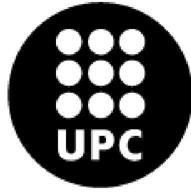


UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONA**TECH**

Facultat d'Informàtica de Barcelona

Modelling Contention in Multicore Hardware Resources during Early Design Stages of Real-Time Systems

Author:

David Trilla Rodríguez

Master in Innovation and Research in Informatics
High Performance Computing Specialization

July, 2016

Director:

Francisco J. Cazorla — IAAA-CSIC, Barcelona Supercomputing Center

Codirector:

Jaume Abella — Barcelona Supercomputing Center

Tutor:

Mateo Valero — Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona Supercomputing Center

---

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Abstract

This thesis presents a modelling approach for the timing behavior of real-time embedded systems (RTES) in early design phases. The model focuses on multicore processors – accepted as the next computing platform for RTES – and in particular it predicts the contention tasks suffer in the access to multicore on-chip shared resources. The model presents the key properties of not requiring the application's source code or binary and having high-accuracy and low overhead. The former is of paramount importance in those common scenarios in which several software suppliers work in parallel implementing different applications for a system integrator, subject to different intelectual property (IP) constraints. Our model helps reducing the risk of exceeding the assigned budgets for each application in late design stages and its associated costs.

The timing contention model builds on the concept of an execution profile that is extracted by each software supplier in isolation for each of its applications, and that provides an estimate of the usage of resources made by the application. Execution profiles of the different applications – which can be shared among suppliers without jeopardizing confidentiality (IP) of each application – are then combined with our contention model to derive an application's execution time increase due to multicore contention.

Contents

Abstract **2**

1 Introduction **9**

1.1 Structure of the Thesis 11

2 Background **12**

2.1 Timing Analysis 14

2.1.1 Static Deterministic Timing Analysis 15

2.1.2 Measurement-Based Deterministic Timing Analysis 15

2.1.3 Probabilistic Timing Analysis 16

2.2 Scheduling Domains 17

3 Contention Modeling **19**

3.1 Application Process 21

3.2 Histogram Approach 21

3.2.1 Histogram Realization 23

3.3 Restricted Access Interleaving Information 24

3.4 Global Notation & Parameter Description 25

4 Execution Profile **28**

4.1 Profile Generation 28

4.2 Profile Information 29

5 Cache Contention Model **32**

5.0.1 Set collision distribution 34

5.0.2 Increment in stack distance 34

5.0.3 Final Step 35

5.1 Example 36

6 Bus and Memory Contention Model **39**

6.1 Considerations 41
6.2 Example 42

7 Assumptions & Simplifications 43
7.1 General approach 43
7.2 Core model 43
7.3 Cache model 44
7.4 Bus model 44
7.5 Addressable unit 44

8 Evaluation 45
8.1 Methodology & Experimental Framework 45
8.1.1 Target platform 45
8.2 Benchmarks 46
8.2.1 Workloads 47
8.3 Metrics 47
8.4 Results 48
8.4.1 Average-Based Model 48
8.4.2 Cache Contention Model 48
8.4.3 Bus Contention Model 49
8.4.4 Multicore Execution Time 50
8.4.5 Performance and Overhead 51

9 Related Work 53

10 Conclusions & Future Work 55
10.1 Future Work 56

11 Acknowledgements 57

12 Published Work 58

List of Figures

2.1 Block Diagram of the 4-core NGMP architecture. 13
2.2 Example of the MBPTA probability-WCET curve 16
2.3 EP usage to derive Δt 17

3.1 Application steps of our approach. 20
3.2 Example of use of the histogram-based modeling approach 22
3.3 Access Sequence *Seq1* generated by a task τ_j 26

5.1 Visualization: Window of vulnerability 36
5.2 Visualization: Contenders' injection 37
5.3 Visualization: Contenders' collision 37
5.4 Visualization: Discarding repeated accesses 38

8.1 Accuracy of the cache contention model 48
8.2 Accuracy of the bus contention model 49
8.3 Performance accuracy of the complete model. Results have been normalized w.r.t. the actual measurements. 50
8.4 Average overheads of our contention prediction model. 51

List of Tables

2.1 Timing analysis techniques summary 14

3.1 Global notation. 25

4.1 Specific information in the Execution Profile and Notation 29
4.2 Operation types in the NGMP and their assumed latencies 30

8.1 Workloads used for evaluation 47

Acronyms

API application programming interface.

BCM bus contention model.

CCM cache contention model.

CM contention model.

EDP early design phases.

EP Execution Profile.

ESA European Space Agency.

FP floating point.

FPU floating point unit.

IP intelectual property.

LDP late design phases.

MBDTA measurement-based deterministic timing analysis.

MBPTA measurement-based probabilistic timing analysis.

MBTA measurement-based timing analysis.

NGMP Next Generation Microprocessor.

OEM original equipment manufacturer.

PTA probabilistic timing analysis.

RTES real-time embedded systems.

SDTA static deterministic timing analysis.

SPTA static probabilistic timing analysis.

STA static timing analysis.

VM virtual machine.

WCET worst case execution time.

Chapter 1

Introduction

The design of real-time embedded systems (RTES) comprises multiple development phases that affect both hardware and software components. During early design phases (EDP) critical decisions about the hardware to use and the time budget assigned to each task are taken. This may affect not only the subsequent design phases, but even the final delivered product.

Those decisions taken during EDP are mainly affected by uncertainties and lack of information in terms of functional and non-functional requirements of the system (e.g. how often will a certain task be triggered, what will be the execution time of a task under different hardware resources?, will a certain schedule be feasible?). During EDP all the components of a system are in a preliminary state and this affects their timing requirements too. With this conditions developers have to asses this uncertainties by bounding them with estimates. This may lead to two different results:

- Overestimating time budgets: In this case, the developers have assumed safety bounds that exceeds the worst cases that could be observed on the system. This may result in an over-designed system with significant spare capacity and an unnecessary increased cost in wasted resources.
- Underestimating time budgets: In this case, the developers did not successfully predict the worst cases that could be observed on the system. This may lead to many changes in the task structure in the late design phases (LDP) of the system, which are costly and might even delay the delivery of the product.

This situation gets even worst if we add-in multicores into the equation. Multicores which are being considered for the future RTES as the main computing platforms. In this type of hardware tasks running on the different cores have its execution time affected by the usage of the shared resources that their co-runner tasks do. This is so because tasks interfere each other in many difficult to predict ways, thus affecting the predictability of the execution time and its estimates.

In those cases in which software is developed by different software suppliers, not only the predictability of the system is affected, but also the relationship between software developers and original equipment manufacturers (OEMs). This is so because since the timing estimates depend now on information of other companies or even competitors, that might be protected under IP rights.

In the context defined in this thesis, each software supplier is provided with a virtual machine (VM) that mimics the functional behavior of the target hardware, the NGMP in our case. This allows the supplier to develop and validate the functionality of its software. Each VM can be attached a timing simulator of the underlying multicore processor to derive timing estimates including the impact of contention. The main problem of this approach is that timing simulators incur a high timing overhead: virtualization incurs performance penalties that are as low as few percentage points and can range up to 1x-2x slowdown depending on the virtualization technology and whether the host's ISA is the same as the simulated ISA. Instead, full-fledged timing simulators can be much slower because for each simulated instruction the timing simulator executes hundreds or even thousands of native instructions to model the delays incurred by the simulated instruction on the simulated CPU, cache, interconnection network, etc.

This may lead to slowdowns in the 100x-1000x range. This is undesirable, for software suppliers who, despite willing to obtain timing estimates for their applications, cannot pay this overhead in the speed of the VM. Our approach i) controls time overhead by performing a characterization of each application in isolation, that despite being a slow process it is performed only once per application; and ii) speeds up the much more frequent computation of contention. Furthermore, detailed timing simulators require information about co-runner tasks that is unlikely to be available due to IP restrictions. This work will also refer to the interference as contention and contention modeling as the process of estimating the inter-task interference.

To develop a solution to these problems, we will focus on a specific hardware, the Next Generation Microprocessor (NGMP). The NGMP is a good representative of multicores used in RTES and it is in the European Space Agency (ESA) roadmap for becoming the processor used for the future missions and will be maintained for years [5].

The objective of this thesis is to present a solution to the mentioned problems that affects the EDP in form of a model. This model will try to provide the timing estimates when the target architecture is a multicore under virtualized environments. The model will derive the amount of inter-task interference involved in using shared hardware resources and use it to provide an increase in execution time of a task under.

1.1 Structure of the Thesis

This thesis is structured as follows. The chapter 2 describes in which context this concepts are set out, regarding the industrial environment where it could be applied and how already the timing analysis process works. Chapters 3 and 4 describe the basic concepts of this model, what it intends to do, with what information and how to acquire it. Chapters 5 and 6 describe the details of the model, how the behavior under a multicore scenario is represented in the shared cache and in the bus. Chapter 7 will provide numerical data as part of the evaluation of the model. To conclude chapters 8, 9 and 10 close with the related work, conclusions and future work to expand this thesis respectively.

Chapter 2

Background

In single-core integrated-architectures (such as IMA [1] in avionics and AUTOSAR [4] in automotive) early in the design the OEMs assigns a CPU quota (budget) to each software supplier – together with the functionality to perform. In terms of timing, OEMs usually implement budgets via time partitioning: time is split into windows each of which is assigned to a different application, and hence to its corresponding supplier. From the supplier point of view, other than some overheads due to context switches, time analysis of its applications can be done in isolation.

Interestingly, the interaction in I/O resources can be handled via forcing that the I/O operations of an application occur during its assigned window or during a specific period designated for that purpose (e.g. at the end of each time window in the context of cyclic-executive scheduling). Hence, single-core CPUs allow each supplier to easily design applications to fit in its assigned quota or negotiate with the OEM a larger quota. This can occur during the EDP, which reduces the cost of any change that is required on the timing or functional behavior of the system.

Multicores complicate this approach because the timing behavior of an application depends both on its own behavior and its co-runners behavior. Conceptually, the execution time of an application in a multicore (et^{muc}) can be broken down into two components as follows:

$$et^{muc} = et^{solo} + \Delta t \tag{2.1}$$

where et^{solo} is the execution time of the application in isolation and Δt is the execution time increase the application suffers due to contention in the access to multicore shared resources.

While suppliers have confidence on the estimates derived for et^{solo}, the same cannot be said about Δt since it depends on co-runners the supplier does not know, and might not be allowed to know due to IP restrictions. Several studies show that Δt can be as high as et^{solo} [14][15], so it can have a great impact on the scheduling plan defined by the OEM to determine the budget and the specific time windows given to each application. If violations to assigned budgets are discovered during the LDP this may require costly application re-coding, changing the scheduling plan or even changing or adding more multicore CPUs if there are not enough computation capabilities to guarantee the execution of all the required functionalities. This, of course, may significantly increase the overall product (system) cost and time-to-market. Therefore, obtaining early and tight estimates of Δt is of great help to reduce the risk of LDP changes.

There is a general consensus in the literature [8][11] that during the EDP accuracy of the timing estimates is not the only metric to consider, with tight upper-bounding estimates being rather required for LDP. Instead, the speed to obtaining those estimates plays a key role to allow engineers to explore a vast set of design choices in a timely manner. However, no particular figure is reported for the required accuracy in timing predictions during the EDP, which in our view is end-user dependent. In the context of multicores, it has been reported that the impact of contention in execution time can be as high as 20x for some kernels and as high as 5.5x for some EEMBC Automotive benchmarks [10].

