REFILL	388	∼390
	CPU_CYCLES	16828	-
	APU_EXT_MEM_REQUESTS	326	∼326
1000	L1D_CACHE	38180	32000+6000+14
	APU_L2D_CACHE	390	383+7
	L1I_CACHE_REFILL	7	∼7
	L1D_CACHE_REFILL	383	384
	APU_L2D_CACHE_REFILL	388	∼390
	CPU_CYCLES	71671	-
	APU_EXT_MEM_REQUESTS	389	∼390

Table 6.4: L1 write cache hits - A53_0

those for the L1 write cache hits microbenchmark, since both microbenchmarks cause all-misses behavior for just one iteration. Similar conclusions can be reached for 10 iterations since both microbenchmarks still cause all-misses behavior.

When increasing iterations to 100, instead, the number of misses in the L1 Data cache (L1D_CACHE_REFILL counter) increases to 3,200, in line with the 32 store misses per iteration of the microbenchmark since now store instructions are writing data in more than L1 Data cache size. We also note that the number of L1 Data cache accesses (L1D_CACHE counter) includes 6 hits per iteration, as in the all-hits case, and the reason is still the same of that case.

Regarding the APU_L2D_CACHE counter, we observe that it includes two types of accesses: L1 cache misses (mostly L1D_CACHE_REFILL counter) and L1 Data cache write-backs (APU_L1D_CACHE_WB counter), being the latter indicated as `wb` in the table. The former corresponds to data requested from L1 caches, whereas the second corresponds to data evicted from the L1 Data cache. In the case of 1 and 10 iterations, the amount of data accessed is 2 kB and 20 kB respectively, thus not enough to fill the cache (32 kB) and to cause any L1 Data cache eviction. However, after 16 iterations the L1 Data cache gets full, as shown with the expression below:

$$Data\_size = N_{iters} \cdot str_{iter} \cdot S = 16 \cdot 32 \cdot 64 = 32kB \tag{6.6}$$

where `N_iters` corresponds to the number of iterations, `str_iter` to the number of stores per iteration and `S` to the stride value.

Therefore, except those first 512 stores (32 stores per iteration during 16 iterations), all remaining stores cause an L1 Data cache eviction, and so an additional L2 cache access. Hence, in the case of 100 iterations we expect 3,200 L2 accesses due to store misses and 3,200-512=2,688 L2 accesses due to L1 Data cache line eviction. Thus, the total number of observed L2 accesses (5,910) matches quite well the expectations (3,200+2,688=5,888).

Results for 1,000 iterations follow the same trends with 32,000 L1 Data cache misses, getting around 31,500 L1 Data cache write-backs and 63,500 L2 cache accesses.

A somewhat unexpected result corresponds to the APU_L2D_CACHE_WB counter (not shown in the table), which is always 0. The L2 cache has 1 MB capacity. Hence, when using the microbenchmark with 1, 10 and 100 iterations we are unable to fill it and therefore, no evictions occur. However, when performing 1,000 iterations, 2 MB of data are fetched, i.e. twice L2 data cache size, thus meaning that the latest half of the 32,000 store accesses should produce L2 evictions. If this was the case, APU_L2D_CACHE_WB would be 16,000 and APU_EXT_MEM_REQUESTS would be 48,000 (32,000 misses + 16,000 evictions). However, L2 evictions are neither counted by the L2 evictions counter nor by the memory access counter. This suggests that either those events are not monitored correctly or their configuration (as described in the specifications) is not properly described. This has been double-checked and no error was found in the code that reads the PMCs, which has been implemented as indicated in the specification. Also, the fact that this issue affects two different event counters provides some indications that the event may not be properly monitored by the hardware, thus not counting it as expected. Overall, another relevant conclusion is that *APU_L2D_CACHE_WB and APU_EXT_MEM_REQUESTS event counters may not work properly.*

Finally, we want to note that the execution time (in cycles) of this microbenchmark reaches 1,220,000 cycles approximately for 32,000 store misses. This leads to the conclusion that the processor can perform a store operation in memory every 40 cycles on average:

$$C\_st\_miss \approx \frac{C_{1000\ iterations}}{M_{1000\ iterations}} \approx \frac{1220000}{32000} \approx 40 \tag{6.7}$$

where `C_st_miss` corresponds to the number of cycles per store miss operation, `C` to the laboratory results of the counter CPU_CYCLES and `M` to the laboratory results of the counter L1D_CACHE_REFILL.

This is in contrast with load latencies, which were largely above 100 cycles as addressed in Section (6.1.1), thus reflecting that stores can be processed asynchronously to some extent without stalling execution despite the fact that memory latency may be above 100 cycles.

L1 (write) Misses - A53_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	48	32+6+10
	APU_L2D_CACHE	38	32+6+wb
	L1D_CACHE_REFILL	32	32
	APU_L2D_CACHE_REFILL	37	31+6
	CPU_CYCLES	1191	-
	APU_EXT_MEM_REQUESTS	38	∼38
	APU_L1D_CACHE_WB	0	0
10	L1D_CACHE	390	320+60+10
	APU_L2D_CACHE	326	320+6+wb
	L1D_CACHE_REFILL	320	320
	APU_L2D_CACHE_REFILL	326	320+6
	CPU_CYCLES	9700	-
	APU_EXT_MEM_REQUESTS	326	∼326
	APU_L1D_CACHE_WB	0	0
100	L1D_CACHE	3810	3200+600+10
	APU_L2D_CACHE	5910	∼3200+6+wb
	L1D_CACHE_REFILL	3201	3200
	APU_L2D_CACHE_REFILL	3205	3200+6
	CPU_CYCLES	93470	-
	APU_EXT_MEM_REQUESTS	3207	∼3206
	APU_L1D_CACHE_WB	2703	∼2688
1000	L1D_CACHE	38010	32000+6000+10
	APU_L2D_CACHE	63508	∼32000+6+wb
	L1D_CACHE_REFILL	32000	32000
	APU_L2D_CACHE_REFILL	32005	∼32000
	CPU_CYCLES	1220134	-
	APU_EXT_MEM_REQUESTS	32006	∼32006
	APU_L1D_CACHE_WB	31503	∼31488

Table 6.5: L1 write cache misses - A53_0

6.1.2 Cortex-R5 laboratory results

This section presents the experimental results obtained for the main core of the RPU that includes the two Cortex-R5 cores.

Level-1 data cache: accessing different sets

Focusing on the memory read operations, firstly accesses to different sets were performed through the use of the same microbenchmark used for Cortex-A53 (Algorithm 4). Therefore, the same reasoning regarding the array size and the stride value for Cortex-A53 to get cache hits and misses holds also for the ARM v7 architecture, with the only difference that cache line size differs across Cortex A53 and R5 cores, as explained in Section 2.4.

