Controlling execution time variability using COTS for Safety-critical systems

Jingyi Bin

UNIVERSITÉ PARIS-SUD

ÉCOLE DOCTORALE : Sciences et Technologie de l'Information, des Télécommunications et des Systèmes

Institut d'Electronique Fondamentale (IEF)
Thales Research & Technology - France (TRT)

DISCIPLINE : Physique - Systèmes Embarqués

THÈSE DE DOCTORAT

soutenance le 10/07/2014

par

Jingyi BIN

Controlling Execution Time Variability Using COTS for Safety Critical Systems

Directeur de thèse :	Alain MERIGOT	Professeur (IEF, Université Paris-sud)
Co-encadrant :	Sylvain GIRBAL	Ingénieur (TRT)
Composition du jury :		
Président :	Daniel ETIEMBLE	Professeur (Université Paris-sud)
Rapporteurs :	Laurent PAUTET	Professeur (LTCI, Télécom ParisTech)
	Michel AUGUIN	Directeur de Recherche (Université de Nice)
Examinateurs :	Alain MERIGOT	Professeur (IEF, Université Paris-sud)
	Claire PAGETTI	Chargée de recherche (ONERA)
	Sylvain GIRBAL	Ingénieur (TRT)

I would like to dedicate this thesis to my loving parents BIN Wenjin and XU Zhanchun and my loving husband WANG Weijia. Although my parents were not with me in France during my PhD study, their words of encouragement and push for tenacity kept me moving on. They are always my most trusted supporters. My dear husband was the one who has accompanied me everyday during these three years to share my happiness, anxiousness and depression. I feel so thankful to him for understanding me and helping me when I was losing the confidence. He is the witness throughout the process.

Acknowledgements

I would like to express the deepest appreciation to my PhD adviser– Professor Alain Mérigot who has given me constructive advices and guidances during my PhD study. In addition, he has shown a great patience and warm encouragements when my thesis did not move on smoothly. Without his constant help, I would not have finished my thesis.

I would like to give the most sincere appreciation to my co-adviser Dr. Sylvain Girbal and co-workers Dr. Daniel Gracia Pérez and Dr. Arnaud Grassat in Thales Research & Technology (TRT). All of them has helped me greatly in the technique domain of research and shown me serious, proactive altitude of researchers. Additionally, Sylvain taught me other professional capabilities, like making a simple and efficient slides ppt and a convincing presentation, which are also helpful in the future work.

I also would like to show my gratitude to the head of Embedded Systems Lab in TRT - Philippe BONNOT who has shown so much concern for my PhD progress and helped me a lot to adapt to the company's affaires.

Besides, I would like to thank all the colleagues in Embedded Systems Lab in TRT. Without their help and concern, I would not have happily integrated into the group during these three years.

Last, I would like to thank all the commitee members of my defense: Prof. Daniel ETIEMBLE who accepted our invitation as the committee president, Prof. Michel AUGUIN and Prof. Laurent PAUTET who gave thoughtful and detailed comments as my thesis reviewers, Dr. Claire PAGETTI who provided encouraging and constructive feedback as the examiner. I appreciate all of them for their questions and remarks during my thesis defense.

Abstract

While relying during the last decade on single-core Commercial Off-The-Shelf (COTS) architectures despite their inherent runtime variability, the safety critical industry is now considering a shift to multi-core COTS in order to match the increasing performance requirement.

However, the shift to multi-core COTS worsens the runtime variability issue due to the contention on shared hardware resources. Standard techniques to handle this variability such as resource over-provisioning cannot be applied to multi-cores as additional safety margins will offset most if not all the multi-core performance gains. A possible solution would be to capture the behavior of potential contention mechanisms on shared hardware resources relatively to each application co-running on the system. However, the features on contention mechanisms are usually very poorly documented.

In this thesis, we introduce measurement techniques based on a set of dedicated stressing benchmarks and architecture hardware monitors to characterize (1) the architecture, by identifying the shared hardware resources and revealing their associated contention mechanisms. (2) the applications, by learning how they behave relatively to shared resources. Based on such information, we propose a technique to estimate the WCET of an application in a pre-determined co-running context by simulating the worst case contention on shared resources produced by the application's co-runners.

Contents

List of Figures

List of Tables

Listings

Introduction

In recent years, most of the research in computer architectures has concentrated on delivering average-case performance, but high performance is not the only criterion for some type of applications. For example, hard real-time systems have to satisfy stringent timing constraints, which needs a reliable guarantee based on the **Worst Case Execution Time (WCET)** to make sure that required deadlines can be respected. Missing deadlines may cause huge damages and loss of lives in **safety-critical systems**, like the avionic, healthcare and so forth. As a consequence, the **performance predictability** is more required than high performance in the safety-critical domain.

However, considering the increasing processing performance requirement in safety-critical industries, the next generation architectures have to guarantee the predictability while providing the sufficient average performance. Based on this point, multi-core Commercial Off-The-Shelf (COTS) architectures are considered as an appropriate candidate to provide a long-term answer to the increasing performance demand with an acceptable power consumption and weight. In addition, compared to in-house solutions which aim at proposing new predictable architectures, COTS solutions have lower Non-Recurring-Engineering (NRE) costs and shorter Time-To-Market (TTM).

Despite above advantages of multi-core COTS architectures, they have a critical issue to the safety-critical domain - reduced predictability due to the **contention** of co-running applications on **shared hardware resources** within multiple cores, like the interconnect bus and the memory. There are thus many related on-going researches concentrating on estimating the performance variability by analysing the interference on concurrent shared resources. Since multi-core COTS are inherently very complex with some undisclosed contention mechanisms and the behavior of applications on shared resources is also a gray- or black-box to users, the variability estimation is very difficult in such circumstance.

In the thesis, we proposed two approaches to overcome the difficulty for the overall objective - estimate the execution time variability of co-running safety-critical applications on a multi-core COTS architecture. We first presented a methodology characterizing the underlying architecture and applications to learn

undisclosed hardware features, especially the contention mechanisms on shared resources, and to master how an application behaves relatively to shared resources. Based on the characterized information, we then proposed an alternative technique to estimate the WCET of an application when it ran with a set of pre-determined applications by simulating the worst case contention on shared resources produced by co-runners. Compared to the state of the art using the measurement-based approach, our approach can provide a tighter estimation of the execution time upper bound.