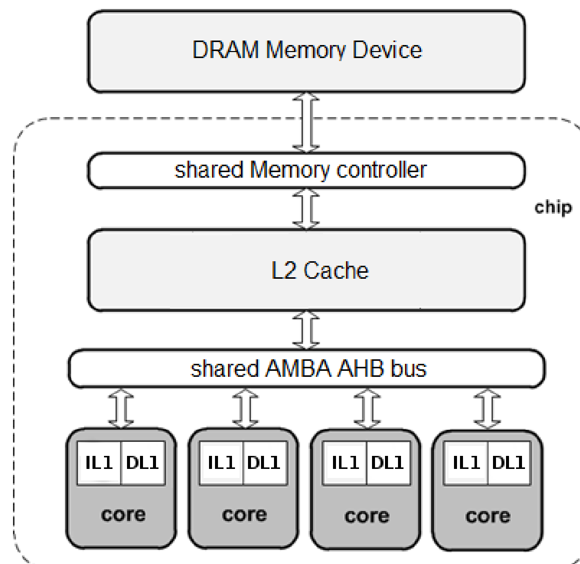


Figure 2.1: Block Diagram of the 4-core NGMP architecture.

Our reference architecture is the Cobham Gailser Next-Generation Multipurpose Processor (NGMP) [5], sketched in Figure 2.1. The NGMP comprises four LEON4 cores, each core having a private instruction cache (iL1) and data cache (dL1), and a global (shared) unified second level cache (uL2). Cores and caches are connected with a bus. A memory controller acts as interface between the processor cores and memory.

The following section describes the state of the art of the field in which this thesis is placed, but in advance let's state that this thesis focuses on measurement-based timing analysis techniques. While a discussion of when static or measurement-based timing analysis approaches are convenient is out of the scope of this thesis, it is a fact that different industries for different systems (functions) use both [24][3]. Hence, research on both techniques is needed so both are able to support multicore timing analysis in early and late phases of the system design.

2.1 Timing Analysis

Timing analysis is one of the most important subjects in RTES, verifying that systems meet their requirements and that they are not going to fail due to unaccomplished timing constraints while doing their critical tasks is a subject of interests for the embedded industry. There are several ways in which today's embedded critical systems are being timing analyzed. The main focus of this techniques is to find the worst case execution time (WCET).The WCET explains the maximum time that a task will take and for which a system has to be designed in order to take that into account. Predicting the WCET is a difficult task and has serious implications.

To fulfill this objective the industry follows several distinct methods [23], which we summarize in the following table.

	Static	Measurement
Deterministic	SDTA	MBDTA
Probabilistic	SPTA	MBPTA

Table 2.1: Timing analysis techniques summary

2.1.1 Static Deterministic Timing Analysis

Static deterministic timing analysis (SDTA) approach derives WCET estimates without simulation. Basically, SDTA analyzes the data and control flow paths and builds a model of the hardware that allows to derive timing estimates. In other words, SDTA derives an equation that has as inputs (or variables) highly detailed and highly specific parameters about the behavior of the program and the architecture, the result of this equation is the WCET.

The main problem with this technique, is the need of in depth detail in order to be accurate [3]. The work-around for when not enough information is available is to upper bound the time consumption values for those resources. This is specially true for the microarchitecture, that manufacturers usually try to protect by hiding details and not providing a highly detailed definition of their hardware in their manuals.

2.1.2 Measurement-Based Deterministic Timing Analysis

In the measurement-based deterministic timing analysis (MBDTA) measurements and static analysis are combined to form an hybrid analysis that derives a single WCET but using measured inputs. This is an approach to avoid the strict requirements of SDTA, by doing so, the explicit hardware modeling can be somehow avoided and adding parametrization allows to decrease the complexity of the models [21]. The first phase of MBDTA consists on collecting execution-time measurements of the software programs of interest, this execution times are strongly dependent on (a) the conditions in which the runs are made, since those must represent the conditions during operation, and (b) the selected inputs that should provoke the worst case execution time.

This means that the end user controls the state conditions and inputs and is responsible for that, which at the same time is a difficult task, since for example, deriving all the potential cache layouts or a significant subset of them may lead to a lot of options. This is still a problem in measurement-based timing analysis (MBTA) since it leads to a lack of control that also brings low coverage of the sources of jitter, hence it decreases the confidence on the WCET value provided.

2.1.3 Probabilistic Timing Analysis

Probabilistic timing analysis (PTA) tackles a lot of the problems inherent with the traditional techniques of static timing analysis (STA). The idea is to produce more than one WCET with an assigned probability, that allows the system designer to choose the amount of pessimism desired with a function of how improbable is an event to happen. By this definition, low probability WCETs that are inherent to STA and cannot be discarded, with PTA can be now assumed if the probability of those events of high WCET are irrelevant enough to be observed during operation time of the system. There are two different ways to apply PTA.

Static Probabilistic Timing Analysis

In static probabilistic timing analysis (SPTA) each analyzed event from STA is modeled with a discrete spectrum of latencies with attached probabilities to each one of them. Then in order to obtain WCET this events (i.e. instructions, accesses to the cache) combine their vectors of latency-probability by using convolution operations. By this convolution steps the final result is a limited set of WCETs with attached probabilities.

Measurement-Based Probabilistic Timing Analysis

The measurement-based probabilistic timing analysis (MBPTA) takes a more inductive approach, by collecting execution times and applying extreme value theory, it is able to predict how WCET will behave when the cases of worst execution time are extreme. With an enough number of runs MBPTA is able to depict a curve that provides a value of WCET given a certain probability. We can observe an example of this in Figure 2.2.

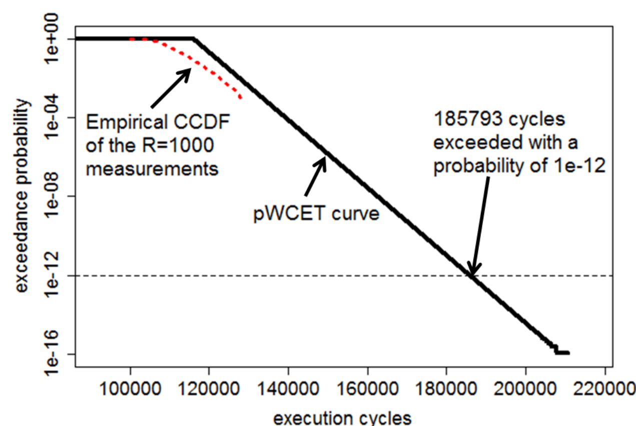


Figure 2.2: Example of the MBPTA probability-WCET curve

The proposal of this thesis fits in the Measurement-Based Deterministic Timing Analysis, since our inputs come directly from measurements but we derive a static model to derivate the WCET estimates.

2.2 Scheduling Domains

The industry has maintained conservative behavior with respect to scheduling, changes impact many fields of development, and when a critical task or schedule has been verified it is preferable not to change it. This is also true for the scheduling techniques, were cyclic executive scheduling remains as widely used in many domains such as avionics and automotive – though this proposal is valid for other scheduling approaches. This approach allows for an almost complete abstraction from the protected information that describes architecture or software analyzed and allows for tighter WCET estimates.

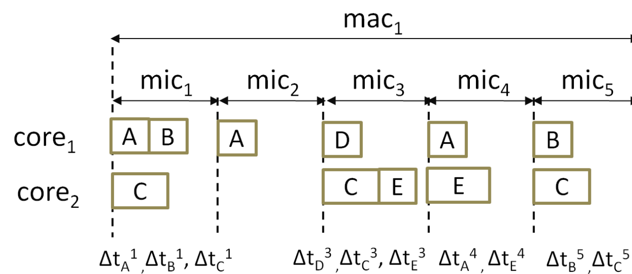


Figure 2.3: EP usage to derive Δt

Cyclic executives divide time into major cycles (*mac*), which are further divided into minor cycles (*mic*). In each *mic*, several jobs are executed in a non-preemtable manner. Each job is required to finish in a *mic* (also called frame). Usually, *mic*s have the same duration to simplify implementation. While the jobs executed in each *mic* may vary making those *mics* different, all *macs* are identical. That is, each *mac* has exactly the same sequence of *mics* and set of jobs called in each *mic*. Despite its static nature, due to its simplicity a cyclic executive is often the preferred scheduling solution in real-time systems in domains such automotive and avionics in which the ARINC 653 standard recommends its use for partitions [2].

As an illustrative example, Figure 2.3 shows a cyclic-executive based scheduling plan provided by the OEM from where each supplier can determine the co-runners of its application in each minor cycle (*mic*). For instance, in mic_1 applications A and B interact with C in the multicore so it is required to derive Δt_A^1, Δt_B^1 and Δt_C^1, where Δt_A^1 corresponds to Δt of application A in mic_1. If both $et_A^{muc} + et_B^{muc}$ and et_C^{muc} fit in a *mic*, no change to the schedule (for this first *mic*) is required. The same process is repeated for all *mics.* Interestingly, in mic_3 if $et_C^{muc} > et_D^{muc}$ then E suffers no contention from D, i.e. $\Delta t_E^3 = 0$. It is worth noting that in the first iteration of this process between the OEM and the suppliers, the OEM creates a scheduling plan assuming no contention or a nominal contention based on its previous experience. This initial value is refined through the different iterations of our approach.

Chapter 3

Contention Modeling

From now on, this thesis is going to explain the process of modeling contention, to summarize ,the purpose of this thesis is to present and provide a framework with following key capabilities:

- A model that obtains accurate estimates of execution time under high uncertainty environment (corresponding to the EDP).
- Allow the solution to be implemented on any of the actors involved in the system development process without compromising IP constraints.
- Base the proposal on the characterization of the tasks obtained from the measured statistics of its execution in isolation.
- The solution will be light-weigth, and with low overhead. The estimates must be obtained quickly so different scheduling plans can be tested in a faster way than with traditional simulators even under VM environments.

Our approach builds upon the concept of an Execution Profile (EP) which encapsulates for each task information about its resource usage and is described later in Chapter 4. The process to apply our approach involves the steps of generating the EP for each task and then combining several EPs – in accordance with a scheduling plan – by means of a contention model to derive Δt, see Figure 3.1.

The main shared resources we consider in our target architecture (see Figure 2.1) are the uL2 cache, the bus and the memory bandwidth. For the former we explain our contention model in Chapter 5 and for the latter two in Chapter 6. Our model keeps no state information

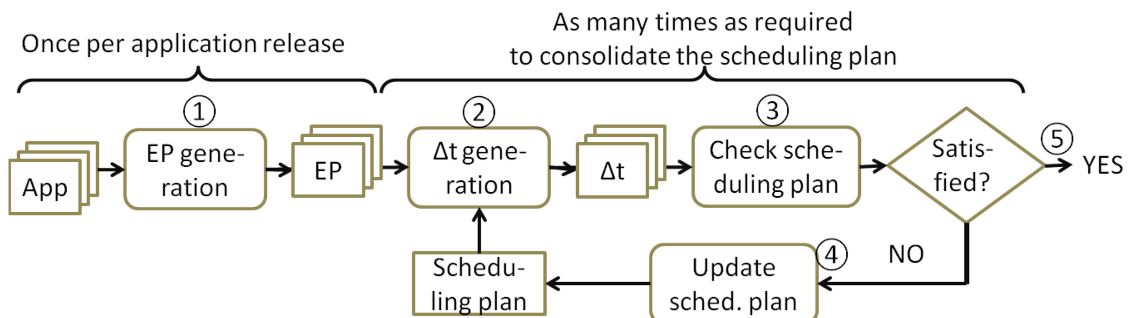


Figure 3.1: Application steps of our approach.

about executed instructions, i.e. it models each instruction in isolation.

First, the cache contention model predicts the increment in number of misses that τ_j suffers due to the contention created by its co-runners, Δm_j^{uL2}. Then it derives the execution time increment (Δt_j^{uL2}) caused by those Δm_j^{uL2} hits in the execution in isolation that become misses due to its co-runners.