L1 read cache hits. L1 read data cache hits are obtained using a fixed array size of 24 kB and stride of 32 bytes, which corresponds to one cache line size of Cortex R5 processor, and the results are shown in Table (6.6).
Considering the formula about computing the number of cache line refills (Equation 6.1), it can be highlighted that the results obtained with the L1D_CACHE_REFILL counter matches the expected ones, since that 768 is the number of load instructions performed by the specific microbenchmark and the number of the cache line refills experienced by L1 Data cache: 768 loads with stride 32, for a total of 24 kB. Therefore, it corresponds also to the number of cache line refills (misses) that L1 Data cache is experiencing. The only difference between the practical and the theoretical results of the L1D_CACHE counter is that there are other accesses to the L1 Data cache that are related just to the implementation of the microbenchmark.
Then, the number of external memory requests measured by RPU_EXT_MEM_REQUESTS counter matches the one of the cache line refills measured by L1D_CACHE_REFILL counter since, in the first iteration, the processor has to fetch all the array elements from the main memory, i.e. each external memory request corresponds to a cache miss.
Afterwards, as we increase the number of iterations, since that the size of the array completely fits in the L1 Data cache, the number of L1 Data cache line refills and external memory requests respectively measured by L1D_CACHE_REFILL and RPU_EXT_MEM_REQUESTS counters does

not change. As expected, instead, more accesses to the L1 Data cache are performed increasing the number of iterations since that many accesses are carried out to each array element stored in such cache. Therefore, the numbers obtained match always the expected results computed with Equation (6.1).

L1 (read) Hits R5_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	778	768+10
	L1D_CACHE_REFILL	768	768
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	43193	-
	RPU_EXT_MEM_REQUESTS	768	768
10	L1D_CACHE	7690	7680+10
	L1D_CACHE_REFILL	768	768
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	64318	-
	RPU_EXT_MEM_REQUESTS	768	768
100	L1D_CACHE	76810	76800+10
	L1D_CACHE_REFILL	768	768
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	275546	-
	RPU_EXT_MEM_REQUESTS	768	768
1000	L1D_CACHE	768010	768000+10
	L1D_CACHE_REFILL	768	768
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	2387851	-
	RPU_EXT_MEM_REQUESTS	768	768

Table 6.6: L1 read cache hits accessing different sets - R5_0

L1 read cache misses. The array size chosen to cause cache misses in L1 cache of the main core Cortex-R5 is 40 kB with stride value of 32 bytes, obtaining the results shown in Table (6.1.2).
When just one iteration is performed, the number of accesses to the L1 Data cache counted by L1D_CACHE is almost equal to the one predicted theor-

etically. In fact, the access to one array element corresponds to one access to L1 Data cache, as in the Cortex A53 experiments presented previously in this section, and 1,280 is the number of array elements measured as expected from Equation (6.1), which corresponds to the number of cache lines that L1 Data cache has to handle. The remaining 10 accesses correspond to other memory instructions in the microbenchmark excluding the loads placed on purpose in the loop.

When considering the L1 Data cache line refills and the external memory requests counted with 1 iteration by L1D_CACHE_REFILL and RPU_EXT_MEM_REQUESTS counters respectively, the results match the expected ones: around 1,280 L1 Data cache misses and memory accesses are needed to fetch 40 kB of data.

As we increase the number of iterations, we observe a trend similar to the Cortex-A53 case: a large number of load hits in the L1 Data cache due to the pseudo-random replacement policy, with the miss rate being around 40% that is easily computed considering (6.4). For instance, with 10 iterations we have around 12,800 L1 Data cache accesses (L1D_CACHE), out of which 5,380 approximately are L1 Data cache refills (L1D_CACHE_REFILL) and main memory accesses (RPU_EXT_MEM_REQUESTS).

Another relevant information is that the number of L1 Data cache write-backs is zero. This is in contrast with the experiment of L1 read cache misses in the case of the Cortex-A53 core (6.1.2), where the number of write-backs measured by APU_L1D_CACHE_WB counter was similar to the number of refills despite cache lines being clean. We regard the case of the Cortex-R5 core as the correct case, since it is the one matching the specifications.

L1 (read) Misses R5_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	1290	1280+10
	L1D_CACHE_REFILL	1281	1280
	L1I_CACHE_REFILL	17	∼18
	CPU_CYCLES	72057	-
	RPU_EXT_MEM_REQUESTS	1286	1280
	RPU_DCACHE_WB	3	0
10	L1D_CACHE	12810	12800+10
	L1D_CACHE_REFILL	5382	12800
	L1I_CACHE_REFILL	17	∼18
	CPU_CYCLES	324630	-
	RPU_EXT_MEM_REQUESTS	5388	12800
	RPU_DCACHE_WB	6	0
100	L1D_CACHE	128010	128000+10
	L1D_CACHE_REFILL	51462	128000
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	3118184	-
	RPU_EXT_MEM_REQUESTS	51721	128000
	RPU_DCACHE_WB	6	0
1000	L1D_CACHE	1280010	1280000+10
	L1D_CACHE_REFILL	512262	1280000
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	31055075	-
	RPU_EXT_MEM_REQUESTS	512534	1280000
	RPU_DCACHE_WB	5	0

Table 6.7: L1 read cache misses accessing different sets - R5_0

Level-1 data cache: accessing the same set

In the case of R5 processor, the same microbenchmark (Algorithm 5) used for the Cortex A53 core was employed in order to access the same set of cache lines in the L1 Data cache. Therefore, maintaining the same stride value of 8 kB between each replacement computed with (5.1), namely the distance of one way with respect to the next one, and changing the array size

depending on the number of replacements to be implemented as presented in Algorithm (2), it was possible to get the results shown in Table (6.1.2).
The laboratory results obtained for the L1D_CACHE counter match always the expected ones since each load instruction causes one access to the L1 data cache, with the minor difference caused by other (few) memory accesses excluding the load pattern tested, which are instructions needed for the implementation of this specific microbenchmark.
Regarding the counter of cache line refills (L1D_CACHE_REFILL), it is surprisingly always equal to the number of times that the array elements are read from the main memory. This is an unexpected behavior of the L1 Data cache of R5 since, given that it implements a pseudo-random replacement policy, we expect results analogous to those of the Cortex-A53 core, where an increasing number of replacements leads to an asymptotic miss rate approaching of 100%. However, as shown in these experiments, with 4 replacements (8 lines competing for 4 ways in the set), all accesses become misses. While this is counterintuitive, it is possible and compatible with the fact that our results for the read misses case reveal that the replacement policy is pseudo-random. In particular, in the previous experiment of L1 read cache misses when accessing different sets in Cortex R5 (6.1.2) we traversed a 40 kB array, thus placing 5 cache lines per set (thus with 1 replacement). Depending on how the pseudo-random replacement policy is implemented, it may lead to systematic scenarios where 1 replacement is random, but 4 replacements systematically evict all 4 lines in the set.
While the actual implementation is unknown, a recent work has proposed a replacement policy with exactly those characteristics [32]. In particular, such policy proposes to perform a random permutation of the cache ways to select the next cache line to be evicted. Hence, the first choice is random, the second choice is random among the non-evicted lines, and so on and so forth until the N^{th} replacement (where N stands for the number of cache ways) evicts systematically the only way not evicted so far, and a new random permutation is generated. Such a policy, therefore, evicts 1 random line with 1 replacement, and all 4 lines in the set with 4 replacements (in random order). In general, even if it is somewhat unexpected, it can happen that different behaviors for the same replacement policy across core types are experienced. Hence, under the assumptions that our measure-

ments are following the aforementioned general hypothesis, Cortex-A53 and R5 cores differ on the behavior of some PMCs and on the implementation of the pseudo-random replacement policy.