To clearly present the proposed approaches, the remaining thesis is organised as below:

- Part A - Context: Present the research context, the state of the art, our target architecture platform with the corresponding software environment and our contributions.
- Part B - Quantifying Runtime Variability: Demonstrate the variability that an application may experience while co-running with others due to the contention on shared hardware resources in multi-cores. To better understand the organized experiments, we also present our used applications and stressing benchmarks.
- Part C - Architecture and Application Characterization: Present first the characterization methodology, and second the measurement techniques, and third experimental setup with all the experiment designs and implementations, and fourthly the experimental results which are splitted into the architecture section and the application section.
- Part D - Alternative Technique of WCET: Present first the methodology, and second the measurement techniques, and third the two detailed estimation methods with experimental results.
- Part E - Conclusion: Conclude the achievements throughout the thesis and propose some constructive future work based on the estimation results.

Part A

Context

Chapter A.1

Context

A.1.1 Safety-critical Domain

Safety-critical domains [6] such as the avionic, automotive, space, healthcare or robotic industry are characterized by stringent hard **real-time** constraints which are usually defined as a set of **deadlines** to respect. To ensure the correct functionality of a safety-critical application, we should make sure that such application can finish its execution before a required dealine. Missing a single deadline may have some catastrophic consequence (i.e. the air crash and the explosion in a nuclear station) on the user or the environment and should be avoided at all cost. Therefore, the **time predictability** is a major concern instead of delivering high average performance. In order to ensure that an critical application can finish execution before its required deadline, we usually determine its **Worst Case Execution Time(WCET)** which has to be guaranteed shorter than this deadline.

A.1.2 Worst Case Execution Time (WCET)

A real-time system consists of a number of tasks and each task typically shows a certain variation of execution times depending on the input data or different behavior of the environment. The WCET of a task is defined as its longest execution time. In most cases, it is quite impossible to exhaustively explore all possible execution times and thereby determine the exact WCET.

Today, there are some practical methods to compute the WCET. In single-core architectures, this WCET computation usually relies on analysis tools based on static program analysis tools [38, 27], detailed hardware model, as well as measurement techniques through execution or simulation [15]. However, these analysis techniques and tools are not currently able to provide an exact computa-

tion of the WCET, especially for multi-cores, only delivering an estimated upper bound, introducing some safety margins as depicted in Figure A.1.1.

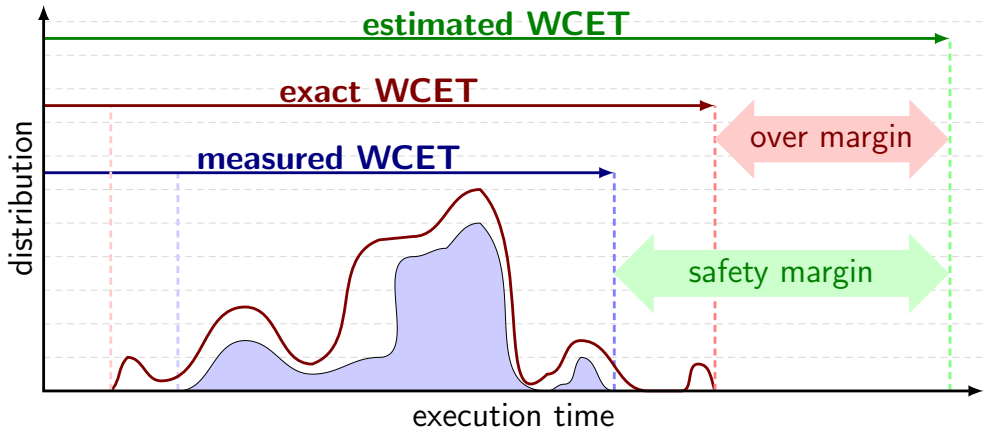


Figure A.1.1: *Estimation of the Worst-Case Execution Time, and the over-estimation problem*

In Figure A.1.1, the red curve is the real execution time distribution, and the blue one represents measured execution time distribution observed during experiments. As we explained above that it is quite impossible to exhaustively explore all possible execution times through experiments, the real distribution is thereby an envelope of the measured one. That's why the measured WCET is shorter than the real one in Figure A.1.1. In order to make a sure over-bound the real WCET, we add a **safety margin** to the measured WCET to get estimated WCET which is used to compare with a pre-determined deadline.

A.1.3 Evolution of Architectures

The evolution of architectures is mainly targeting the consumer electronic market that represents 97% of the overall market. As this consumer electronic market is mostly driven by best-effort performances, the design complexity has increased by integrating more and more high-performance techniques into architectures. Figure A.1.2 shows the evolution of architectures and how the average time and the WCET vary according to such evolution.

A cache is a smaller, faster memory physically existing between the CPU and the memory to temporarily store the copy of data so that future request for that data can be served faster. Most modern CPUs have multiple independent levels of cache with small fast caches backed up by larger, slower caches to deal with a fundamental tradeoff between cache latency and hit rate.