In a second step, we account for the impact on the bus and the memory controller that each access suffers when accessing those resources. The access latency of each uL2 hit is increased by the contention on the bus:

$$l_{uL2h}^{muc} = l_{uL2h}^{solo} + \Delta t_{@}^{bus} \tag{3.1}$$

Likewise, the time it takes the memory to serve a uL2 miss increases due to contention on the bus and memory:

$$l_{uL2m}^{muc} = l_{uL2m}^{solo} + \Delta t_{@}^{bus} + \Delta t_{@}^{mem} \tag{3.2}$$

From the number of hits and misses in the private caches; the increment in misses in the uL2; and hit and miss delays in the access to the cache/memory (as presented in Equation 3.1 and Equation 3.2) we derive the overall execution time increase due to the bus and the memory, called Δt_j^{BUS} and Δt_j^{MEM}, respectively. With this, the overall execution time on multicore is predicted as:

$$et_j^{muc} = et_j^{solo} + \Delta t_j^{uL2} + \Delta t_j^{BUS} + \Delta t_j^{MEM} \tag{3.3}$$

3.1 Application Process

At the core of our approach we find the *Contention Model* (or *CM*), which combines (mixes) the EP of those applications that co-run in the multicore to predict Δt, see ② in Figure 3.1.

The OEM distributes the scheduling plan to every supplier together with the EP of all applications. This allows each supplier to determine those applications that are co-runners of its own ones. Each supplier uses the CM to estimate Δt (②) for each of its applications. Δt for each application along its corresponding et^{solo} is sent back to the OEM. If there is no violation of the budgets (③), the scheduling plan is deemed as valid (⑤). On the contrary, the OEM can increase the budget given to a supplier – if some slack is available – or change the scheduling plan. On its side, the supplier can also try to reduce the CPU requirements of its application (④).

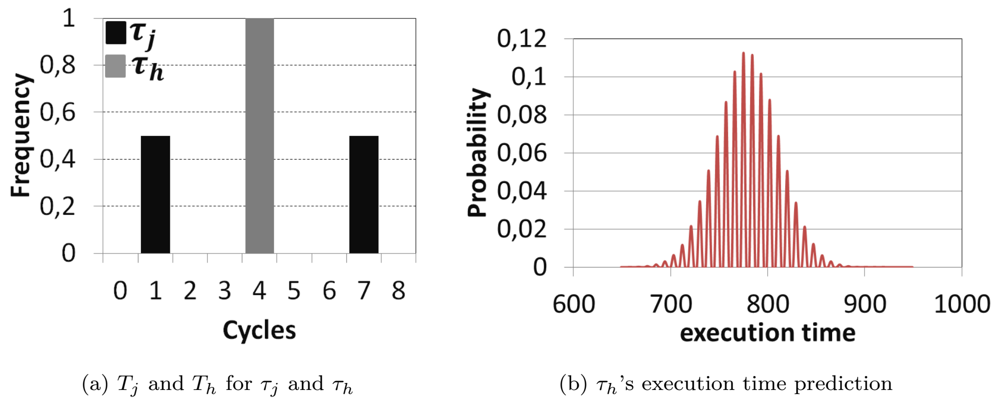
3.2 Histogram Approach

Under a simulator environment, the state of the hardware model is kept in software data structure. Caches are a clear example of this, and are modelled by a two dimensional table, with each cell being a `struct` containing information about its LRU bits, validity bits, data, etc. On each access, the software has to search its internal data structures and update its contents and this is well known to be a time consuming process. In order to reduce simulation speed, we abstract our model from the execution history and will simulate each instruction in isolation.

We use the information in the EPs either from the task under analysis τ_j and the contender tasks to make a prediction in their timing behavior. Most of this information will be stored in form of histograms or distributions, since we find that these kind of statistics are able to build a representative scenario and are easy to capture and allow for a light weight model to be accurate enough. The following example demonstrates the need of this distributions (histograms):

Given two tasks τ_j and τ_h, we want to model the impact on their execution time when both tasks share a signle-entry cache. For this case τ_j accesses have a given address and τ_h accesses go to a different address. Cache hits take 1 cycle, while misses take 10 cycles. Under this scenario, consecutive accesses from the same task are hits, while interleaved accesses evict each other's data and thus are misses. Basically how frequent each of the tasks accesses the cache, will tell how often they produce a consecutive access and how often the other task

evicts data. Let's assume the available data about each of this two tasks to be the frequency of access as depicted in Figure 3.2(a). These histograms, which are respectively called T_j and T_h, depict how every τ_h access is sent to cache 4 cycles after the last one completed, while 50% of τ_j requests are sent one cycle after the previous one and the other 50% every 7 cycles after the previous one completes.



(a) T_j and T_h for τ_j and τ_h (b) τ_h's execution time prediction

Figure 3.2: Example of use of the histogram-based modeling approach

Let's analyze this case under two possible methods:

- ***Average***: If we use information about average both τ_j and τ_h are assumed to send one request to the cache at the same rate of every 4 cycles. As a result all accesses would interleave and be misses. If τ_j performs 100 accesses, its predicted execution time would be 1,000 cycles. This fails to capture the fact that τ_j has a bimodal distribution, which means that the number of requests from τ_j among requests of τ_h varies: it can be 0, 1, 2 or 3 and hence some accesses can be hits.

- ***Histogram***. With the approach based on histograms, for every τ_j access we derive the time since the last access according to its histogram. To that end we define a random variable X modelling frequencies in the histogram as probabilities. For instance, T_j is the random variable capturing the distribution of cycles between consecutive τ_j's accesses.

3.2.1 Histogram Realization

We refer to a realization of a random variable or distribution in form of histogram X as the process to obtain a random sample from that distribution. We notate it, from a histogram X as x, i.e. the name of the random variable but in lower case. Hence, t_j is one particular value obtained from the distribution T_j. For instance, to obtain t_j we generate a random number (r) between 0 and 1, so $r \in [0, 1)$. Given that the time between accesses for τ_j is 1 or 7 cycles with 50% probability each, t_j is 1 or 7 as follows:

$$t_j = \begin{cases} 1\ cycle & if \quad (r < 0.5) \\ 7\ cycles & if \quad (r \geq 0.5) \end{cases}$$

This process for obtaining one value from a random variable, which can be performed for histograms with any number of points and density, is called *realization* as we previously have mentioned. We represent it as $x = rand(X)$, that for the case of the time between accesses is $t_j = rand(T_j)$.

Coming back to the example in Figure 3.2, the histogram based approach results in τ_j and τ_h experiencing hits and misses – as it would be expected based on their frequency of access. To obtain the predicted execution time for τ_j, we perform several *runs of the model*. In each run, the access delay of each access is obtained by performing one realization of T_j (e.g., $t_j = 7$) and as many as needed of T_h to determine how many accesses of τ_h occurred since the previous access of τ_j. The estimate obtained for the execution time distribution for τ_j is as shown in Figure 3.2(b). We observe that the resulting distribution captures the fact that the alignment between tasks' accesses impacts each task's execution time. The average execution time is 779 cycles instead of 1,000 as with the average-based model.

Our results in Section 8 show for real benchmarks that taking averages instead of considering the histogram leads to high inaccuracies since accesses interleave systematically in the same way despite the fact that, in reality, they interleave in many different ways. Instead, histogram-based interleaving captures tasks' access interleaving much more accurately.

3.3 Restricted Access Interleaving Information

A key element in our approach is that we assume no information about the distribution (over time) of the accesses of a given program to hardware resources. For instance, for τ_j in Figure 3.2(a) we know 50% of the accesses are sent 1 cycle after the previous completes and the other 50% 7 cycles. Our model does not record, for instance, information about whether those accesses concentrate on the initial phase of the execution of the program or at the end, that is, how they interleave with other task instructions. If we assume that τ_j and τ_h run in parallel it could be the case that all τ_j accesses occur before those of τ_h, so in reality they are not going to suffer inter-task contention in cache. Likewise, we do not record information about whether τ_j accesses of the different types interleave.

There are several reasons behind this choice. (i) Keeping time-dependent distribution information would increase the size of EP, since we could not summarize it in a histogram but we would need to keep the exact sequence of accesses and how they interleave over time. This would also result in more complex and time-consuming modelling. (ii) This approach would also affect time composability [18][17] since provided contention bounds would be specific to how requests interleave, which changes when tasks suffer any type of shift.

Since our model aims at predicting the worst-case contention among tasks, not the average, whenever two tasks partially overlap in the scheduling plan we pessimistically increase their et^{muc} assuming they fully overlap and hence, suffer continuously high contention. Despite this adds some pessimism, it simplifies EP and makes the CM lighter – which is critical since the CM is used in an iterative manner to adjust the scheduling plan, so it has to incur a small slowdown.

3.4 Global Notation & Parameter Description

Throughout this thesis, special and abbreviated notation will be used to refer to the different statistics the following table summarizes, for an easy and quick reference, the abbreviations used and to what they refer to:

Table 3.1: Global notation.

		Symbol	Description	Comments
per task (τ_j)	cache	TS_j	Time between τ_j's access to the same Set	Apply to each cache: iL1(i), dL1(d) and uL2(u) for the NGMP, e.g., Ki_j and Ku_j are the stack distance of the access to iL1 uL2 respectiv.
		K_j	StacK distance of τ_j's access to cache	
		E_j	sEt distance of τ_j's access to cache	
		H_j	Hit Rate of τ_j's access to cache	
		d_j	Set dispersion of τ_j's accesses to cache	-
		Δm_j^{uL2}	Increment in miss count in uL2 that τ_j suffers due to its contender tasks	-
		et_j^{solo}	Execution Time estimate for τ_j in isolation	-
	Time	et_j^{uL2}	Exec. Time estimate for τ_j factoring in Δm_j^{uL2}	-
		et_j^{muc}	Execution Time estimate for τ_j in multicore	All increments are caused by τ_j's contender tasks accesses to the different hardware shared resources
		Δt_j	Time increment τ_j suffers in multicore	
		Δt_j^{BUS}	Time increment τ_j suffers due to bus sharing	
		Δt_j^{MEM}	Time increment τ_j suffers due to mem. sharing	
		Δt_j^{uL2}	Time increment τ_j suffers due to L2 sharing	
		$etbus_j^{uL2}$	Time task τ_j uses the bus factoring in Δm_j^{l2}	-
		$ubus_j^{uL2}$	τ_j's utilization of the bus factoring in Δm_j^{l2}	-
		$ubus_{c(j)}$	Utilization of the bus of τ_j's contender tasks	-
		$abus_j$	Availability (percentage of cycles) of the bus for τ_j when running with its contenders	-
	instr-uction	n_{total}	Total instruction count of τ_j	-
		I_{mix}	Percentage of instructions of each type for τ_j	-
		n_y	Number of instructions of type y	-
per access (@A)		$t_{@Al}$	Time between accesses $@A_l$ and $@A_{l-1}$	-
		$k_{@Al}^{solo}$	Stack distance of access $@A_l$ in singlecore	-
		$k_{@Al}^{muc}$	Stack distance of access $@A_l$ in multicore	-
		$\Delta t_{@}^{bus}$	Time increment an access suffers in the bus	-
		$a_{@Al}^{h}$	Number of intermediate accesses τ_h generates between $@A_l$ and $@A_{l-1}$	-
		$\Delta k_{@Al}^{h}$	Increment in $@A_l$'s set distance caused by the intermediate accessed generated by τ_h	-
		$\Delta t_{@}^{mem}$	Time increment an access suffers in memory	-
instruction		l_y	Latency of the instructions of type y	-
cache		s	Number of sets in cache	Apply to each cache level, e.g. s_d is the number of dL1 sets
		w	Number of ways in cache	

The main parameters used for our model are those in Table 3.4. We introduce some of them in this section, while others are presented as they are used. In Table 3.4, starting bottom up, we observe that parameters provide cache, per-instruction, per-access information and per-task information. The latter is further broken down into cache, time and instruction information of the task. For cache information of the task we use the convention $metric - cache - task$: *metric* are the initials of the metric described (in capital letters to mean it is a random variable, i.e. histogram); *cache* is cache initial, that for the NGMP is i for the instruction cache, d for the data cache and u for the unified L2 cache. Finally *task* is the task id that is added as subindex. For instance, Kd_j is the stack distance of the accesses to dL1 of task τ_j. When we talk about a cache in general we omit the cache initial, e.g K_j.