L1 Misses: accessing same set - R5_0			
Replacements	PMC events	Lab. results	Expected results
4	L1D_CACHE	8011	8000+11
	L1D_CACHE_REFILL	8000	<8000
	CPU_CYCLES	483899	-
12	L1D_CACHE	16011	16000+11
	L1D_CACHE_REFILL	16000	<16000
	CPU_CYCLES	983038	-
20	L1D_CACHE	24011	24000+11
	L1D_CACHE_REFILL	24000	<24000
	CPU_CYCLES	1680904	-
28	L1D_CACHE	32011	32000+11
	L1D_CACHE_REFILL	32000	<32000
	CPU_CYCLES	2336809	-
36	L1D_CACHE	40011	40000+11
	L1D_CACHE_REFILL	40000	<40000
	CPU_CYCLES	2921230	-
76	L1D_CACHE	80011	80000+11
	L1D_CACHE_REFILL	80000	<80000
	CPU_CYCLES	5842789	-
116	L1D_CACHE	120011	120000+11
	L1D_CACHE_REFILL	120000	<120000
	CPU_CYCLES	8764792	-

Table 6.8: L1 read cache misses accessing the same set - R5_0

Level-1 data cache: store instructions

Similar microbenchmarks to the ones that perform writes to the L1 Data cache of Cortex A53 are employed for Cortex R5 processor and some variables have to be properly changed since the target architecture is different from

the previous one. In particular, the stride value is 32 instead of 64 bytes and the number of stores in the loop is 64 instead of 32. Therefore, as in the Cortex A53 core case, two different experiments have to be distinguished: one that causes cache store hits and the other one that causes cache store misses in the L1 Data cache.

L1 write cache hits. The results shown in Table (6.9) were obtained employing the microbenchmark aimed to cause cache hits with store instructions (Algorithm 7). It is important to observe that, in this case, the stride value from each store instruction was set to 32 bytes and a total of 64 store instructions are performed in each iteration as explained before.
Considering the case with just one iteration, the L1D_CACHE counter increases more than expected and this is due to the fact that there are 20 accesses to L1 Data cache that are counted without considering the 64 store instructions of the microbenchmark implemented. Out of those 20 accesses, 6 of them are within the loop, as in the case of the Cortex-A53 cores. Then, we can conclude that the laboratory results match the expected ones.
The L1D_CACHE_REFILL counter measured a number of L1 Data cache misses almost equal to the expected result. Analogous conclusions can be reached for the number of memory accesses measured by RPU_EXT_MEM_REQUESTS counter. The fact those counters are slightly lower than expected is very likely because the microbenchmark finalizes its execution while some stores are still waiting to be written-back and thus not counted yet, since write-backs are typically processed asynchronously.
In any case, for one iteration the number of misses follows the expected result computed with Equation (6.1): 64 store misses to fetch 2 kB of data. For 10 iterations, the behavior is still all-misses since 20 kB of data are fetched and never accessed again. However, as for the Cortex-A53 core case, the full array (24 kB) is fetched after 12 iterations, thus with 768 L1 Data cache misses ($64 \cdot 12 = 768$). For this reason, the remaining iterations in the cases with 100 and 1,000 iterations perform all-hits accesses. The relation between number of accesses, the number of stores per iteration and the stride, and the data size fetched, is shown in the following equation:

$$Data\_size = N_{iters} \cdot str_{iter} \cdot S = 12 \cdot 64 \cdot 32 = 24kB \tag{6.8}$$

where `N_iters` corresponds to the number of iterations, `str_iter` to the number of stores per iteration and `S` to the stride value.
The first 12 iterations have an all-misses behavior and the remaining ones an all-hits behavior. Hence, by observing the increase in execution time from 100 to 1,000 iterations (258,000 cycles), we can conclude that one store hit is processed every 4 cycles on average:

$$C\_st\_hit\_R5 = \frac{C_{1000\ iterations} - C_{100\ iterations}}{D_{1000\ iterations} - D_{100\ iteration}} \approx \frac{258000}{63000} \approx 4 \tag{6.9}$$

where `C_st_hit` corresponds to the number of cycles per store hit operation, `C` to the laboratory results of the counter CPU_CYCLES and `D` to the laboratory results of the counter L1D_CACHE.
Note that, in the case of loads, the average hit latency was around 3 cycles for the R5 cores, i.e. load hit operations are faster than store ones.

L1 (write) Hits - R5_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	81	64+6+14
	L1I_CACHE_REFILL	9	∼10
	L1D_CACHE_REFILL	61	64
	CPU_CYCLES	3474	-
	RPU_EXT_MEM_REQUESTS	60	∼64
	RPU_DCACHE_WBACK	0	0
10	L1D_CACHE	711	640+60+14
	L1I_CACHE_REFILL	9	∼10
	L1D_CACHE_REFILL	637	640
	CPU_CYCLES	34846	-
	RPU_EXT_MEM_REQUESTS	636	∼640
	RPU_DCACHE_WBACK	0	0
100	L1D_CACHE	7019	6400+600+14
	L1I_CACHE_REFILL	9	∼10
	L1D_CACHE_REFILL	768	768
	CPU_CYCLES	66968	-
	RPU_EXT_MEM_REQUESTS	768	∼768
	RPU_DCACHE_WBACK	0	0
1000	L1D_CACHE	70094	64000+6000+14
	L1I_CACHE_REFILL	9	∼10
	L1D_CACHE_REFILL	768	768
	CPU_CYCLES	324959	-
	RPU_EXT_MEM_REQUESTS	768	∼768
	RPU_DCACHE_WBACK	0	0

Table 6.9: L1 write cache hits - R5_0

L1 write cache misses. Results for the all-misses store microbenchmark (Algortihm 9) are shown in Table (6.10).
Considering the case when just 1 or 10 iterations of the microbenchmark are executed, the L1D_CACHE counter shows the same results obtained in Table (6.9), which corresponds to the microbenchmark for store hits, since the first 12 iterations experience only misses in both cases.
In this case, it is not useful to write 2 MB data as for the L1 write cache misses experiment of Cortex A53 core (6.1.1) because there is no L2 cache in Cortex R5 processor cluster. For this reason, we write 64 kB data, i.e. twice the L1 Data cache size, in order to get the same behavior as the 4-replacements case when accessing the same set (6.1.2).
The results obtained are on the one hand consistent and on the other unexpected. They are consistent since that the number of write-back operations measured by RPU_DCACHE_WBACK counter plus the number of L1 Data cache refills measured by L1D_CACHE_REFILL counter match precisely the number of memory requests measured by RPU_EXT_MEM_REQUESTS counter, as expected from the description of PMC's behavior in 4.2. However, results are unexpected because the miss rate is not 100% as for the 4-replacements case when accessing a single set. In particular, miss rates for the stores are around 84% for 100 iterations (5,400 misses out of 6,400 stores), and around 75% for 1,000 iterations (48,000 misses out of 64,000 stores) according to Equation (6.4). This reveals that the behavior of the pseudo-random replacement policy may vary noticeably even when the number of cache lines fetched (and the pattern to access them) remains the same for any given set, as it is the case for our experiments accessing a single set or all of them. While such behavior is possible, it reveals that the pseudo-random replacement policy is far from being sufficiently random, thus leading to systematic behavior in specific scenarios. This plays against time predictability in general and hence, suggests that one cannot rely much on cache hits in the L1 Data cache of this processor unless all data fits so that an all-hits behavior is expected.
Finally, in terms of memory latency, the processor needs 56 cycles on average to perform one store operation per cache miss:

$$C\_st\_miss \approx \frac{C_{1000\ iterations}}{M_{1000\ iterations}} = \frac{2715226}{48489} \approx 56 \tag{6.10}$$

where `C_st_miss` corresponds to the number of cycles per store miss operation, `C` to the laboratory results of the counter CPU_CYCLES and `M` to the laboratory results of the counter L1D_CACHE_REFILL.
If we take into account that a number of store hits are served in between store misses, then this latency may be even lower. Furthermore, if we consider that each memory store operation carries out also another access due to the dirty eviction, we can take it into account multiplying by a factor 2 the `M` parameter, concluding that a memory request for one store operation per cache miss may be served every 28 cycles on average. This is in contrast with the case of the previous experiment of load misses of Cortex R5 (6.1.2) where, for instance, 120,000 misses are served in around 8,760,000 cycles, with an average of about 70 cycles per load miss. Hence, as in the Cortex-A53 case, stores can be processed asynchronously to some extent without stalling execution even if memory latency is around 70 cycles.

L1 (write) Misses - R5_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	81	64+6+14
	L1I_CACHE_REFILL	10	∼10
	L1D_CACHE_REFILL	61	64
	CPU_CYCLES	3572	-
	RPU_EXT_MEM_REQUESTS	60	∼64
	RPU_DCACHE_WBACK	0	0
10	L1D_CACHE	711	640+60+14
	L1I_CACHE_REFILL	10	∼10
	L1D_CACHE_REFILL	637	640
	CPU_CYCLES	34944	-
	RPU_EXT_MEM_REQUESTS	636	∼640
	RPU_DCACHE_WBACK	0	0
100	L1D_CACHE	7014	6400+600+14
	L1I_CACHE_REFILL	10	∼10
	L1D_CACHE_REFILL	5457	6400
	CPU_CYCLES	300378	-
	RPU_EXT_MEM_REQUESTS	9917	>6400
	RPU_DCACHE_WBACK	4451	∼5376
1000	L1D_CACHE	70044	64000+6000+14
	L1I_CACHE_REFILL	10	∼10
	L1D_CACHE_REFILL	48489	64000
	CPU_CYCLES	2715226	-
	RPU_EXT_MEM_REQUESTS	96893	>64000
	RPU_DCACHE_WBACK	47832	∼62976

Table 6.10: L1 write cache misses - R5_0

6.2 Experiments with contenders

The single-core tests on the Cortex A53 and Cortex R5 were performed step by step. From the results obtained with such tests, it was verified that each microbenchmark works as expected, forcing the target cache to perform specific operations, and some unexpected behaviors of the platform were revealed. Then, it is possible to perform several experiments with such microbenchmarks when all the cores in each cluster of the target platform are turned on, thus potentially contending for the shared hardware resources. In this section, firstly the different experiments performed are explained in detail. Afterwards, results collected from the PMCs are summarized and discussed comparing them with the ones obtained with the tests performed on single core mode.

6.2.1 List of experiments

Several experiments are defined to be executed for the Zynq UltraScale EG+ board. In particular, 36 experiments are performed, noting that:

1. The goal of experiments 1-12 is to collect the results from the PMCs of the main core of the RPU cluster, which is **R5_0** of Cortex R5 processors. Those experiments are listed in Table (6.11).

2. For what concerns the remaining experiments, they collect results of the PMCs related to the main core of the Cortex A53 cluster, which is **A53_0**. In this case, 24 experiments are performed, which are listed in two different tables in order to distinguish the ones focused just on load instructions (6.12) and the ones aimed to perform just store operations (6.13).

6.2.2 Task analysis: main core of the Cortex R5 cluster

Considering the experiments from number 1 to 12 (Table 6.11), it is possible to note that the first 6 ones focus on load instructions, whereas the other ones focus on store instructions.

Task analysis - Main core R5_0						
#	**R5_0**	R5_1	A53_0	A53_1	A53_2	A53_3
Exp1	**LoadL1**	LoadL1	LoadL1	LoadL1	LoadL1	LoadL1
Exp2	**LoadL1**	LoadMem	LoadL1	LoadL1	LoadL1	LoadL1
Exp3	**LoadL1**	LoadMem	LoadMem	LoadMem	LoadMem	LoadMem
Exp4	**LoadMem**	LoadL1	LoadL1	LoadL1	LoadL1	LoadL1
Exp5	**LoadMem**	LoadMem	LoadL1	LoadL1	LoadL1	LoadL1
Exp6	**LoadMem**	LoadMem	LoadMem	LoadMem	LoadMem	LoadMem
Exp7	**StoreL1**	StoreL1	StoreL1	StoreL1	StoreL1	StoreL1
Exp8	**StoreL1**	StoreMem	StoreL1	StoreL1	StoreL1	StoreL1
Exp9	**StoreL1**	StoreMem	StoreMem	StoreMem	StoreMem	StoreMem
Exp10	**StoreMem**	StoreL1	StoreL1	StoreL1	StoreL1	StoreL1
Exp11	**StoreMem**	StoreMem	StoreL1	StoreL1	StoreL1	StoreL1
Exp12	**StoreMem**	StoreMem	StoreMem	StoreMem	StoreMem	StoreMem

Table 6.11: Task analysis for main core of Cortex R5 cluster - Microbenchmarks executed in each experiment by each core

In each cell of such table, there are written the abbreviations of the names of the microbenchmarks implemented in each core for the specific experiment to be performed. Therefore, it is important to highlight what they are aimed to do:

- `LoadL1` is the microbenchmark that has to generate high workload on the L1 Data cache using just load instructions. This means that it is focused on memory read operations and no cache misses are experienced (load hits).
- `LoadMem` aims to cause systematic cache misses in the L1 Data cache, forcing the system to perform just memory read operations at the main memory. Therefore, L1 Data cache has to fetch the data always from the main memory (load misses), in order to cause such microbenchmark high contention.
- `StoreL1` is the algorithm that has to cause cache hits in L1 Data cache using store instructions. This means that the same data is written in specific memory addresses that can be completely mapped in the cache lines of the L1 Data cache.

- `StoreMem` has the goal to generate cache misses in L1 Data cache with store operations (store misses). In fact, this microbenchmark writes data in many memory addresses in such a way that they cannot be mapped in the first level cache and eviction is experienced for each store instruction. Note that this microbenchmark uses a very large array size (2 MB) instead of the 64 kB one used in the last set of results shown before so that roughly no store hits occur despite the pseudo-random replacement policy in the L1 Data cache.