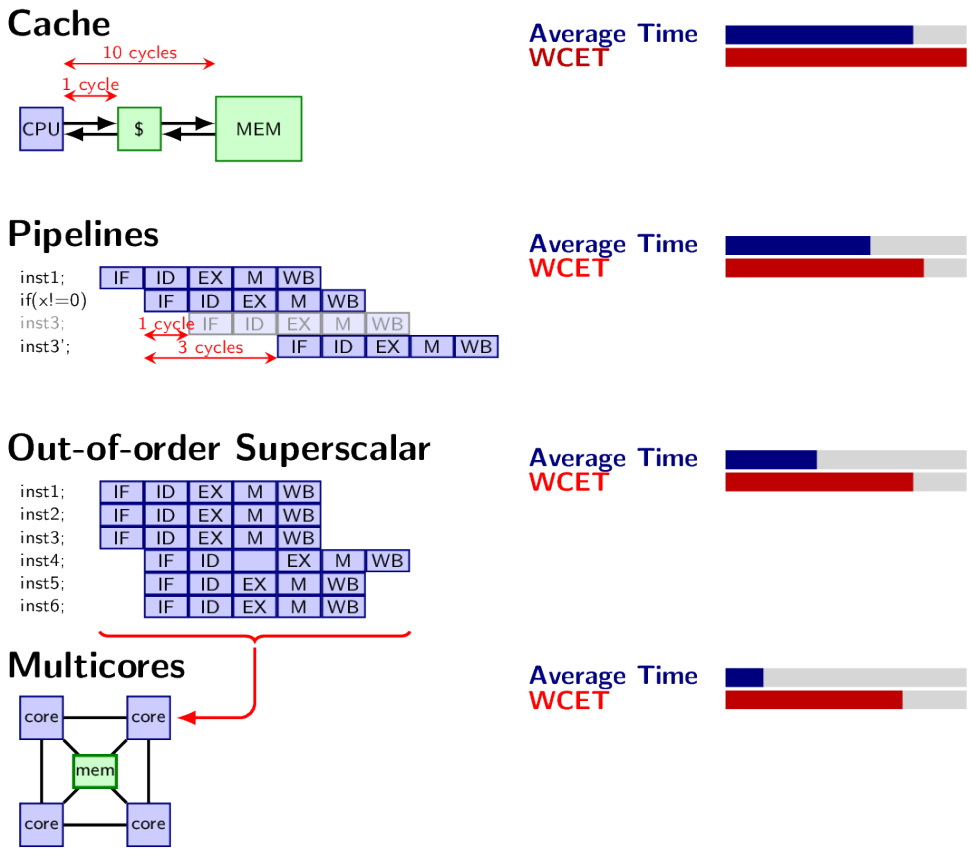


Figure A.1.2: *Evolution of architecture and corresponding average time and WCET*

A pipeline is a concept inspired from assembly line where a set of processing elements are connected in series so that they can be arranged in parallel. Pipelining doesn't decrease the processing time of a single instruction, but it can apply the parallelism among a series of instructions to increase the throughput of CPUs. Based on pipelining, a superscalar pipeline further increases the CPU throughput by executing more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. In addition, an out-of-order execution can make use of instruction cycles which would be wasted because of some types of operations with costly delay. For example, a processor can avoid being idle while data read from memory for the next instruction.

From the early 2000s, considering higher-clock-speeds-produced exponential increasing thermal and power dissipation along with design complexity increases in single-cores, processor design trends shifted to multi-cores. The motivation of multi-cores design is the parallelism of processors that can address the issue

of power while maintaining performance where higher data throughput may be achieved with lower voltage and frequency.

Thanks to above techniques, the average time decreased step by step in the blue right column of Figure A.1.2. Compared to the average time, the WCET time decreased much slower due to the **runtime variability** source brought by above design techniques, like cache misses, structural or data hazards during pipeline executions and the contention on shared hardware resources in multi-cores and so forth.

A.1.4 Challenge of Using Multi-cores for Safety-critical Applications

Figure A.1.3 shows the roadmap of code size in safety-critical industries where the code size of avionics increased from 10^4 magnitude in 1970s upto 10^7 in 2000s. This exponential increase of code size implicates the exponential needs in performance and functionalities [1, 6, 5] in the industries. To match the performance requirement, multi-cores have been considered as a potential platform candidate taking account of their increasing average performance.

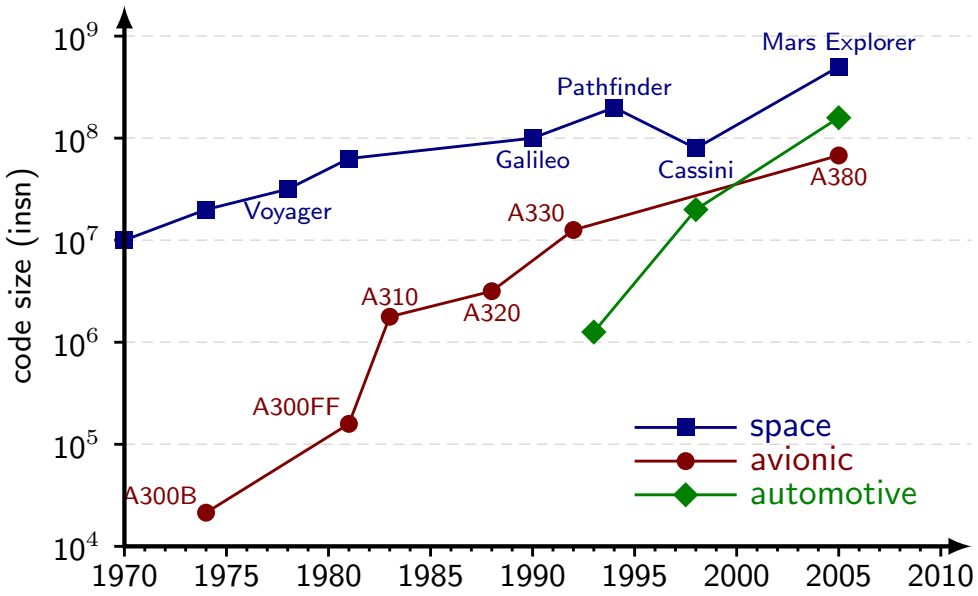


Figure A.1.3: *Evolution of code size in space, avionic and automotive embedded systems*

However, as multi-cores are mostly driven by the consumer electronic market which aims at best-effort performances, the use of multi-cores will make safety-

critical industries to face more and more **runtime variability** issues while offering the increasing performance.

Shared Resource Problem In nowaday multi-cores, each core has its private resources, like L1 D/I caches, but all the cores usually share the last level cache and the same main memory through a common interconnect bus.

If users run several applications independently in different cores in a multi-core, such applications have to share the same hardware resources, like the interconnect bus and the memory, even though they do not communicate the data with each other. For example, two applications want to access to the interconnect at the same time, the interconnect has to decide which one should go first. The interconnect arbitration may add a delay to both application's execution time. This delay thus results in the runtime variability.