We introduce some of Table 3.4's parameters via the example sequence in Figure 3.3. For each access we report its address, accessed cache set and the time in which it happens.

Access number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	@A	@D	@A	@B	@F	@C	@B	@E	@A	@A	@F	@A	@B	@E	@C	@A	@F
accessed set	s0	s1	s0	s0	s2	s0	s0	s1	s0	s0	s2	s0	s0	s1	s0	s0	s2
time	1	4	10	14	16	20	22	25	32	36	40	41	43	50	56	58	60
TS_j	0	0	9	4	0	6	2	21	10	4	24	5	2	25	13	2	20
E_j	∞	∞	1	0	∞	1	0	5	1	0	5	1	0	5	1	0	5
K_j	∞	∞	0	∞	∞	∞	1	∞	2	0	0	0	1	0	2	2	0

Figure 3.3: Access Sequence *Seq1* generated by a task τ_j

- *Time between accesses to the same set* (TS) captures the time between accesses targeting the same set. For instance, accesses #2 and #8 to addresses $@D$ and $@E$ respectively, are consecutive accesses to set 1 (set_1). They occur in cycles 4 and 25 respectively, which results in a time between accesses to the same set of 21 cycles.

- *Set distance* (E) for a given set set_i is the number of other sets (different than set_i but not necessarily unique) accessed since the last time set_i was accessed. For instance, access #2 in *Seq1* accesses set_1, which is accessed again by access #8. In between there are 5 accesses to sets different than set_1, hence set distance equals 5 for #8.

- *Stack distance* (K). The stack distance of an access $@A_l$ is defined as the number of unique (i.e. non-repeated) addresses mapped to the same set where $@A_l$ is mapped and that are accessed between $@A_l$ and the previous access to it, i.e. $@A_{l-1}$. Note that stack distance is similar to the concept of reuse distance, though the latter does not break down accesses per set. For instance, in *Seq1* the accesses to set_0 are ($AABCBAAABC$) with stack distances ($\infty 0 \infty \infty 120012$) respectively. The stack distance of a task τ_j (K_j) is built from the stack distances of its accesses.

It is worth noting that common eviction cache policies such as LRU have the stack property [13], which determines whether a given address is still in cache. For LRU, the focus of this thesis, each set in a cache can be seen as an LRU stack with lines sorted based on their last access cycle. The first line of the LRU stack is the Most Recently Used (MRU) and the last is the LRU. Interestingly, i) the position of a line in the LRU stack defines its stack distance; ii) those accesses with a stack distance smaller than the number of cache ways (w) result in a hit and vice versa; and iii) the sensitivity of the access of τ_j to be evicted from cache by accesses of a contending task τ_h depends on its stack distance: the higher it is, the higher its sensitivity.

Chapter 4

Execution Profile

The execution profile is the characterization of a task in form of statistic values and stored in a file. The time modeling process collects relevant data that represents how the application uses hardware resources in isolation. The result of this data collection process is an EP per application, that will be used by the contention model to obtain predictions on Δ_t.

4.1 Profile Generation

The process of generating the execution profile starts by instrumenting a VM so we recover for every emulated instruction data such as the opcode, program counter and data address for the memory operations. This instrumentation is usually already available in some form of application programming interface (API) that enables access to that kind of information. This extracted information or raw data is then processed by two different components:

- Cache simulator: The cache simulator will pick the program counter addresses and data reference addresses and perform a cache simulation in order to collect data such as cache hit rates, stack distance or set distance.
- Instruction Processing Module: An instruction processing module is in charge of mapping the instructions to their respective instruction types and computing the instruction mix values.

While the overall process is slow, since the instrumentation adds some overhead to the execution and it takes some time to process all the data, this process is only done once per application release.

4.2 Profile Information

The EP generation deploys a high-level cache simulator and some extra modules that process the information coming from the instrumentation of the VM. As a result we obtain an EP that is the input for contention modeling. Table 4.1 summarizes these contents and provides a notations for them.

Table 4.1: Specific information in the Execution Profile and Notation

Hi_j, Hd_j, Hu_j	cache hit rates (miss rates derived as $(1 - Hx_j)$)
TSu_j	Time between accesses going to the same set in uL2
Ku_j	uL2 stack distance of τ_i (including all data accesses)
Eu_j	uL2 set distance of τ_i (including all data accesses)
I_{mix}	Percentage of instructions of each type
n_{total},n_y	Total and per-type instruction count
l_y	Nominal back-end latency per instruction type
et_j^{solo}	τ_j's execution time in isolation

- ***Instruction count***: For each task we keep the total number of instructions executed n_{total}.
- ***Per-type instruction count***: We keep the number of instructions of each type (n_y) in the task. This can be obtained from the opcode of each instruction. It follows that $\sum_{y \in \mathcal{Y}} n_y = n_{total}$, where $\mathcal{Y}$ is the set of all instruction types.
- ***Instruction mix0***. I_{mix} provides the distribution of instructions across types, i.e. n_y/n_{total} for each type y.
- ***Per-type instruction latencies***. It provides information about the latency of each different type of instruction (l_y). This information can be derived by benchmarking [9] or can be found in the user manuals provided by the chip vendor. The information provided covers the core latency of operations and the latency of the local caches, global caches and the main memory for load/store operations. We differentiate the following instruction types ($\mathcal{Y}$) since they are common in several RISC architectures: integer short latency (e.g., `add`, `cmp`), integer long latency (e.g., `idiv`, `imult`), control (e.g., `bne`), floating point short latency (e.g., `fpadd`, `fpmult`), floating point long latency (e.g., `fpdiv` and `fpsqrt`) and memory operations (e.g., `ld` and `st`). Some of these types can be further divided. For instance, the floating point long latency type can be split into divisions (`fpdiv`) and square roots (`fpsqrt`) since the execution time of these two instruction types can be quite different.

It is noted that each instruction instance may suffer variability in its execution time due to two factors. First, *input-data dependence* that occurs when instructions such as floating-point division take variable latency depending on the particular values (input-data) operated. And second, *pipeline state* dependence: in this case, a given instruction may have variable latency depending on its predecessor instructions. In our aim to model the worst-case we handle these sources of jitter by assuming as the latency for every type an upperbound to those latencies. This provides an upperbound to the execution of the instruction. This incurs relatively low inaccuracy while keeping the model simple and fast. For the NGMP, which has mostly a stall-free pipeline, so removing pipeline-state dependences, Table 4.2 shows the latency we assume in our model for every instruction type.

Table 4.2: Operation types in the NGMP and their assumed latencies

operation type	jitter	min-max latency	Assumed latency
int. short latency	NO	1	1
int. long latency	YES	1-35	35
control	NO	1	1
fp. short latency	NO	4	4
fp. long latency	YES	16-25	25

- ***Local caches information***: For the local caches, our cache simulator provides the hit rate. In particular for each τ_j we keep Hi_j and Hd_j.
- ***Global caches information***: As for the local caches, we record information on the hit rate of the application for the global caches, i.e. Hu_j for the NGMP.
- ***Inter-access latency***: For every core instruction executed we have its latency l_y. For instruction and cache accesses the cache simulator is used to determine whether they hit/miss in the different cache levels, for which we have an associated latency. With this information, for every two accesses to uL2 we can predict the execution time of the instructions between them, and hence we can derive the time between consecutive accesses. The histogram (TSu_j) is derived by counting how many times each latency occurs between two consecutive uL2 accesses to the same set.
- Ku_j ***and*** Eu_j: From the cache simulator it is straightforward to derive *stack distance* and *set distance* since we have the memory operations accessing uL2 and the set they access.

- ***Solo performance*** . In our environment, software suppliers are provided with a virtualized environment (e.g. for the SPARC based NGMP in our case) that runs on a host platform (e.g. x86) where et^{solo} – that is the first addend in Equation 3.3 – cannot be derived. We derive et^{solo} by applying a simple approach in which for each instruction we add its front-end (of the pipeline) latency and its back-end latency.

$$et_j^{solo} = \sum_{y \in \mathcal{Y}} [n_y \times (fend(y) + bend(y))] \tag{4.1}$$

Instruction's front-end latency depend on whether they hit or miss in iL1 and uL2. For each of these scenarios their associated latency is different. Whether an instruction hits/misses in cache is provided by the cache simulator. For core operations, such as `add` or `mult`, l_y gives an estimate of their execution time in the back-end. For memory operations such as `load` or `store`, their back-end latency also depends on whether they hit or miss in dL1 and uL2.

Chapter 5

Cache Contention Model

The main objective of the cache contention model is to derive the amount of data in the shared cache that will be evicted due to access of tasks' contenders, or in other words, the increase in the miss rate due to contention.

The process of modeling the contention is based on Monte Carlo experiments. Each instruction that accesses the cache is simulated for all the different statistic figures that describe those accesses[1]. This means that the model iterates for a chosen number of instructions, modeling one by one. At first the instruction type is defined by performing a realization on I_mix. Based on the instruction type y extracted, the model obtains its back-end latency l_y.

The targeted object to analyze are the uL2 hits which are the only ones that could become a miss under a multicore environment, iL1 and dL1 hits are unaffected because L1 caches are private and non-inclusive, and uL2 misses will still miss in contention because our tasks do not share data. This model does not keep track of the execution history, meaning that each uL2 access $@A_l$ of the task under analysis τ_j will be modeled independently from the others.

L2 cache misses can be described by its stack distance value, a stack distance value greater than the number of ways of a cache will always be a miss, and the other way around, a stack distance with a value smaller than the number of ways, will be a hit. In that way the model will compute the increase in stack distance that an access $@A_l$ suffers due to access injection

[1] Our model does not need to simulate the entire number of instructions of a given task, instead a fixed subset number of instructions, fit for a tight timing simulation, can be simulated through the Monte Carlo experiments, and then data can be extrapolated linearly to the real number of instructions. This enables the model to keep its light-weight/low-overhead capabilities.

of contenders between the previous access to the same data $@A_l - 1$. To summarize, if the stack distance in isolation of $@A_l$ ($k^{solo}_{@A_l}$) is smaller than the number of uL2 ways (w_{uL2}) but its multicore stack distance ($k^{muc}_{@A_l}$) exceeds the number of ways, we will account that access to be increasing the total miss count (Δm^{l2}_j). We define this in equation 5.1.

$$(k^{solo}_{@A_l} \leq w_{uL2}) \quad \text{and} \quad (k^{muc}_{@A_l} > w_{uL2}) \tag{5.1}$$

Deriving ($k^{muc}_{@A_l}$) is the main objective of this model. To attack this problem, we will first determine how long is the vulnerability window of an access to cache (i.e. The time between two consecutive accesses to the same address $@A$), with this value it is possible to determine then how many lines the contenders inject in this period of time. Two considerations have to be taken into account, a) contenders have to fetch lines that go to the same set of $@A$, b) all this accesses have to be unique, repeated accesses will hit on cache and won't be increasing the stack distance by occupying other ways.