The descriptions given about the aforementioned microbenchmarks give the possibility to understand the goal of each experiment. In the first experiment, for instance, all the cores are running at the same time the `LoadL1` microbenchmark, meaning that the analysis aims to observe if the task to be handled by the main core of Cortex R5 cluster is affected by the actions of the other cores. In fact, since that no cache misses should be experienced, the data handled in each core has to be kept in each local L1 cache. Therefore, no contention happens and this can be noted with the results obtained with the PMCs of the target processor architecture.
In general, experiments are built to test the impact of contention incrementally. For instance, `exp1` keeps all activities local in L1, while `exp2` makes the neighbor core in the same cluster to create contention, keeping the main core with local accesses in L1. The third experiment (`exp3`) raises the level of contention with memory accesses from the cores in the other cluster. Then, `exp4`, `exp5` and `exp6` are analogous, but also with the main core accessing memory significantly so that the impact of contention is even worse.

6.2.3 Task analysis: main core of the Cortex A53 cluster

For what concerns the remaining 24 experiments, they observe the dynamics of the main core in the APU of the target Zynq board when load instructions (Table 6.12) and store instructions (Table 6.13) are performed.
Each cell of the aforementioned tables represents the microbenchmarks employed in each core for each experiment, as explained in Section (6.2.2) for Table (6.11). Note that the same (conceptual) microbenchmarks are used

also for these cases.
Two additional microbenchmarks devised for the L2 cache of Cortex A53 processors are included in the task analysis of the Cortex A53_0 core, which are exclusively used for Cortex A53 processors since L2 cache is present just in the APU:

- `LoadL2` is the one that aims to cause cache hits in the L2 cache through the use of load instructions. Therefore, cache misses are experienced in L1 Data cache and no information is fetched from the main memory.

- `StoreL2` is the code introduced for the experiments from number 25 to 36. The goal of this algorithm is to force the storing of specific data in L2 cache, causing cache misses in L1 and cache hits in L2. Therefore, after some iterations, such data is not fetched from the main memory as it happens in the case of the `LoadL2` microbenchmark and it is found always in L2 Data cache.

Due to the presence of the L2 Data cache that is a hardware resource shared among the four cores of the Cortex A53 cluster, we had to perform more experiments in order to consider the different scenarios that can occur in avionics and/or automotive systems when consolidating some critical tasks. The goal of each experiment is quite similar to that of the experiments from number 1 to 12 performed for R5_0 (see Table 6.11). The experiment number 16, for instance, is almost equal to experiment number 3, since that each core executes the `LoadMem` microbenchmark, except the main core, which executes the `LoadL1` microbenchmark. These experiments are different just because the target core is not R5_0 but A53_0, meaning that `LoadL1` and `LoadMem` are executed now by this last one.
The experiments from number 17 to 20 and from 29 to 32, for instance, are different from the previous ones. In fact, both sets of experiments aim at forcing cache hits in the L2 Data cache for the main core of the APU when there are contenders that can create contention. Note that such contenders are all the other processors, namely the three cores of Cortex A53 and the two cores of Cortex R5 clusters.

Task analysis - Main core A53_0						
#	R5_0	R5_1	**A53_0**	A53_1	A53_2	A53_3
Exp13	LoadL1	LoadL1	**LoadL1**	LoadL1	LoadL1	LoadL1
Exp14	LoadL1	LoadL1	**LoadL1**	LoadL2	LoadL2	LoadL2
Exp15	LoadL1	LoadL1	**LoadL1**	LoadMem	LoadMem	LoadMem
Exp16	LoadMem	LoadMem	**LoadL1**	LoadMem	LoadMem	LoadMem
Exp17	LoadL1	LoadL1	**LoadL2**	LoadL1	LoadL1	LoadL1
Exp18	LoadL1	LoadL1	**LoadL2**	LoadL2	LoadL2	LoadL2
Exp19	LoadL1	LoadL1	**LoadL2**	LoadMem	LoadMem	LoadMem
Exp20	LoadMem	LoadMem	**LoadL2**	LoadMem	LoadMem	LoadMem
Exp21	LoadL1	LoadL1	**LoadMem**	LoadL1	LoadL1	LoadL1
Exp22	LoadL1	LoadL1	**LoadMem**	LoadL2	LoadL2	LoadL2
Exp23	LoadL1	LoadL1	**LoadMem**	LoadMem	LoadMem	LoadMem
Exp24	LoadMem	LoadMem	**LoadMem**	LoadMem	LoadMem	LoadMem

Table 6.12: Task analysis for main core of Cortex A53 cluster - Microbenchmarks executed in each experiment by each core only based on load instructions

Task analysis - Main core A53_0						
#	R5_0	R5_1	**A53_0**	A53_1	A53_2	A53_3
Exp25	StoreL1	StoreL1	**StoreL1**	StoreL1	StoreL1	StoreL1
Exp26	StoreL1	StoreL1	**StoreL1**	StoreL2	StoreL2	StoreL2
Exp27	StoreL1	StoreL1	**StoreL1**	StoreMem	StoreMem	StoreMem
Exp28	StoreMem	StoreMem	**StoreL1**	StoreMem	StoreMem	StoreMem
Exp29	StoreL1	StoreL1	**StoreL2**	StoreL1	StoreL1	StoreL1
Exp30	StoreL1	StoreL1	**StoreL2**	StoreL2	StoreL2	StoreL2
Exp31	StoreL1	StoreL1	**StoreL2**	StoreMem	StoreMem	StoreMem
Exp32	StoreMem	StoreMem	**StoreL2**	StoreMem	StoreMem	StoreMem
Exp33	StoreL1	StoreL1	**StoreMem**	StoreL1	StoreL1	StoreL1
Exp34	StoreL1	StoreL1	**StoreMem**	StoreL2	StoreL2	StoreL2
Exp35	StoreL1	StoreL1	**StoreMem**	StoreMem	StoreMem	StoreMem
Exp36	StoreMem	StoreMem	**StoreMem**	StoreMem	StoreMem	StoreMem

Table 6.13: Task analysis for main core of Cortex A53 cluster - Microbenchmarks executed in each experiment by each core only based on store instructions

6.2.4 Final results and Research Observations

To automatize the process of data acquisition, a set of Bash scripts was used. Since the number of experiments is very large and consequently the number of counters is huge, only the results of the CPU_CYCLES and INST_RETIRED counters are shown in Table (6.14). Final results are shown in Table (6.15), which are represented also in Figure (6.1) for the sake of convenience. Note that each experiment is performed disabling firstly the data prefetcher of all the cores, including the target main core under analysis, to avoid uncontrolled behavior of any core.

In this last table, it is important to highlight that `CPI` stands for *Cycles Per Instruction* and that `Mem` stands for *main memory*. Moreover, all the experiments are categorized in such a way that the reader can understand better the type of microbenchmark studied in the main core, which memory level they are stressing, the type of instructions used (load or store) and the target processing cluster.