Despite all the improvements in the WCET estimation domain [19, 10] over the last decades, the over-estimation remained mostly constant as the predictability of the architecture decreased [38], thus making the use of WCET analysis tools difficult for real industrial programs running on multi-core architectures [18, 21]. Possible interference on shared hardware resources among co-running tasks significantly increases the complexity of timing analysis, forcing it to have a full knowledge of co-running tasks at software level, and detailed resource contention models at hardware level. Unfortunately, the underlying multi-core architecture and co-running applications both usually behave as a gray- or black-box to users.

Undisclosed Hardware Feature Problem Embedded architectures come with detailed ISA and block diagram, but many aspects of the micro-architecture remain **undisclosed** such as the exact SoC network topology, contention, arbitration and prefetcher mechanisms. If such information is not necessary to guarantee correct functional behavior, it could have a significant impact on the **timing behavior**, that is as much important for safety-critical real-time systems. For instance, the Freescale QorIQ platform P4080 presented in chapter A.3 has no information about the exact behavior and the topology of its shared interconnect. Since the lack of the understanding about contention mechnisms of these important resources which can be potential performance bottlenecks in the context of co-running applications, we are hardly able to predict the performance variability derived from the contention on these shared resources. As a consequence, black-box multi-cores prevent users to accurately understand and predict co-running application behavior/performance on it.

In addition to architectures, co-running applications are also black-box. We do not know how a standalone application behaves on shared hardware resources, not to mention the interference of co-running applications on shared resources.

A.1.5 Objectives

Considering the challenge of using multi-cores to estimate the runtime variability of co-running safety-critical applications, we proposed two approaches in this thesis to:

- Characterize the underlying architecture to discover its undisclosed features, especially the shared resource related mechanisms and characterize target applications to observe their behavior on shared resources.
- Estimate the execution time upper bound of co-running applications based on the information of characterizations.

Chapter A.2

State of the Art

In this chapter, we present first the WCET analysis on single-cores to introduce static and measurement based techniques computing upper bounds on execution times. Second, we present several representative proposals about designing new architectures with less runtime variability. Third, we detail some work on the WCET estimation using multi-core Commercial Off-The-Shelf (COTS) architectures. Last, we present different methods of performance evaluation which provides the behavior information of applications, allowing users to apply during the WCET estimation.

A.2.1 WCET Analysis in Single-cores

Hard real-time systems need to satisfy stringent timing constraints [1, 6, 5]. In general, upper bounds on the execution time are needed to show the satisfaction of these contraints. Figure A.2.1 proposed in [38] by Wilhelm et al. facilitate the understanding of timing analysis by depicting some real-time properties.

A real-time system consists of a number of tasks and each task typically shows a certain variation of execution times depending on the input data or different behavior of the environment. The upper curve refers to the distribution of all the possible execution times of a task where the BCET (Best Case Execution Time) and the WCET (Worst Case Execution Time) are respectively the shortest execution time and the longest execution time. Most of the time, the BCET and WCET can not be accurately determined by exhausting all the possible executions. Therefore, the industries usually measure the end-to-end execution times of a task only for a partial set of all the possible executions to estimate the execution time bound. This measured execution time distribution is depicted by the lower curve in Figure A.2.1 where we have the minimal observed execution time which usually overestimates the BCET and the maximal observed execution time

which usually underestimates the WCET. The upper and lower timing bounds are computed by the methods exploring all the possible executions. The upper bound provides the worst case guarantee to envelope the worst case performance. There are thus two criteria for timing analysis methods:

- **Safety**: Making sure that the WCET is greater than any possible execution time.
- **Tightness**: Keeping the real worst case execution time close to the upper bound. A large **overestimation** implies a great resource **over-provision**.

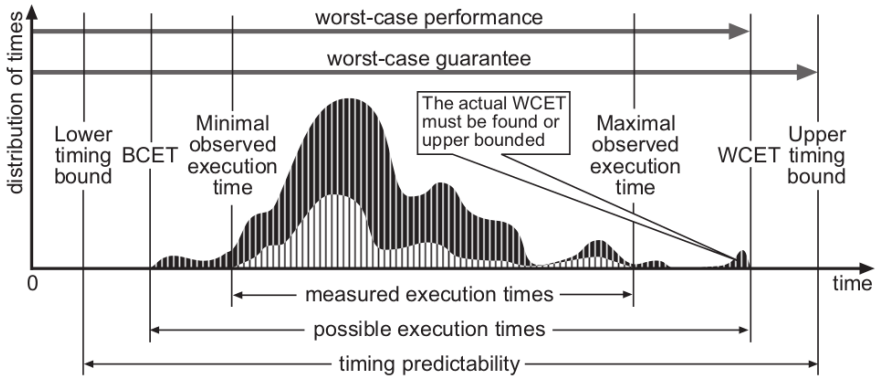


Figure A.2.1: *Basic notions related to timing analysis. The lower curve represents a subset of measured executions. Its minimum and maximum are the minimal observed execution times and maximal observed execution times. The darker curve, an envelope of the former, represents the times of all executions. Its minimum and maximum are the best case and worst case execution times, abbreviated BCET and WCET.*

To compute the WCET on single-core processors [38, 18], two main classes of methods have been adopted: the **static analysis** and the **measurement-based analysis** (dynamic timing analysis).

A.2.1.1 Static Analysis

Static analysis methods [38] perform some analysis on the application code to extract possible control flow paths which are then combined with an abstract model of the system to compute the upper bound of all the execution times. This analysis do not rely on executing code on real hardware or on a simulator. Therefore, static methods are able to produce bounds allowing safe schedulability analysis of hard real-time systems. There are three key components in static methods:

- **Control-flow Analysis**
  The purpose of control-flow analysis is to determine all the possible execution paths of a task under analysis. The input of this analysis can be the task's control flow graph (CFG), call graph and some additional information such as maximum iterations of loops and so forth. We can finally obtain the actually feasible paths, including the conditional dependencies by eliminating infeasible and mutually exclusive paths. All the flow information performs as behavioral contraints of the task. There are different matured techniques to automate the flow analysis, like the pattern matching method, Bound-T method for loops analysis. The result of control-flow analysis is an annotated syntax tree for the structure- based approaches in the bound calculation, and a set of flow facts about the transitions of the control-flow graph. These flow facts are translated into a system of constraints for the methods using Implicit Path Enumeration (IPET) in the bound calculation.