First of all, we determine the window of vulnerability, the time in which an access of the task under analysis is stored in the cache and until its LRU value is reseted ($t^{solo}_{@A_l}$). For a given access $@A_l$ we perform a realization on its stack distance ($k^{solo}_{@A_l} = rand(Ku_j)$), if the stack distance in isolation is greater than the number of ways, this access was a miss in isolation, thus we discard it, otherwise we keep this value as the number of times the task under analysis τ_j visits the set of $@A_l$ until it fetches $@A_l$ itself. Now we just need to know how much time it takes for τ_j to produce an access to uL2, if we multiply how many times we access a set in cache since access $@A_{l-1}$ to $@A_l$ by the time it takes to send this accesses (tsu_j) we get the total amount of time in which the access $@A_l$ is vulnerable to have its stack distance increased. This timing measure is derived from a realization on the time between accesses to the same set ($k^{solo}_{@A_l} = rand(Ku_j)$).

$$t^{solo}_{@A_l} = tsu_j \times k^{solo}_{@A_l} \tag{5.2}$$

Now that we have the vulnerability window, its time to compute the number of accesses injected by the contenders ($\tau_h \in c(\tau_j)$) to the same uL2 set. To do so, we extract the time between accesses to the same set, in the same way as in the previous paragraph for the contenders ($tsu_h = rand(TSu_h)$) and divided by the vulnerability window ($t^{solo}_{@A_l}$). The resulting value is the proportion of how many accesses a contender can fit in $@A_j$ time window as described in the first addend in equation 5.3. If $t^{solo}_{@A_l}$ and tsu_h are not multipliers, an extra access may be injected by τ_h due to some extra cycles that are left until τ_j accesses its data. We model this behavior on the second addend of equation 5.3 by computing how

many cycles are left until renewing the LRU stack for $@A_l$ ($t^{solo}_{@Al}$ mod tsu_h) and generating a random number between 1 and tsu_h that will describe if task τ_h was able to inject an access before τ_j.

$$a^h_{@Al} = \left\lfloor \frac{t^{solo}_{@Al}}{tsu_h} \right\rfloor + min\left(\left\lfloor \frac{t^{solo}_{@Al} \mod tsu_h}{rand(1, tsu_h)} \right\rfloor, 1\right) \tag{5.3}$$

Up until now, we have considered that all accesses from the contenders, are interfering on the task under analysis, the truth is that memory mapping as well as the usage of the different sets for different tasks have a very important impact on accesses colliding on the same set. Along with that, accesses that repeat on the same set won't be incrementing the stack distance of our access $@A_j$. We solve the first issue with a set collision distribution model and mitigate the effects of the second one by implementing limits on the increment of stack distance.

5.0.1 Set collision distribution

In order to model the effect of not using the whole ̈set spectrum ̈of the cache, we define the concept of set dispersion, as the probability that contenders' accesses go to the same set where $@A_l$ is mapped. To compute this probability du_h, we first derive the average layout occupancy of the cache, or in other words, the proportion of sets that our tasks uses, taken from the average of the set distance distribution Eu_h and dividing that by the total number of uL2 sets. Therefore the probability of a task τ_h mapping an access to the same set as τ_j is the intersection of both cache utilization percentages (i.e. $P^h_{du} = du_j \times du_h$).

$$a^h_{@Al} = \begin{cases} a^h_{@Al}(as\ in\ Eq.\ 5.3) & : \quad rand(0,1) < P^h_{du} \\ 0 & : \quad rand(0,1) \geq P^h_{du} \end{cases} \tag{5.4}$$

5.0.2 Increment in stack distance

This other effect depicts the situation where, a contender τ_h produces in the time interval of the vulnerability window, 4 accesses ($a^h_{@Al} = 4$) but all this accesses are accesses to the same data (e.g. BBBB, where B is a cache block). In this case we have to account for those accesses not increasing the stack distance in 4 but just in 1 since they will fill the same cache block, just one way. To do so, we perform realizations on the stack distance of the contenders ($rand(Ku_h)$) to obtain how many of those interfering accesses are repeated. We

describe this behavior in equation 5.5 by limiting the amount of accesses that a contender can inject.

$$\Delta k^{h}_{@Al} = min(a^{h}_{@Al}, rand(Ku_{h}) + 1) \tag{5.5}$$

5.0.3 Final Step

Finally, we can account for the true increment of stack distance of $k^{solo}_{@A_l}$ by all the contenders accesses, allowing us to derive the stack distance in the multicore environment ($k^{muc}_{@A_l}$).

$$k^{muc}_{@A_l} = k^{solo}_{@A_l} + \sum_{\tau_h \in c(\tau_j)} \Delta k^{h}_{@Al} \tag{5.6}$$

Once this is computed for all the instructions or an arbitrary amount of instructions them, and according to the condition stated in Equation 5.1, the amount of hits that become misses Δm^{uL2}_{j} can be derived. By applying the penalty in cycles for each miss we can derive the increase in terms of cycles that task τ_j suffers due contention as stated in Equation 5.7.

$$\Delta t^{uL2}_{j} = \Delta m^{uL2}_{j} \times (l_{uL2miss} - l_{uL2hit}) \tag{5.7}$$

5.1 Example

Let's explain the whole process through an example of a 1 access analysis.

Setup:

- 4 way cache
- 2 Contenders (τ_h, τ_k)

Process:

1. *Window of vulnerability*: Span of time between an access to address under analysis $@A_j^i$ and the next hit-access to $@A_j^{i+1}$.

   - $\tau_j = \begin{cases} k_{@A_l}^{solo} = 2 \\ tsu_j = 20\ cycles \\ t_{@A_l}^{solo} = tsu_j \times k_{@A_l}^{solo} = 20 \times (2\,(+1)) = 60\ cycles \end{cases}$
   - Depicted situation:

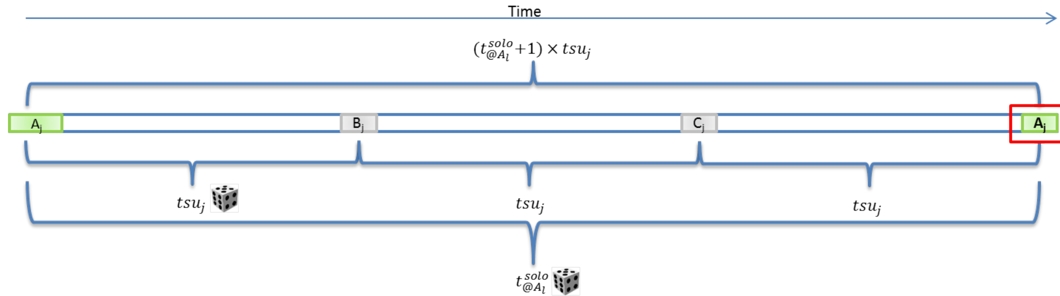


Figure 5.1: Visualization: Window of vulnerability

2. *Contender's accesses injected*: Amount of accesses from contenders that will potentially increase the stack distance of $@A_j$.

   - $\tau_h = \begin{cases} tsu_h = 30\ cycles \\ a_{@Al}^{h} = \left\lfloor \frac{t_{@Al}^{solo}}{tsu_h} \right\rfloor + min\left(\left\lfloor \frac{t_{@Al}^{solo} \bmod tsu_h}{rand(1,tsu_h)} \right\rfloor, 1\right) = \left\lfloor \frac{60}{30} \right\rfloor + min\left(\left\lfloor \frac{0}{15} \right\rfloor, 1\right) = 2 \end{cases}$
   - $\tau_k = \begin{cases} tsu_k = 25\ cycles \\ a_{@Al}^{k} = \left\lfloor \frac{t_{@Al}^{solo}}{tsu_k} \right\rfloor + min\left(\left\lfloor \frac{t_{@Al}^{solo} \bmod tsu_k}{rand(1,tsu_k)} \right\rfloor, 1\right) = \left\lfloor \frac{60}{25} \right\rfloor + min\left(\left\lfloor \frac{10}{8} \right\rfloor, 1\right) = 3 \end{cases}$
   - Depicted situation:

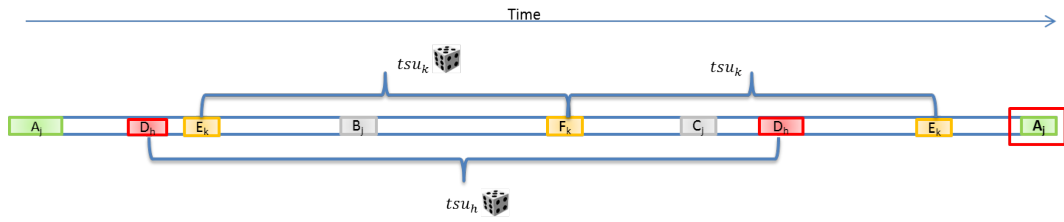


Figure 5.2: Visualization: Contenders' injection

3. *Set collision*: Discard the accesses from the contenders that do not collide in the same set as $@A_j$.

   - $\tau_j = \left\{ du_j = 60\% \right.$

   - $\tau_h = \begin{cases} du_h = 40\% \\ P^h_{du} = du_j \times du_h = 0.6 \times 0.4 = 0.24 \\ rand(0,1) = 0.1 < 0.24 \Rightarrow a^h_{@Al} = a^h_{@Al} \end{cases}$

   - $\tau_k = \begin{cases} du_k = 90\% \\ P^k_{du} = du_j \times du_k = 0.6 \times 0.9 = 0.54 \\ rand(0,1) = 0.75 \geq 0.54 \Rightarrow a^k_{@Al} = 0 \end{cases}$

   - Depicted situation:

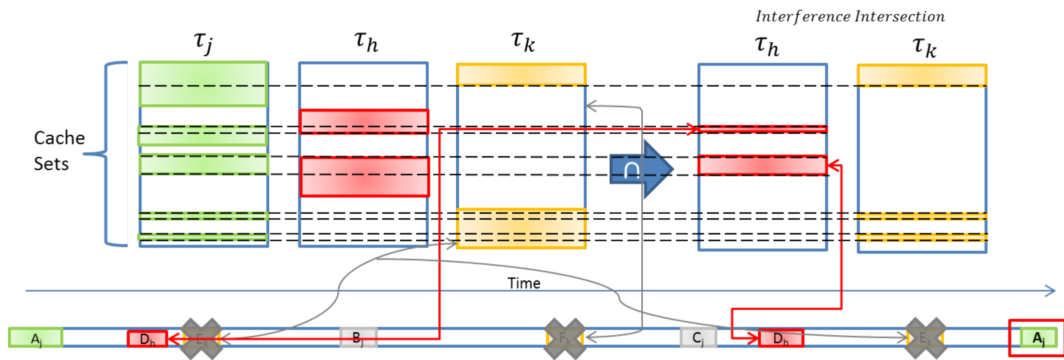


Figure 5.3: Visualization: Contenders' collision

4. *Repeated contenders' accesses*: Discard repeated accesses to the same set since those are not increasing LRU stack distance.

   - $\tau_h = \begin{cases} rand(Ku_h) = 0 \\ \Delta k^h_{@Al} = min(a^h_{@Al}, rand(Ku_h) + 1) = min(2, 0 + 1) = 1 \end{cases}$

   - Depicted situation:

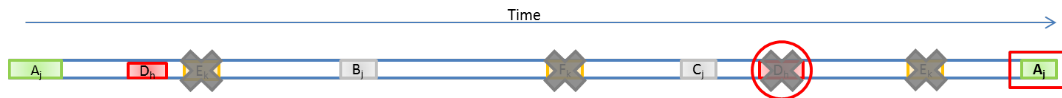


Figure 5.4: Visualization: Discarding repeated accesses

After all the process we see how task τ_k was not able to inject any access into the same set of A_j and thus was not able to increase the stack distance of that access. However task τ_h had two accesses to the same set, but both accesses were to the same block, hence we have to discard one since it is not increasing the LRU stack distance. In the end, the access $@A_j$ increased it's stack distance from 2 to 3, meaning that it is able to remain in the LRU algorithm and not be evicted by the interference.

Chapter 6

Bus and Memory Contention Model

Under the targeted system of the NGMP we assume that bus accesses are not split, meaning that the bus is not relinquished until the access is served either by the L2 cache or the main memory, this implies that one access to the bus contains the delay for the L2 cache and the main memory thus allowing us to simplify the model since it will inherently include the contention suffered in memory.