The `CPI` values are obtained dividing the counter of the CPU cycles of the target main core (CPU_CYCLES) by the number of instructions executed in the corresponding experiment (INST_RETIRED).

Experiment	CPU_CYCLES	INST_RETIRED
Exp 1	394118	130240
Exp 2	394528	130240
Exp 3	394647	130240
Exp 4	8889679	130240
Exp 5	9049911	130240
Exp 6	9174290	130240
Exp 7	534830	133580
Exp 8	539924	133580
Exp 9	567259	133580
Exp 10	4579026	133313
Exp 11	5082107	133313
Exp 12	12047451	133313
Exp 13	1287787	1282160
Exp 14	1297474	1282160
Exp 15	1305180	1282160
Exp 16	1290713	1282160
Exp 17	4639647	130103
Exp 18	5139346	130103
Exp 19	4832349	130103
Exp 20	4818012	130103
Exp 21	16545149	130102
Exp 22	18928491	130102
Exp 23	17066002	130102
Exp 24	19619430	130102
Exp 25	189313	133609
Exp 26	190034	133609
Exp 27	199632	133609
Exp 28	199599	133609
Exp 29	859128	133251
Exp 30	2045853	133251
Exp 31	1131668	133251
Exp 32	1197269	133251
Exp 33	4268173	133112
Exp 34	5624223	133112
Exp 35	8723100	133112
Exp 36	8361755	133112

Table 6.14: CPU cycles and instructions performed in each experiment

Processing unit	Instructions	Microbenchmark	Experiment	CPI
RPU	Load	L1	Exp 1	3.03
			Exp 2	3.04
			Exp 3	3.04
		Mem	Exp 4	68.26
			Exp 5	69.49
			Exp 6	70.45
	Store	L1	Exp 7	4.01
			Exp 8	4.06
			Exp 9	4.27
		Mem	Exp 10	34.35
			Exp 11	38.13
			Exp 12	90.38
APU	Load	L1	Exp 13	1.00
			Exp 14	1.01
			Exp 15	1.02
			Exp 16	1.01
		L2	Exp 17	35.67
			Exp 18	39.51
			Exp 19	37.15
			Exp 20	37.04
		Mem	Exp 21	127.18
			Exp 22	145.50
			Exp 23	131.18
			Exp 24	150.81
	Store	L1	Exp 25	1.42
			Exp 26	1.43
			Exp 27	1.51
			Exp 28	1.51
		L2	Exp 29	6.45
			Exp 30	15.36
			Exp 31	8.51
			Exp 32	9.00
		Mem	Exp 33	32.07
			Exp 34	42.26
			Exp 35	65.55
			Exp 36	62.83

Table 6.15: Cycles per instruction results

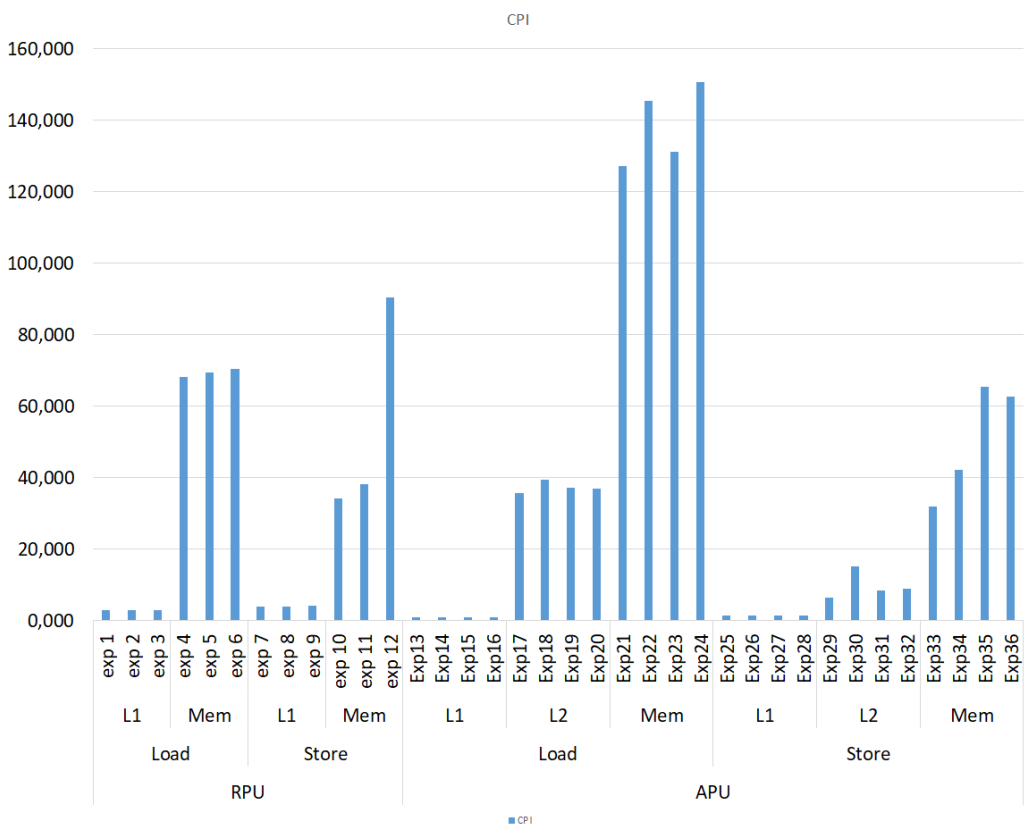

Figure 6.1: Cycles per instruction plot of the experiments with contenders

Focusing on the plot represented in Figure (6.1), instead, it is possible to comment the results in a easier better way. As expected, the CPI values are higher in the experiments that force the specific processing unit to access the main memory, like the ones from number 4 to 6 or from number 21 to 24. In fact, even though the number of instructions performed in the experiments from number 1 to 3 is the same also in the experiments from 4 to 6, the number of execution cycles is quite bigger.

Cortex-R0 results

First of all, we observe that the CPI for experiments accessing only L1 cache are insensitive to contention. For instance, in the case of load hits, experiments 1-3 have roughly the same CPI, and so it is the case for

experiments 7-9 for store hits. Note that, while this behavior is expected, it is not always the case in all architectures since cache snooping and cache inclusion characteristics may lead to some interference even if the task under analysis does not use any shared resource.