- **Processor-behavior Analysis**
  The execution time of a task depends on the selected hardware behavior, like the pipeline structure, historical states of the cache and the arbitration mode of the bus, which can be derived from the abstract processor model, the memory hierarchy and the interconnect bus. The abstract processor model depends on the class of used processor. For some simple processors without the pipeline and caches, the timing construction is simple to abstract. Even for processors with a simple scalar pipeline, maybe a cache, the abstract also can be achieved by analyzing different hardware features separately, since there are no timing anomalies. However, for complex processors, some high performance enhancing techniques prevent to create the timing construction. For example, out-of-order executions produce timing anomalies making a cache hit resulting in a longer execution time than a cache miss, which can be hardly modeled. Most approaches use Data Flow Analysis, a static program-analysis technique, to get static knowledge about the contents of caches, the occupancy of functional units and processor queues, and of states of branch-prediction units.

- **Bound Calculation**
  The purpose of this phase is to compute an upper bound of all execution times of the whole task based on the flow and timing information derived in the previous phases. There are three main classes of methods proposed in literature: structure-based, path-based, and techniques using Implicit Path Enumeration (IPET). In the structure-based method, the annotated syntax tree derived in the flow analysis is traversed from bottom to up to

first determine the execution time bound of each basic block and second combine all the computed bounds to deduct the upper bound of the whole task. The path-based calculation is to find the longest execution path based on the CFG with timing nodes derived in previous phases to determine the upper bound. In IPET, the flow of a program is modeled as an assignment of values to *execution count variables*. The values reflect the total number of executions of each node for an execution of the task. Each entity with a *count variable* also has a *time variable* giving the contribution of that part of the program to the total execution time. The upper bound is computed by maximizing the sum of products of *count variables* and *time variables*.

Although the static analysis can offer a sound and safe bound, there are some limitations during the processor-behavior analysis. First, the contention on shared resources can not be accurately modeled. In addition, the knowledge of the hardware is usually limited for users. As a consequence, the measurement-based analysis is proposed to overcome it.

A.2.1.2 Measurement-based Analysis

Measurement-based methods [38] replace the processor-behavior analysis in the static methods by performing measurements on the target hardware (or a detailed simulator) to estimate the WCET. This analysis usually measures the execution times of code segments, typically of CFG basic blocks. As in the static analysis, the measurement-based analysis also use the control flow analysis to find all possible paths and then use the bound calculation to combine the measured times of the code segments into an overall time bound.

However, the measured execution times of basic blocks would be unsafe if only a subset of input data or initial states were considered. Unsafe execution times may produce an unsafe upper bound which can not be accepted by safety-critical industries but can be used to provide a picture of the actual variability of the execution time of the application.

A.2.1.3 Commercial WCET Tools

aiT aiT WCET Analyzer is the first software tool of the well-known statical industiral tool AbsInt [9] designed in the *IST project DAEDALUS* according to the requirements of Airbus France for validating the timing behavior of critical avionic software, including the flight control software of the A380, the worlds largest passenger aircraft. The purpose of aiT tool is to obtain upper bounds for the execution times of code snippets (tasks) in executable by statically analyzing a tasks intrinsic cache and pipeline behavior based on formal cache and pipeline models.

The workflow of aiT is shown in Figure A.2.2 proposed in [38]. The analysis is composed of three main steps: 1) Reconstruct CFG from the given executable program and then translate the CFG into CRL (Control Flow Representation Language, a human-readable intermediate format designed to simplify analysis and optimization at executable/assembly level) served as input of the next step. 2) Value analysis determines potential values in the processor registers for any possible program point. The analysis results are used to identify possible addresses of memory accesses for cache analysis, to determine infeasible paths resulting from conditions being true or false at any point of the analysed program, and to analyse loop bounds. The following cache analysis statically analyze the cache behavior of a program using a formal model examining sure hits and potential misses. The pipeline analysis models the prpcessor's pipeline behavior based on the current state of the pipeline, the resources in use, the contents in prefetch queues and the results obtained during cache analysis. It aims at finding the WCET estimate of each basic block of the program. 3) Bound calculation based on the path analysis determines the worst-case execution path of the program relying on the timing information of each basic block.

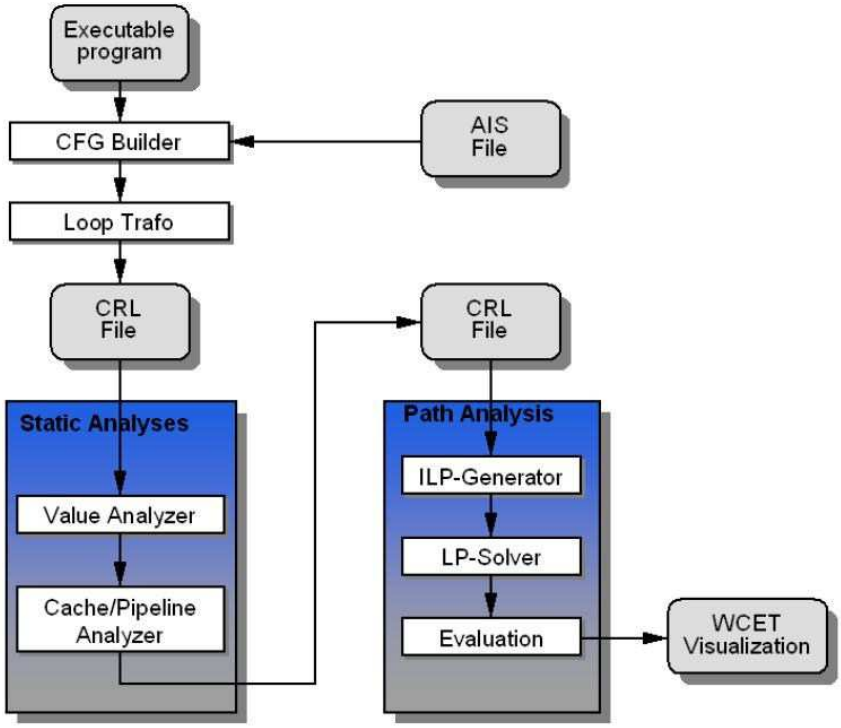


Figure A.2.2: *Workflow of aiT WCET analyzer.*

aiT includes automatic analysis to determine the targets of indirect calls and branches and to determine the upper bounds of iterations of loops. However, this