This model will estimate the contention (Δt_j^{BUS}) in cycles faced by task τ_j on the bus (and memory). In order to do so we derive in the following order this four measures:

1. $etbus_j^{uL2}$: The Execution Time in Bus will describe how many cycles our task τ_j spends on the bus. This value will already factor in the impact of the uL2 cache interference analyzed previously in the cache contention model (CCM).

2. $ubus_{c(j)}$: The Utilization of the Bus by the contenders relative to their execution time.

3. $abus_j$: Availability of the bus, or probability of finding the bus free of contenders.

4. Δt_j^{BUS}.

$etbus_j^{uL2}$ is obtained by adding the time spent by τ_j serving uL2 cache hits and misses in isolation and factoring in the estimates of increase in misses from the CCM. The following equation defines the time spent in bus:

$$\begin{aligned} etbus_j^{uL2} &= ((n_{uL2h}^{solo} - \Delta m_j^{uL2}) \times l_{uL2hit}) + ((n_{uL2m}^{solo} + \Delta m_j^{uL2}) \times l_{uL2miss}) = \\ &= (n_{uL2h}^{solo} \times l_{uL2hit}) + (n_{uL2m}^{solo} \times l_{uL2miss}) + \left(\Delta m_j^{uL2} \times (l_{uL2miss} - l_{uL2hit})\right) \end{aligned} \tag{6.1}$$

In equation 6.1, n_{uL2h}^{solo} and n_{uL2m}^{solo}, correspond respectively to the number of uL2 hits and misses in isolation, and Δm_j^{uL2} to the hits in isolation that become miss due to intertask interference as derived by the CCM. We now use the total execution time derived from the execution time in isolation and adding the number of increased misses multiplied by a penalty for each miss, this is described in equation 6.2.

$$et_j^{uL2} = et_j^{solo} + \Delta m_j^{uL2} \times (l_{uL2miss} - l_{uL2hit}) \tag{6.2}$$

Now if we divide the time spent in bus by the total execution time, we obtain the utilization of the bus for a given task in isolation taking into account a possible cache contention, $ubus_j^{uL2}$.

$$ubus_j^{uL2} = \frac{etbus_j^{uL2}}{et_j^{uL2}} \tag{6.3}$$

Next we compute the contention of all the contenders, this can be done by adding together the utilization factors of the tasks involved, of course we have to account for the case of utilization surpassing the 100% mark, check section 6.1(a) for further explanation.

$$ubus_{c(j)} = \sum_{\tau_h \in c(\tau_j)} ubus_h^{uL2} \tag{6.4}$$

With this values we can now compute the availability of the bus for the core under analysis. This information is derived from the complement of the proportion of cycles that the contenders occupy and the total window of occupied bus, this last factor is computed from the utilization of the contenders plus the utilization of the core under analysis. This step makes a strong assumption on the tasks bus utilization, check section 6.1(b) for a further explanation.

$$abus_j = 1 - \frac{ubus_{c(j)}}{1 + ubus_{c(j)}} \tag{6.5}$$

Finally, the inverse of the availability will provide how often τ_j will have to retry an access to the bus in order to get the hardware resource. Then this value is multiplied by the total time spent in the bus $etbus_j^{uL2}$ to obtain the increase in time in the bus due to contention, Δt_j^{BUS}.

$$\Delta t_j^{BUS} = \left(\frac{1}{abus_j}\right) \times etbus_j^{uL2} \quad (6.6)$$

6.1 Considerations

There are two main consideration taken in this model as practical assumptions:

- a) **Joint utilization over 100%**: When adding bus utilization together we can find the scenario where $ubus_{c(j)}$ is greater than 100%. For instance, let's assume τ_j has two contenders, τ_h and τ_m, each with bus utilization of 60%. This results in a bus utilization of $ubus_{c(j)} = 1.2$. We will consider that contention will obviously increase the time window since total bus utilization of τ_j and its contenders cannot exceed 100%.
- b) **Circular dependence**: The time window increases by the true contention that the other tasks cause on τ_j. However, the actual impact of contention on the bus is the result of the model. For instance, recalling the previous example, the impact of τ_h and τ_m bus accesses will increase τ_j execution time, thus increasing bus availability (same number of accesses over a larger time window). However, such increased bus availability is already needed to compute the time window, thus creating a circular dependence.
  To break this dependence, we upper-bound contention impact in the time window with the total utilization of the other tasks. See the right addend of Equation 6.5, where the time window available is 1 (available utilization in isolation) plus the time that other tasks access the bus ($ubus_{c(j)}$). Hence, the actual bus utilization is approximated by dividing the utilization of the other tasks by the total time window. τ_j finds the bus available ($abus_j$) when it is not being used by others.

6.2 Example

Let's assume a simple example to completely figure out how the bus contention model (BCM) works:

$$\begin{cases} 1 & l_{uL2hit} \\ 10 & l_{uL2miss} \end{cases} \quad \tau_j = \begin{cases} 1900 & n^{solo}_{uL2h} & \textit{L2 hits} \\ 100 & n^{solo}_{uL2m} & \textit{L2 misses} \\ 300 & \Delta m^{uL2}_j & \textit{L2 miss increase} \\ 10000 & et^{solo}_j & \textit{isolation cycles} \end{cases} \quad \tau_h = \begin{cases} 1000 & n^{solo}_{uL2h} \\ 150 & n^{solo}_{uL2m} \\ 150 & \Delta m^{uL2}_h \\ 5000 & et^{solo}_h \end{cases}$$

1. *Time in bus*:

   - τ_j: $etbus^{uL2}_j = (n^{solo}_{uL2h} \times l_{uL2hit}) + (n^{solo}_{uL2m} \times l_{uL2miss}) + \left(\Delta m^{uL2}_j \times (l_{uL2miss} - l_{uL2hit})\right)$ $= (1900 \times 1) + (100 \times 10) + (300 \times (10 - 1)) = 5600$

   - τ_h: $etbus^{uL2}_j = (n^{solo}_{uL2h} \times l_{uL2hit}) + (n^{solo}_{uL2m} \times l_{uL2miss}) + \left(\Delta m^{uL2}_h \times (l_{uL2miss} - l_{uL2hit})\right)$ $= (1000 \times 1) + (150 \times 10) + (150 \times (10 - 1)) = 3850$

2. *Bus Utilization*:

   - τ_j: $et^{uL2}_j = et^{solo}_j + \Delta m^{uL2}_j \times (l_{uL2miss} - l_{uL2hit}) = 10000 + 300 \times$ $\times (10 - 1) = 12700$ $ubus^{uL2}_j = \frac{etbus^{uL2}_j}{et^{uL2}_j} = \frac{5600}{12700} = 0.44$

   - τ_h: $et^{uL2}_h = et^{solo}_h + \Delta m^{uL2}_h \times (l_{uL2miss} - l_{uL2hit}) = 5000 + 150 \times$ $\times (10 - 1) = 6350$ $ubus^{uL2}_h = \frac{etbus^{uL2}_h}{et^{uL2}_h} = \frac{3850}{6350} = 0.60$

   - $ubus_{c(j)}$: $ubus_{c(j)} = \sum_{\tau_h \in c(\tau_j)} ubus^{uL2}_h = 0.6 = 0.6$

3. *Availability*: Availability of the bus, or probability of finding the bus free of contenders.

   - τ_j: $abus_j = 1 - \frac{ubus_{c(j)}}{1 + ubus_{c(j)}} = 1 - \frac{0.6}{1 + 0.6} = 0.625$

4. *Increased Cycles*:

   - τ_j: $\Delta t^{BUS}_j = \left(\frac{1}{abus_j}\right) \times etbus^{uL2}_j = \left(\frac{1}{0.625}\right) \times 5600 = 8960$

Chapter 7

Assumptions & Simplifications

Our model balances accuracy in its prediction and the execution time overhead to run it. Results presented in Section 8 show that our model achieves a good balance between both. However in this section we detail the variety of trade-off decisions that we had to consider.

7.1 General approach

The most remarkable assumption in our models is that we take frequencies observed when characterizing applications as probabilities. However, events such as cache hits/misses do not have a randomized nature as it would be required to attach probabilities to their occurrence. For the sake of simplicity we make this assumption for any process to limit the complexity of the cache model.

7.2 Core model

In general our model does not work with the temporal distribution of events. For instance, we assume that instructions of each type are equidistant in the code. Despite we have histograms we do not consider how instructions are distributed over time. For computing processor core time, we over-approximate the execution time of each instruction in the core instead of tracking pipeline and data dependences, that is, we assume the longest latency of a jittery instruction, e.g. for the NGMP we assume that `fplong`, i.e. fp long-latency instruction we assume, always takes 25 cycles when in reality it can take either 16 or 25

cycles depending on the input values operated. This heavily simplifies the processor core time model with low impact on accuracy.

7.3 Cache model

For the cache model, stack distances and set distances are not maintained per set, but we have one stack distance histogram and one set distance histogram for the whole cache. Alternatively, to increase accuracy we could keep information in a per-set basis, but this would add some non-negligible complexity to the model, would increase the amount of information recorded by a factor of s (the number of sets in cache), and would make results dependent on the actual sets (and so memory locations) of the different tasks. Another source of innacuracy is that we average set distance to approximate set dispersion.

7.4 Bus model

In the bus model there are two main sources of inaccuracy. First, when determining the bus availability for a task the time window assumed is upper-bounded. Second and more important, our bus model assumes that bus accesses are blocking, i.e. they stall the processor pipeline. However, the LEON4 cores in the NGMP have a store buffer able to manage a pending store without stalling the pipeline. Since dL1 caches are write-through in the LEON4 core, write operations to uL2 are abundant. Hence, the bus contention effect will not be as linear as assumed, resulting in under-estimations of the model.

As shown in the result section, the overall accuracy of the model is acceptable as execution time estimates to be used during the EDP, while it incurs low execution time overhead.

7.5 Addressable unit

For sake of simplicity we have assumed in our explanations that each access corresponds to a cache line. When the addressable unit is smaller than a cache line, accesses to different addresses can be mapped to the same cache line. This has no impact on our previous formulation. For instance, let us assume the sequence (A, A, B, C, B, A), in which B and C go to the same line. We can simply abstract this sequence as (A, A, B, B, B, A), hence considering that the access to C corresponds to another access to B, so cache stack distances would be (∞ 0 ∞ 0 0 0 1). This allows us applying the formulation presented.

Chapter 8

Evaluation

8.1 Methodology & Experimental Framework

This work will compare the proposed model against a simulator with a determined target platform. This simulator called SoCLib [20] has been validated and on average produces a 3% deviation on the results reported by the real hardware for a specific benchmark suite, the EEMBC automotive benchmarks.

8.1.1 Target platform

As it has been already said, the reference platform for this work is the NGMP [5] depicted in Figure2.1 and described in the following points:

- Core: The NGMP has 4 cores based on the LEON4 core. These are SPARC cores with a seven-stage scalar in-order pipelines. All these cores have an instruction and data first-level caches (iL1,dL1). All the cores are interconnected through an AHB AMBA bus to a shared second-level cache (uL2).
- Floating point: The floating point unit (FPU) of the NGMP core has a double path for floating point (FP) operations. There is one path for divisions and square roots, and another path for the rest of FP operations which is fully pipelined providing a latency of 4 cycles. The pipeline of the divisions and square roots is not pipelined meaning that its latency is inversely proportional to its throughput.

- Bus: Arbitrated by a round-robin policy and with 128-bit width the bus of the NGMP follows the AHB AMBA standard. It has 5 masters (4 cores and I/O master) and 2 slaves (level 2 cache and I/O slave). On the analyzed architecture the bus split feature is not build in yet.