Observation 1: *Load hits and store hits are insensitive to contention. Hence, critical tasks operating mostly with local data can be consolidated with any other software without specific constraints.*

In the case of load misses, we observe that the CPI is also nearly constant regardless of contention. A closer look at the results reveals, as pointed out before, that load frequency is lower than store frequency. Thus, the fact that load misses stall execution in the core prevents other contenders to create excessive contention. Hence, the task under analysis, although it experiences some relevant contention, does not suffer a noticeable increase in its CPI thanks to the intrinsic parallelism of the bus-memory interconnection that serves multiple load requests from the different cores simultaneously. In fact, in the case of experiment 6, a very slight CPI increase is observed when memory must serve load requests from all cores simultaneously. We, therefore, identify a key information for task consolidation:

Observation 2: *Load misses cause limited contention in practice, so critical tasks can be consolidated with programs with such profile without experiencing a relevant increase in their execution time.*

When analyzing experiments 10-12, thus for store misses, we realize that, as shown before, CPI without contention is lower than for loads. However, those tasks are much more sensitive to contention that load-based ones. For instance, when adding a store miss contender (`exp11`), CPI of the main core increases by more than 10%:

$$F = \left(1 - \frac{CPI_{exp10}}{CPI_{exp11}}\right) \cdot 100 \approx 10\% \tag{6.11}$$

where `F` corresponds to the percentage growth of CPI in `exp11` with respect

to the CPI in `exp10`.
When placing store miss contenders in all other cores, instead, the CPI grows by a factor of 2.6x, i.e. around 60% higher CPI than the one measured in `exp_10`. The reason behind this behavior is that store misses perform accesses at high rate to memory, caused by increased number of dirty cache line evictions. Thus, this leads to the third key observation:

Observation 3: *Store misses cause high contention in memory, so critical tasks sensitive to contention must not be consolidated with store-miss programs as contenders.*

Cortex-A53 results

As for the R0 cores, load-hit and store-hit tasks are highly insensitive to contention. This can be seen comparing experiments 13-16 (load hits) as well as 25-28 (store hits). As shown, the CPI remains almost constant regardless of the contention caused by the tasks in the other cores, whose impact is negligible regardless of whether they access their local L1 caches, the shared L2 cache or main memory. Hence, similar conclusions can be reached for load-hit and store-hit programs in the A53 and R0 cores:

Observation 4: *Load hits and store hits are insensitive to contention, regardless whether it occurs in the L2 cache or in memory. Hence, critical tasks operating mostly with local data can be consolidated with any other software without specific constraints also in the A53 cores.*

Regarding experiments targeting L2 cache, they have been devised so that their data fits comfortably in L2 and hence, contention can only occur in the access ports and queues, but not due to mutual evictions. In practice, mutual evictions can be avoided using cache partitioning, which is not considered in the scope of this Master thesis for the sake of simplicity.
Experiments 17-20 evaluate the case for load L2 hits. We observe that making contenders access also the L2 cache with `exp18` and using the formula for CPI growth (6.11) leads to a CPI increase of around 10% with

respect to the CPI measured with `exp17` where L2 cache is not accessed by contenders. Instead, if contenders access main memory (`exp19-exp20`), their L2 access frequency is lower with respect to the one obtained with `exp17` and their contention is lower (below 5%). Hence, load L2 hit tasks are not very sensitive to contention and only contention in the access to L2 is relevant. From these results, we raise the following observation for software consolidation:

Observation 5: *Load L2 hits are highly insensitive to contention and only they may suffer some little contention if contenders access L2 frequently. Hence, Load L2 hit tasks may be better consolidated with tasks keeping data locally in L1 or accessing mostly memory, but integrating them with L2-hungry tasks has limited effects.*

When analyzing the case of store L2 hits (`exp29-exp32`), we observe similar trends but with much larger magnitude. In particular, recalling formula for CPI growth (6.11), store L2 hits caused by contenders of `exp30` can make CPI grow above 2x (around 60%) the one measured with `exp29` where contenders do not cause store hit in L2 cache. Regarding memory traffic generated with `exp31` and `exp32`, it generates lower L2 cache contention with respect to the one generated with `exp30` since contenders cause L2 cache misses, causing more contention in main memory. By the way, L2 cache contention is unavoidable also in such experiments and a CPI growth of the task under analysis by 30-40% is experienced with respect to measured with `exp29` according to (6.11). Overall, this exposes two key facts: store L2 hits create very high contention and are very sensitive to contention. The main reason for this behavior is the fact that stores are normally processed asynchronously, so that the latency of store L2 hits can be normally hidden with some queues and buffers in L1 and L2 caches. However, as long as those buffers and queues get saturated due to contention, back pressure is created and execution in the core gets stalled, thus leading to significant relative CPI increases. Hence, we raise the following observations:

Observation 6: *Store L2 hit critical tasks must not be consolidated with tasks creating high L2 contention. Hence, they can be consolidated with*

tasks keeping their data locally in L1 or, at most, with load misses tasks that, despite creating some contention in the L2, such contention is limited.

Observation 7: *Store L2 hit tasks have the ability to increase contention in L2 dramatically. Hence, they are compatible with critical tasks as long as those tasks are insensitive to L2 contention, either because they keep their data locally in L1 or because they already experience high latency accessing main memory so that the relative impact of L2 contention is low.*

Load misses (`exp21-exp24`) have already a high CPI, so the relative impact of contention is low. Still, such impact is relevant when contenders access L2 cache (`exp22`) or memory frequently (`exp23-exp24`), which may increase the CPI. For instance, according to (6.11), CPI measured with `exp22` increases by around 13% compared to the CPI measured with `exp21`, while with `exp24` CPI increases by around 17%. Thus, the following observation holds:

Observation 8: *Load misses are sensitive to contention to some extent. Hence, consolidating load misses critical tasks with tasks that mostly access L1 globally is the most convenient solution. Still, having some L2 or memory intensive tasks running together has limited effects.*

Finally, store misses (`exp33-36`) are highly sensitive to contention, especially if such contention occurs in memory. In particular, recalling formula for CPI growth (6.11), store L2 hits caused by contenders of `exp34` can increase CPI by around 25% with respect to the CPI measured with `exp33` where L2 cache is not accessed by contenders. Instead, store misses contenders (`exp35-exp36`) make the CPI of the task under analysis grow by 2x (around 50%) the one measured with `exp33`:

Observation 9: *Store misses are highly sensitive to contention, especially if such contention occurs in memory. Hence, store misses critical tasks must be consolidated with tasks causing low contention such as, for instance, L1 hit tasks and those accessing shared resources not too often (e.g. load misses).*

Observation 10: *Store misses create very high contention in memory. Therefore, critical tasks running together with this type of contenders must keep their data locally in L1 as much as possible to keep their CPI low.*

Chapter 7

Conclusions

The adoption of high-performance hardware platforms on critical real-time embedded systems is a need in domains such as automotive and avionics where the real-time applications are constantly increasing in complexity. This work proposed the integration and adaptation of a measurement-based methodology based on microbenchmarks, to detect interference phenomena in such platforms. In particular, the Zynq UltraScale+ EG has been analyzed to study the interference that cores can experience when accessing shared hardware resources. From the results, it has been shown that software consolidation must be performed carefully to guarantee that the interference does not affect the execution time of critical real-time tasks. The work in this thesis analyzes in detail how contention occurs when accessing different shared hardware resources, such as shared caches and main memory, across different computing clusters, i.e. the real-time Cortex-R5 one and the high-performance Cortex-A53 one, and for different assembly instructions.
Exploiting the proposed benchmarks, it is possible to make valuable conclusions which allow end users to perform appropriate task consolidation and to detect dangerous interference. In particular, our observations provide an hint on what tasks should execute concurrently and what task types must not do it. This limits the contention on shared resources experienced by the task, thus leading to an efficient use of shared hardware resources. Ultimately, this provides evidence that in some cases the WCETs increase slightly with respect to the original ones obtained for tasks in isolation. To summarize, the benchmarks and analysis methods proposed in this thesis can be used

to empirically determine the maximum contention between tasks, make a preliminary quantitative analysis of such interference, select the appropriate task-resource mapping, test the performance counters behavior and even detect errors in their specification (as experienced in this work).