analysis does not work in all cases. If it fails, the users have to supply annotations. In addition, aiT relies on the standard calling convention that may not be respected by some code. In this case, the users have to provide additional annotations describing control flow properties of the program.

RapiTime RapiTime [19, 38] is a measurement-based tool aiming at medium to large real-time embedded systems on advanced processors. It targets the automotive electronics, avionics and telecommunications industries. RapiTime derives timing information of each basic block of the program from the measurements, and then combines all the measurement results according to the structure of the program to estimate the longest path of the program. RapiTime not only computes the upper bound of the program as a single value but also gets the whole probability distribution of the execution time of the longest path in the program. The input of RapiTime can be either source code files or binary code files, and the users have to provide test data for measurements.

Compared to the static tools, RapiTime does not rely on the processor model, so it can model any processing unit based on its measurement results. However, the limitation is also put on the measurement. It has to extract the execution traces of basic blocks in the running system using code instrumentations or other measurement mechanisms. Regarding source code level, RapiTime cannot analyse programs with recursions and with non-statically analyzable function pointers.

SymTA/P (SYMbolic Timing Analysis for Processes) SymTA/P [16, 38] is a hybrid approach combining the static and measurement-based analysis to obtain lower and upper execution time bounds of C programs running on microcontrollers. The key concept of SymTA/P is to combine platform independent path analysis on source code level and platform dependent measurement methodology on object code level using an actual target architecture. SymTA/P uses symbolic analysis on the abstract syndax tree to identify single feasible paths (SFP). SFP is a sequence of basic blocks where the execution sequence is invariant to input data. The result of the analysis is a CFG with nodes containing SFPs or basic blocks that are part of a multiple feasible paths. In the second step, the execution time of each node is estimated on an off-the-shelf processor simulator or an evaluation board by instrumenting C code with measurement points that mark the beginning and the end of each node. However, such measurement can not ensure a safe initial state in all cases, so an additional time delay is added to cover a potential underestimation during such measurement by many techniques. The longest and the shortest paths in the CFG are found by IPET introduced in Section A.2.1.1.

However, data-dependent execution times of single instructions are not explicitly considered. The input data has to cover the worst case regarding data-dependent instruction execution time, which means that the input data has to generate complete branch coverage. In addition, there is no sub-function analysis during cache-analysis. Each function is analysed seperately (assuming an empty cache at function start), and there is no interference assumed when a function is called.

A.2.2 In-house Approach for the WCET Estimate

As we stated in Chapter A.1, multi-core architectures provide high average performance and are increasingly considered as execution platforms for embedded systems in the safety-critical domain. However, the performance variability, namely the reduced predictability is a critical issue that should be avoided in the safety-critical context. To increase the performance predictability, two different approaches have been advocated. The first is the in-house approach and the second is the COTS approach. We present, in this section, some related work about the former and in the next section about the latter.

With the advent of multi-core architectures the WCET analysis methods for single-cores are no longer able to estimate WCETs due to the shared hardware resources and mechanisms (network-on-chip, memory coherency, ...) which add too much performance uncertainty. In order to provide tight WCETs the research community has mainly focused on in-house approach which aims at proposing new and more predictable multi-core architectures implementing proper isolation mechanisms to separate the use of shared resources and provide a adequate level of determinism, as in multiple european and american projects PREDATOR [26], MERASA [33] / parMERASA [34] and PRETS [7].

The MERASA project aims to achieve a breakthrough in hardware design, hard real-time support in system software and the WCET analysis tools for embedded multicore processors. The project focuses on developing multicore processor designs for hard real-time embedded systems and techniques to gurantee the analyzability and timing predictability of every feature provided by the processor.

Intertask interferences in mainstream multicores provoke the main difficulty in analyzing timing behaviour, which renders multicores unusable in safety-related real-time embedded systems. In this context, MERASA architecture is designed in [33] to make analysis of each task independent from coscheduled tasks and allow safe and tight worst-case execution time (WCET) estimation. The general MERASA multicore architecture is based on SMT cores and is capable of running both hard real-time (HRT) and non hard real-time (NHRT) threads. However,

HRT threads receive the highest priority in the fetch stage, the real-time issue stage, and the intracore real-time arbiter. Moreover, each HRT task has access to its local dynamic instruction scratchpad (D-ISP) and data scratchpad (DSP), private instruction, and data cache partitions. Thanks to these tailors, HRT threads are isolated from NHRT threads. In addition, analyzable real-time memory controller (AMC) is proposed to minimize the impact of memory intertask interferences on the WCET. With respect to WCET techniques and tools, the MERASA project uses a static WCET tool and a measurement-based WCET tool to evaluate impact of intertask inference on WCET estimation.

The philosophy of the PRETS project [7] is to propose very predictable architectures whose temporal behavior is as easily controlled as their logical function. The timing control research thus spans all abstraction layers in computing, including programming languages, the ISA level, the memory hierarchy, the interconnect architecture, the DRAM design and so forth.