- Memory: The level 1 caches have a size of 16KB and are 4-way associative, giving them 128 sets. These also implement a write-trhough and write no allocate policies with some write buffers. On the next level, the L2 unified cache, there are 2048 sets and a total of 256KB and 4 ways. It implements a copy-back, write allocate policies.

8.2 Benchmarks

The software used to test our model, will come from three different sources:

- *EEMBC automotive*: This suite of benchmarks [16] is well-known in the real-time domain and adjusted to analyze the capabilities of the embedded architectures and tools designed in that environment. A subset of this benchmarks have been chosen, these correspond to the most čache sensiblebenchmarks, or those that have a larger footprint on the cache, and are prone to have data evicted from the level 2 cache. The following are the benchmarks selected: aifirf (AF), aiifft (AT), bitmnp (BI), cacheb (CB), canrdr (CN), idctrn (ID), iirflt (II), puwmod (PU), rspeed (RS).

- *European Space Agency benchmarks*: These are representative benchmarks actually being used by the ESA. The following are the benchmarks selected: On-board Data Processing (OB), and DEBIE (DE). The OB benchmark contains the algorithms

- *Stressing Kernels*: These are benchmarks that focus their activity on certain hardware resources. Using this kernels will allow the model to be stressed. The following benchmarks will be used: l2full (U), l2half (H), l2miss (M), l1miss (L), mixed-8-12-80 (E). The first 3 benchmarks produce accesses to the level 2 cache, filling it, filling only 2 ways or missing always on it. The l1miss benchmark produces misses on the instruction cache, and finally mixed-8-12-80 (E) is a specific mixture of instructions (8% stores, 12% loads, 80% adds),

8.2.1 Workloads

We run four-task workloads, and select eight different workloads for each task under analysis. The contenders selected correspond to the eight workloads than hurt most the performance of the task under analysis. For the first task we use our model to estimate its execution time bound. This first task is either an EEMBC AutoBench benchmark or an ESA benchmark. The other three (contending) tasks in the workload are `l2full`, `l2half`, `l2miss`, `l1miss` or `mixed-8-12-80`. This creates a stressful scenario in which we can fairly assess the accuracy of the predictions our model. Table 8.1 summarizes the workloads used in this thesis.

Table 8.1: Workloads used for evaluation

BENCHMARK	*8 WORKLOADS IN WHICH IT RUNS*
AF	(LLL), (LLH), (LHL), (HLL), (HLH), (HHL), (LHH), (ULL)
AT	(UUU), (UMU), (MUU), (UUM), (MMU), (MUM), (UMM), (MMM)
BI	(UUU), (UMU), (MUU), (UUM), (MMU), (UMM), (MUM), (MMM)
CB	(LUU), (UUU), (ULU), (UUL), (HUU), (MUU), (UHU), (UMU)
CN	(ULU), (UUL), (LUU), (HUU), (UHU), (UUH), (MUL), (ULM)
DE	(MUU), (UMU), (UUU), (UUM), (MMU), (MUM), (UMM), (MMM)
ID	(UUU), (UMU), (MUU), (UUM), (MMU), (UMM), (MUM), (MMM)
II	(LLL), (LLH), (LHL), (HLL), (LHH), (LEL), (HLH), (LLE)
OB	(UUU), (UMU), (MUU), (UUM), (MMU), (MUM), (UMM), (MMM)
PU	(UUU), (UMU), (MUU), (UUM), (LUU), (ULU), (UUL), (UMM)
RS	(LLL), (LLH), (LHL), (HLL), (LEL), (LLE), (ELL), (HLH)

8.3 Metrics

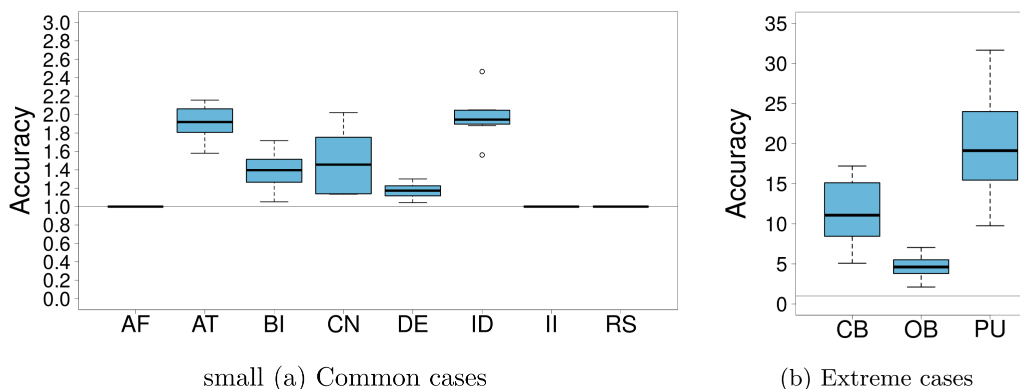
We evaluate the accuracy provided by our model in terms of the increment in the number of L2 misses (Δm_i^{uL2}), bus time ($\Delta t_i^{BUS} + \Delta t_i^{MEM}$) and the overall execution time in multicore (et^{muc}). For each of these three metrics we measure accuracy as $\frac{PredictedValue}{ActualValue}$. Hence the closer to one the better, with values above one showing that the model over-approximates and values below one that the model under-approximates. For each four-task workload, we measure the accuracy in estimating the contention of its first task. As shown in Table 8.1 we create eight workloads for each EEMBC Automotive and the ESA benchmarks. We show the distribution of the accuracy across the eight workloads with a boxplot, thus showing the median, the quantiles, maximum and minimum values and outliers.

8.4 Results

8.4.1 Average-Based Model

For comparison purposes we obtained results for the cache contention model using average values rather than histograms. Our results show that for all the workloads listed in Table 8.1 the average-based model detects almost no contention. This occurs because dividing average values by the number of sets du, as done in Equation 5.4, leads to very low predicted interferences among tasks. This produces contention predictions as low as 0.0034, that is with an inaccuracy of almost 100% ($1 - 0.0034 = 0.9966$).

8.4.2 Cache Contention Model



small (a) Common cases

(b) Extreme cases

Figure 8.1: Accuracy of the cache contention model

As we can see in Figure 8.1(a) Δm_i^{uL2} is accurately estimated for several benchmarks and somehow overestimated for others. The largest deviation are due to the fact that our cache contention model assumes that stack and set distances are homogeneous across sets for the sake of simplicity. However, this is not always the case and, in fact, our `l2full` and `l2half` synthetic kernels are specifically designed to stress all cache sets and half of them, respectively, so that heterogeneous behavior across sets produces large inaccuracies in our model.

The case of the three benchmarks in Figure 8.1(b) is completely different since Δm_i^{uL2} is in the range $[10, 100]$. In this case, benchmarks suffer in the order of dozens extra misses in L2 due to contention, which is negligible for those benchmarks executing millions of instructions. Hence, although our model overestimates Δm_i^{uL2} by a factor of 10-20x, this only represents accounting for around of 1,000 extra L2 misses whose impact in the total execution time is negligible.

It can also be observed that the cache contention model leads to an overestimation of Δm_i^{uL2}. The main reason is the fact that our model assumes that cache accesses are homogeneously distributed in time. However, in reality they typically occur in bursts. Thus, if bursts of all different tasks occur simultaneously, the model is accurate but if, instead, not all of them overlap completely (the common case) real interference is lower than estimated because the number of interfering accesses from other tasks during a burst of the task under analysis is lower than predicted.

8.4.3 Bus Contention Model

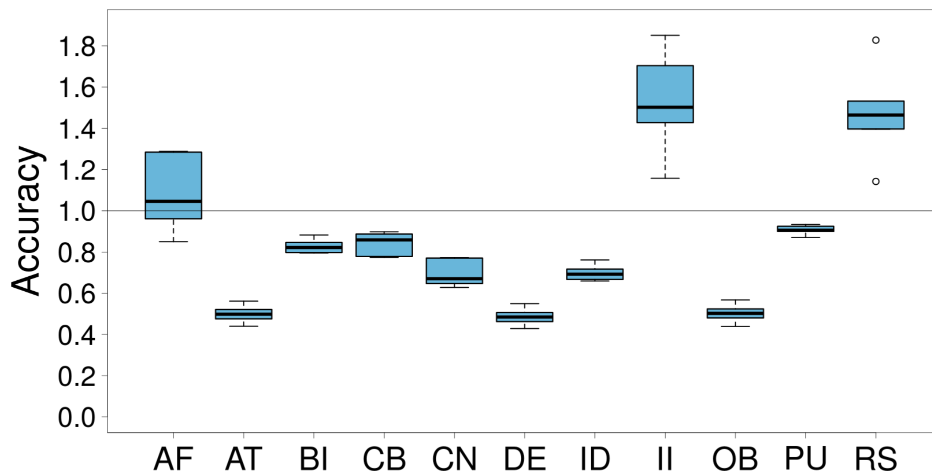


Figure 8.2: Accuracy of the bus contention model

For bus contention, shown in Figure 8.2, we observe a variety of behaviors across benchmarks. Although such contention is somehow overestimated for few benchmarks, it is typically underestimated for most of them. This effect is particularly noticeable for `aiifft`, `debie` and `obdp`. As explained before, the NGMP processor has store buffers that are able to hide part of the latency of stores. However, since stores occur in bursts, whenever they occur in a short period of time they can produce performance degradations much higher than linear. For instance, it has been proven that execution time may grow by a factor of 20x in the NGMP with just 4 cores due to the (bad) interaction of store operations in the different cores [10]. How to better capture this effect in our bus contention model without incurring high overhead is still part of our future work.

8.4.4 Multicore Execution Time

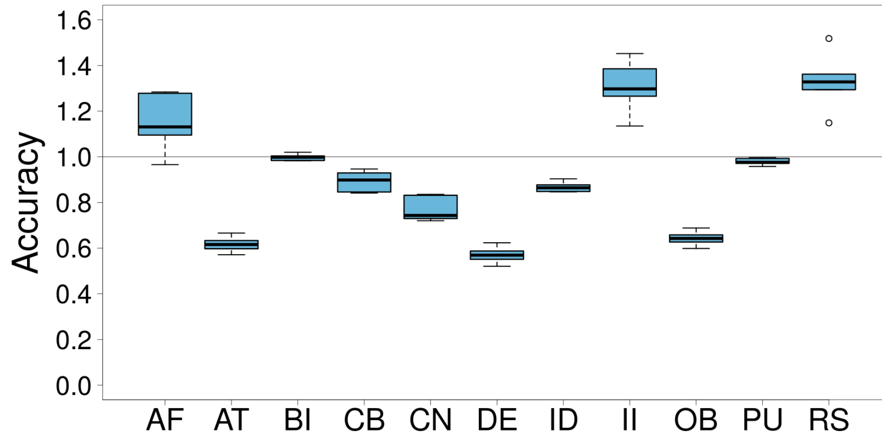


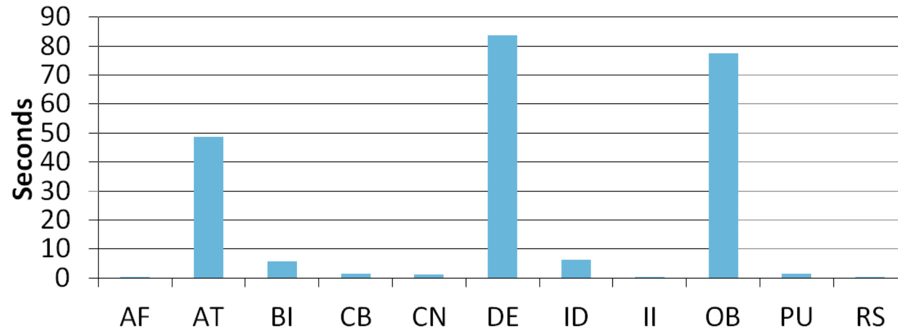
Figure 8.3: Performance accuracy of the complete model. Results have been normalized w.r.t. the actual measurements.