7.1 Future development

The research findings presented in this thesis open the door to a number of research paths:

- Task scheduling. While observations are provided in the form of guidelines for scheduling purposes, scheduling algorithms building upon profiled information from tasks (with PMCs) can be build on top of our observations so that efficient schedules are devised automatically, thus minimizing (or even removing) user intervention.

- Contention models. Even if our analysis reveals the magnitude of execution time increase incurred due to contention, the actual (fully-reliable) bounds cannot be computed. By further testing the platform, we can obtain reliable bounds with the help of analytical contention models, that can be devised so that impact of contention on a given task can be reliably and tightly estimated without having it to run concurrently with other tasks of the system. This can be a very valuable input for task scheduling so that scheduling decisions are based on actual predictions rather than on qualitative observations.

- Other components. The target of the analysis in this thesis has been general purpose processor cores. However, the platform includes other computing resources such as a GPU and an FPGA. Analyzing the contention experienced by those components remains as future work to enable their effective use while having guarantees about their impact of contention on execution time.

Bibliography

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," pp. 365–376, June 2011.

[2] Gabriel Alejandro Fernandez Diaz, *"Enhancing Timing Analysis for COTS Multicores for Safety-Related Industry: a Software Approach"*. PhD thesis, Universitat Politecnica de Catalunya, 2018.

[3] F. Reghenzani, G. Massari, and W. Fornaciari, "The Real-Time Linux Kernel: A Survey on PREEMPT_RT," vol. 52, February 2019.

[4] Wikipedia, "Real-time operating system."

[5] András Vajda, *Programming Many Core Chips*, 2011.

[6] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "Assessing the Suitability of the NGMP Multi-core Processor in the Space domain," *EMSOFT' 12*, pp. 175–184, October 7, 2012.

[7] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[8] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, "WCET analysis methods: Pitfalls and challenges on their trustworthiness," in

10th IEEE International Symposium on Industrial Embedded Systems (SIES), pp. 1–10, June 2015.

[9] E. Mezzetti and T. Vardanega, "On the industrial fitness of WCET analysis," in *Worst-Case Execution Time (WCET) Analysis Workshop*, 2011.

[10] Rapita Systems Ltd., "RapiTime - worst case execution time (WCET) analysis for critical systems. https://www.rapitasystems.com/products/rapitime," 2018.

[11] S. J. Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean, "Open challenges for probabilistic measurement-based worst-case execution time," pp. 69–72, September 2017.

[12] F. Reghenzani, G. Massari, and W. Fornaciari, "The misconception of exponential tail upper-bounding in probabilistic real-time," pp. 1–4, December 2018.

[13] ARM, *ARM Cortex-A Series: Programmer's Guide for ARMv8-A*, 2015.

[14] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, "A Cache Design for Probabilistically Analysable Real-time Systems," in *Design Automation Test in Europe (DATE) Conference*, 2013.

[15] SPARC, *The SPARC Architecture Manual*, 1991,1992.

[16] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-critical Environments," *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 34:1–34:25, Jan. 2012.

[17] G. Fernandez, J. Jalle, J. Abella, E. Quiñones, T. Vardanega, and F. J. Cazorla, "Resource Usage Templates and Signatures for COTS Multicore Processors," in *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, (New York, NY, USA), pp. 155:1–155:6, ACM, 2015.

[18] E. Díaz, E. Mezzetti, L. Kosmidis, J. Abella, and F. J. Cazorla, "Modelling Multicore Contention on the AURIXTM TC27x," in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), pp. 97:1–97:6, ACM, 2018.

[19] "SAFURE - Safety And Security By Design For Interconnected Mixed-Critical Cyber-Physical Systems. https://safure.eu," 2018.

[20] G. Fernandez, F. J. Cazorla, and J. Abella, "Consumer Electronics Processors for Critical Real-Time Systems: a (Failed) Practical Experience," in *9th European Congress on Embedded Real Time Software and Systems (ERTS^2)*, 2018.

[21] J. Nowotsch and M. Paulitsch, "Leveraging Multi-core Computing Architectures in Avionics," in *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, (Washington, DC, USA), pp. 132–143, IEEE Computer Society, 2012.

[22] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement," in *2014 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 109–118, July 2014.

[23] I. Agirre, J. Abella, M. Azkarate-Askasua, and F. J. Cazorla, "On the tailoring of CAST-32A certification guidance to real COTS multicore architectures," in *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–8, June 2017.

[24] S. Girbal, J. Le Rhun, and H. Saoud, "METrICS: a Measurement Environment for Multi-Core Time Critical Systems," in *9th European Congress on Embedded Real Time Software and Systems (ERTS^2)*, 2018.

[25] D. Griffin, B. Lesage, I. Bate, F. Soboczenski, and R. I. Davis, "Forecast-based Interference: Modelling Multicore Interference from Observable Factors," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, (New York, NY, USA), pp. 198–207, ACM, 2017.

[26] G. Fernandez, J. Abella, E. Q. nones, L. Fossati, M. Zulianello, T. Vardanega, and F. J. Cazorla, "Seeking Time-Composable Partitions of Tasks for COTS Multicore Processors," in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pp. 208–217, April 2015.

[27] K. Ganesan, J. Jo, W. L. Bircher, D. Kaseridis, Z. Yu, and L. K. John, "System-level Max Power (SYMPO): A Systematic Approach for Escalating System-level Power Consumption Using Synthetic Benchmarks," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 19–28, ACM, 2010.

[28] NXP Semiconductors, *Measuring Current in i.MX Applications. Application Note, Document Number: AN5381, https://www.nxp.com/docs/en/application-note/AN5381.pdf*, 2016.

[29] ARM, *ARM Architecture Reference Manual: ARMv8 for ARMv8-A architecture profile*, December 19 2017.

[30] Wikipedia, "Locality of reference."

[31] ARM, *Cortex-R5: Technical Reference Manual*, 2010-2011.

[32] P. Benedicte, C. Hernandez, J. Abella, and F. J. Cazorla, "Rpr: A random replacement policy with limited pathological replacements," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, (New York, NY, USA), pp. 593–600, ACM, 2018.

[33] M. Goossens, F. Mittelbach, and A. Samarin, *The LaTeX Companion*. Reading, Massachusetts: Addison-Wesley, 1993.

[34] D. Knuth, "Knuth: Computers and typesetting."

[35] XILINX, *Zynq UltraScale+ Device: Technical Reference Manual*, December 22, 2017.

[36] ARM, *ARM Cortex-A53 MPCore Processor: Technical Reference Manual*, 2016.

[37] ARM, *ARM Cortex-R Series: Programmer's Guide*, 2014.

[38] Wikipedia, "CPU Cache."