B.Lickly et al. [20] focus on integrating timing instructions to a thread-interleaved pipeline and a predictable memory system. With respect to pipeline, based on SPARC v8 ISA, PRET architecture implements a six-stage thread-interleaved pipeline that supports six independent hardware threads. Each thread has its own register file, local on-chip memory (scratchpad memories (SPMs)) and assigned off-chip memory. The thread controller schedules the threads according to a round-robin policy. This pipeline eliminates dependencies among instructions in the same thread. However, structural hazards do exist, which will stall the pipeline. To avoid to stall the whole pipeline, a replay mechanism is used by repeatedly replaying the stalling thread until it can continue. Moreover, in order to provide precise timing control to software, a deadline instruction is offered to allow the programmer to set a lower bound deadline on the execution time of a segment of code through accessing cycle-accurate timers. With respect to memory system, PRET replaces caches with SPMs which are managed by software through direct memory access (DMA) transfers, thus avoiding unpredictability of hardware replacement policies. In order to isolate off-chip memory access among each thread, a memory wheel is provided to determine which thread is allowed to access memory through a fixed round robin schedule, which finally ensure a predictable timing. Although all above timing control techniques are able to ensure the real-time constraints of the design with ease, the timing constraints have to be calculated by hand whenever the code is optimized or modified. The lack of the automated tool for calculating and verifying timing constraints prevent the realistic programming for this PRET.

In [3], the PRETS authors discuss techniques of temporal isolation on multiprocessing architectures at the microarchitecture level, the memory architecture, the network-on-chip (NoC), and the instruction-set architecture (ISA). At the microarchitecture level, temporal interference can be removed by assigning a

top priority to hard real-time threads or using virtual multiprocessor or thread-interleaved pipeline. At the memory level, partitioned caches can be applied to avoid the interference in the cache by pre-allocating a fixed cache region to each core or application. In addition, scratchpad memories (SPMs) are an alternative to caches. SPMs provide constant latency access times and the contents are under software control. For the next lower level of the memory, dynamic random access memory (DRAM) controller allocates private banks to different clients. At the NoC level, there are two approaches: one is time-division multiple-access (TDMA), and the other is priority-based mechanism. At the ISA level, some new instructions with temporal semantics considered as ISA extensions are proposed to enable control over timing.

Reineke et al. [29] propose a novel dynamic random access memory (PRET DRAM) controller is proposed in which the DRAM device is considered as multiple resources shared between one or more clients individually. The physical address space is partitioned following the internal structure of the DRAM device, ie., its ranks and banks. The DRAM controller is split into a backend and a frontend. The frontend connects to the processor, and the backend issues commands to the independent resources in DRAM module with a periodic and pipelined fashion. Therefore, a bank parallelism is realized and interference amongst the resources is removed.

The Proartis project [4] proposes an interesting approach for the WCET problem on multi-cores by proposing architecture designs with randomness. Thanks to the randomness properties of such designs probabilistic approaches can be applied to compute accurate WCETs.

The advantage of these in-house approaches is that they can address temporal and spatial isolation directly, which makes few or no modification in terms of the software, like the operating system. However, hardware modifications especially complex modifications lead to high custom silicon cost and a long time-to-market (TTM). From my best knowledge, there is no MERASA or PRET architecture commercialised in the market. Furthermore, for these in-house architectures, the high predictability is achieved at the cost of their average performance, which makes them not able to sustain the increasing performance requirement shown in Figure A.1.3.

A.2.3 COTS Approach for the WCET Estimate

COTS architectures refer to the commercial components that we can directly get in the market. Therefore, compared to in-house solutions, COTS architectures reduce both the non-recurring engineering costs (NRE) and the TTM [2]. Considering these advantages and multi-core COTS's high average performance

safety-critical industries have been seeking the methods to achieving the worst case timing analysis based on multi-core COTS. The critical issue of using multi-core COTS is reduced predictability resulting from the interference among co-running appications on shared resources. The performance in the context of co-running applications is mostly slown down either by concurrent accesses to a shared bus/memory or by changed states of shared caches.

Pellizzoni et al. [24] propose a methodology computing upper bounds to task delay due to memory contention. The methodology adopts task model based on superblocks. Each task is composed of a sequence of superblocks characterised by a *cache profile* C_i=$\{\mu_i^{min}, \mu_i^{max}, exec_i^L, exec_i^U\}$. μ_i^{min}, μ_i^{max} are the minimum and maximum number of access requests to the main memory, and $exec_i^L$, $exec_i^U$ are lower and upper bounds on computation time for superblock s_i. To bound the amount of concurrently requested access time for a given superblock, all possible interleavings of its bus accesses with concurrent access sequences have to be taken into account. As this may be computationally infeasible, an arrival curve $\alpha_i(t)$ is introduced in the article based on the cache profiles of all the tasks executed on the processing unit i (PE_i). $\alpha_i(t)$ bounds the maximal amount of access time required by concurrent tasks running on PE_i to perform operations in the main memory in a time interval t. Arrival curves are then combined with a representation of cache behavior of the task under analysis to generate a delay bound. The principle of delay analysis is to construct the worst-case scenario which is the scenario that maximizes the latency of a request based on the memory arbitration scheme. For round-robin arbitration, the worst-case scenario is that all other processing units are allowed to perform one access before the PE_i is allowed to do so. In First Come First Served (FCFS) arbitration, an interference bound has to additionally take into account the maximum number of access requests that can be released at the same time by concurrent processing units if this number is greater than one. Pellizzoni et al. finally give a delay bound equation that was evaluated through an extensive simulations to derive the delay bounds.

In addition to [24], superblocks are also applied in [30, 23] to analyse the delay bounds due to the contention on shared bandwidth resources. However, this approach requires the bounds on the amount of computation time and the access times and delays to be compositional, which means that the approach should rely on timing compositional hardware architectures [39]. Unfortunately, many existing hardware architectures exhibit domino effects and timing anomalies and are thus out of the scope of such an approach.

Compared to the bandwidth resources, like the interconnect bus, which bring the runtime delay via their arbitration mechanisms when different applications request them at the same time, there is another type of shared resources, like caches. When one application changes the state of a resource, another application using that resource will suffer from a slowdown. Unfortunately, the behavior of

current shared caches is hard to predict statically. Cache accesses from different cores are typically served on a first-come first-served basis. Their interleaving thus depends on the relative execution speeds of the applications running on these cores, which depend on their cache performance, which in turn depends on the cache state. This cyclic dependency between the interleaving of the accesses and the cache state makes precise and efficient analysis hard or even impossible in general.