We have also analyzed the accuracy of our performance estimates when considering all contention models and assumptions together. The accuracy in determining et^{solo} is high with predictions in the range [1.02, 1.23] for all benchmarks.

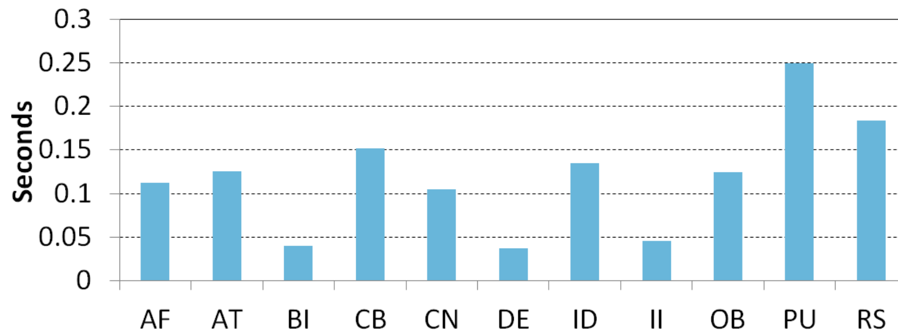
Results are shown in Figure 8.3. We observe that the accuracy of the estimates is mostly dominated by bus contention in the NGMP. The main reason is the fact that the difference between L2 hit and L2 miss latencies is relatively low (9 versus 23 cycles) and only affects Δm_i^{uL2}, whereas the impact of bus contention affects all load misses in L1 and *all store instructions* given that DL1 is write-through. Therefore, inaccuracies due to the effect of the store buffer in terms of bus contention dominate the results.

Overall, our simple analytical model is able to keep performance estimates in the range $[0.6, 1.4]$ w.r.t. the real performance in the NGMP. These are accurate results for execution time estimations for early-design phases. The accuracy of our model allows system designers to really take into account multicore contention in the design choices made (e.g. deciding the scheduling plan).

8.4.5 Performance and Overhead



(a) Time required by the EP generation



(b) Time to perform one estimation of the multicore contention

Figure 8.4: Average overheads of our contention prediction model.

To produce the 'raw data' from the VM as presented in Figure 3.1, we introduce instrumentation instructions to read information about the particular instruction under execution. The number of instructions to add is relatively low: three instructions in our case to read opcode, program counter and data address. From the raw data we run the cache simulator and process instructions to generate the EP. Figure 8.4(a) shows the execution time of this EP generation step, which is run just once per application. If the application has several releases whose resource-usage profile is expected to change, this step is repeated once for each of those releases. The duration of this step depends on the length of the program. For the EEMBC and the real ESA benchmarks, whose execution is in the order of dozens millions of instructions, in the worst case this step takes around 80 seconds. On average EP generation requires around 20 seconds across all benchmarks, which is reasonably short given the low frequency with which this step executes.

Figure 8.4(b) shows the execution time overhead of contention models, once EP information has been produced. The duration of this step is the most important for the feasibility of our approach. This is so because, once the EPs are generated, the system designer needs short turn-around time of the models to be able to evaluate different design choices. We observe that predicting the multicore performance of an application in a workload requires as low as 120ms on average, which enables a vast design space exploration of various system parameters by system designers.

Chapter 9

Related Work

The focus of timing analysis techniques in the literature for EDP has been on single-core architectures. To our knowledge, the work presented in this thesis is the first addressing the challenge of providing timing estimates in EDP for multicores.

Some approaches for single-core architectures work on the assumption that the target processor and/or the corresponding compilation toolchain is available, while others do not. When the target processor is not available, several techniques exist to derive timing estimates that help deciding the hardware platform that best satisfies system requirements [8][22] as well as sizing it. The approach consists in compiling the source code for a given set of potential target ISAs. For each of the ISA there is a parameterizable processor simulator (model) from which timing information is gathered. The model allows changing parameters with high impact on timing such as cache configuration [22]. Then program information (e.g. paths) obtained from the executable and timing from the generic processor model are combined to approximate programs execution time. In our case the target ISA and processor are fixed, so such an approach would not be necessary.

Other approaches do not work at the binary level but at the source code level, or an intermediate representation level, which are available earlier in the design cycle of the system. In some cases, timing is integrated in high level modelling environments such as Matlab/Simulink [6]. The ultimate goal is providing the developer knowledge of the worst-case "as the code is written" [7]. In all cases the focus is on single-core architectures, while our focus is on multicore contention.

In many of the approaches above one of the main challenges lies in deriving a light, yet accurate, timing (cost) model for individual instructions or sequences of them. Some papers [7] assume a WCET-friendly processor design, such as the Java Optimized Processor (JOP). This simplifies the timing model since the processor is predictability aware. Other papers propose methods to derive a timing model from measurements of representative code extracts on the target processor. For instance, authors in [12] work on the concept of C-source-level abstract machine which is calibrated based on measurements to match a target real hardware. In this line, [11] proposes the *timing model code level* that combines measurements and a regression model to perform timing estimates of source code. In this latter work, the timing (cost) model is built, i.e. it is not assumed as an input. In both cases the focus is on constructs that frequently appear on the target programs.

While previous works focus on single-core processors, our focus is on multicore specific aspects. In particular the contention in the access to hardware shared resources. For multicore, it could be possible to run the program under analysis against a set of resource stressing kernels (*rsk*s) which put high load on the shared resources [10, 19]. This would provide a good estimation of the program execution time under heavy (extreme) load conditions. However, it has been shown that this approach leads to inflated execution time estimates, up to 20x bigger than programs' execution time in isolation, which makes it impractical to obtain accurate execution time approximations during EDP.

Previous works show that, while exact bounds are required in LDP, during EDP, instead, approximations to those bound are needed [8][11]. Some accuracy is traded to speed up the estimation process so that engineers can make design space exploration taking into account timing. To our knowledge, no particular figure is reported on accuracy required in EDP. For multicores several works show that the impact of contention can up to 20x for some kernels and up to 5.5x for some EEMBC benchmarks [10]. In this context, we deem the accuracy results obtained by our approach (between 0.6x and 1.4x) as sufficiently precise.

Chapter 10

Conclusions & Future Work

In the Introductory Chapter of this thesis we presented some relevant problems that start emerging in the design of RTES. Among these problems it is the slow-downs a task suffers due to use of virtualization and simulators. In particular when the target (virtualized) platform is a multicore uncertainties arise due to the fact that the timing behavior of a task depends on its co-runners. This increases the complexity – with respect to current practice – to derive timing bounds in early-design stage.

In this thesis we have presented a first approach on how to estimate the impact of multicores in EDP on the NGMP platform. The core of this approach is based on EPs for each application that do not compromise software suppliers' IP and hence can be shared amongst them. This EPs together with the model proposed allow for anyone to derive execution time estimates and impact of the contenders on their developing tasks in a fast manner that allows multiple schedules to be planned. This thesis puts the model to the test showing how the estimates are accurate enough for the EDP and that evaluating an schedule takes less than 0.2 seconds, which makes the model light-weigh in terms of overhead.

This new technique enables RTES developers to analyze and manage with better precision the uncertainties that multicore environments present to the industry. Unlocking this knowledge will allow for cheaper and better designed systems that will benefit the entire user spectrum, from industry to end-user. However it is crucial to maintain this efforts in this fields since there is still many unknowns to be cleared in order provide a smooth transition to multicores to the RTES industry.

10.1 Future Work

During the making of this work some approaches where left aside in order to focus on the completion of the thesis, this bits of improvement that could be added, are detailed as future work:

- *Bus Model*: As denoted in Section 7, the bus model has two main sources of inaccuracy. Both issues should be tackled first, since we have seen how bus contention can have a proportionally large impact in the increase of execution time. However the model still provides acceptable bounds of accuracy.

- *Information per Set*: By gathering information more specifically for each set, instead of gathering information globally, we could capture de non-uniformity of usage across sets.

- *Information per Time*: If we add some kind of information that relates statistics to a precise moment in time, we could locate the different statistic values per regions of the program, that may allow us to model programs with non-uniform behavior.

- *Isolation Accuracy*: Or due the accuracy in isolation is good enough for our predictions to be accurate, an improvement by covering corner cases for the isolation execution time estimation will also have a positive impact on the later predictions since the contention model (CM) has inputs that depend on the results on the simulations and results from isolation.

Chapter 11

Acknowledgements

I could not end this thesis without acknowledging some of the people that made possible this work.

To Francisco J. Cazorla and Jaume Abella, my advisors, for having the patience to deal with specific difficulties and details far-off from their main duties.

To Mikel Fernández and Javier Jalle for collaborating in this research, and helping me get in touch with the main tools used.

To all my colleagues at the CAOS group for their kind help.

To all the BSC team and the Severo Ochoa grants for allowing me to take part in their project.

And as always, thanks to my parents for supporting me no matter my mood.

Chapter 12

Published Work

This work has been published under the following format:

- David Trilla, Javier Jalle, Mikel Fernandez, Jaume Abella, Francisco J. Cazorla, *"Improving Early Design Stage Timing Modeling in Multicore Based Real-Time Systems"*, in proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna (Austria), April 11-14, 2016.

Bibliography

[1] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.

[2] *ARINC Specification 653: Avionics Application Software Standard Standard Interface, Part 1 and 4*, 2012.

[3] J. Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.

[4] AUTOSAR. *Technical Overview V2.0.1*, 2006.

[5] Cobham Gaisler. *NGMP Preliminary Datasheet Version 2.1, May 2013.*

[6] Raimund Kirner et al. Fully automatic worst-case execution time analysis for matlab/simulink models. In *ECRTS*, 2002.

[7] Trevor Harmon et al. Fast, interactive worst-case execution time analysis with back-annotation. *IEEE Trans. Industrial Informatics*, 8(2), 2012.

[8] C. Ferdinand et al. Integration of code-level and system-level timing analysis for early architecture exploration and reliable timing verification. In *ERTS2*, 2010.

[9] G. Fernandez et al. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *DAC*, 2015.

[10] Mikel Fernández et al. Assessing the suitability of the ngmp multi-core processor in the space domain. In *EMSOFT*, 2012.

[11] J. Gustafsson et al. Approximate worst-case execution time analysis for early stage embedded systems development. In *SEUS*, 2009.

[12] R. Kirner and P. Puschner. A simple and efficient fully automatic worst-case execution

time analysis for model-based application development. In *Workshop on Intelligent Solutions in Embedded Systems*, 2003.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2), June 1970.

[14] M. Paolieri et al. *An Analyzable Memory Controller for Hard Real-Time CMPs* . Embedded System Letters (ESL), 2009.

[15] Zheng Pei Wu et al. Worst case analysis of DRAM latency in multi-requestor systems. In *RTSS*, 2013.

[16] Jason Poovey. *Characterization of the EEMBC Benchmark Suite.* North Carolina State University, 2007.

[17] Peter Puschner and Martin Schoeberl. On Composable System Timing, Task Timing, and WCET Analysis. In *WCET Analysis Workshop*, 2008.

[18] P. Puschner et al. Towards Composable Timing for Real-Time Software. In *Workshop on Software Technologies for Future Dependable Distributed Systems*, 2009.

[19] Petar Radojković et al. On the evaluation of the impact of shared resources in multi-threaded cots processors in time-critical environments. *ACM TACO*, 2012.

[20] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.

[21] Michael Tautschnig Raimund Kirner Sven Bünte, Michael Zolda. Improving the confidence in measurement-based timing analysis.

[22] `http://www.absint.com/timingprofiler`. *Timing Profiler.* AbsInt.

[23] R. Wilhelm et al. The worst-case execution time problem: overview of methods and survey of tools. *ACM TECS*, 7(3):1–53, 2008.

[24] Wilhelm R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 7:1–53, May 2008.