Chi Xu et al. in [40] propose CAMP, a fast and accurate shared cache aware performance model for multi-core processors. CAMP estimates the performace degradation due to cache contention of processes running on chip multiprocessors (CMPs). The model uses non-linear equilibrium equations in a least-recently-used (LRU) and pseudo-LRU last level cache, taking into account process reuse distance histograms, cache access frequencies and miss rate aware performance degradation. CAMP models both cache miss rate and performance degradation as functions of process effective cache size, which is in turn a function of the memory access behavior of other processes sharing the cache. CAMP can be used to accurately predict the effective cache sizes of processes running simultaneously on CMPs, allowing the performance prediction.

Fedorova et al. [8] describe a new operating system scheduling algorithm that improves performance isolation on chip multiprocessors (CMP). This algorithm is a cache-fair algorithm ensuring that the application runs as quickly as it would under fair cache allocation, regardless of how the cache is actually allocated.

Zhao et al. [41] presented an approach to dynamic scheduling that is based on their CacheScouts monitoring architecture. This architecture provides hardware performance counters for shared caches that can detect how much cache space individual tasks use, and how much sharing and contention there is between individual tasks.

An approach to static scheduling in a hard real-time context is presented by Guan et al. [13]. They extend the classical real-time scheduling problem by associating with each task a required cache partition size. They propose an associated scheduling algorithm, *Cache-Aware Non-preemptive Fixed Priority Scheduling, FPCA* and a linear programming formulation that determines whether a given task set is schedulable under FPCA . For higher efficiency, they also introduce a more efficient heuristic schedulability test that may reject schedulable task sets.

All these above proposals are static analysis of shared resources to try to provide a sound and safe bound in a co-running context. In addition to them, there are measurement-based approaches that aim at quantifying the slowdown a task experiences when different tasks are executed in parallel. In a single-core setting, a measurement-based estimate is obtained by adding a safety-margin to the longest observed execution time. However, it is not possible to directly extend such measurement-based timing analysis from the single-core to the multi-core

setting because of increased variability of the runtime.

Radojković et al. [28] propose benchmarks that stress a variety of possibly shared resources, including functional units, the memory hierarchy, especially the caches at different levels and the bandwidth to the main memory. These benchmarks are called **resource stressing benchmarks** which aim at maximizing the load on a resource or a set of resources. The interference a resource stressing benchmark causes on a certain resource is meant to be an upper bound to the interference any real co-runner could cause. Therefore the slowdown a program experiences due to interferences on a certain resource when co-running with a resource stressing benchmark bounds the slowdown that could occur with respect to this resource in any real workload. Resource stressing benchmarks are thus considered as a good metric to obtain a workload-independent estimate of the performance slowdown. The authors applied resource stressing benchmarks in multithreaded processors to 1) estimate the upper limit of a performance slowdown due to different shared-resource contention for simultaneously-running tasks. 2) quantify time-critical applications sensitivity to different shared-resource contention. 3) determine if a given multithreaded processor is a good candidate to meet the required timing contraints.

Experiments are carried out on three architectures with different shared resources in order to show the varying timing predictability, depending on the degree of resource sharing. One architecture Atom Z530 offers hyperthreading, and the resources from the front-end of the pipeline to the memory bandwidth are shared. For the second architecture Pentium D, the only shared resource is the bandwidth to the main memory between two cores whereas for the third architecture Core2Quad, the L2 cache is shared as well. In order to show the variance of the possible slowdown, measurements are taken for three different scenarios. In the first scenario, only resource stressing benchmarks are executed concurrently to estimate the worst possible slowdown independently of the application. In the other cases, the application is either executed with another co-running application or with different co-running resource-stressing benchmarks. The experimental results of the first scenario reveal that the techniques like hyperthreading are impractical for hard real-time applications due to a possible significant slowdown caused by the contention on all the shared resources. The measurements for the different scenarios reveal that the slowdown due to co-running resource stressing benchmarks drastically exceeds the slowdown measured in workloads only consisting of real applications. This implies that the workload- independent slowdown determined with co-running resource stressing benchmarks yields a very overestimated upper bound to the slowdown in any real workload. Therefore, the analysis result might be not very useful in practice.

Another measurement-based approach proposed by Nowotsch and Paulitsch [22] quantifies the impact of integration of multiple independent applications

onto multi-core platforms using resource stressing benchmarks. Measurements are carried out on the Freescale P4080 multi-core processor, and access two key parameters: influences introduced by multiple cores concurrently accessing network and memory and additional overhead introduced by coherency mechanisms. Several scenarios are considered:

1. configure L3 caches to SRAM in order to compare accesses to SRAM and DDR memory when coherency flag in turned off

2. compare accesses to SRAM and DDR memory in case of static coherency enabled (i.e. only checks whether memory blocks in the local caches are coherent)

3. compare accesses to SRAM and DDR memory in case of dynamic coherency enabled (i.e. not only checks, but also explicit memory operations enforcing coherency)

The outcomes show that the time for concurrent accesses to DDR memory scales very badly with the number of concurrent cores, in contrast to SRAM. Concerning the different cache coherency settings, the results show that even without coherent accesses, static coherency induces an overhead in execution time that should be considered in timing analysis. The impact of dynamic coherency strongly depends on the type of memory operation (read or write). In case of read with concurrent read, dynamic coherency does not cause any slowdown compared to static coherency. For write operations, the execution is slown down significantly in case of dynamic coherency, regardless of the concurrent operation. This can be explained by the fact that after a write operation, coherency actions are required to keep the memory hierarchy consistent. Throughout all the evaluation results, the article points out that multi-core COTS may be used for safety-critical applications, but the WCET may be a factor of the number of cores being used.

The static approach is always highly complex due to the large number of potential hardware features to be considered during modeling. For complex architectures, several simplifications and unrealistic assumptions have to be set in order to derive a safe modeling, which hinders its widespread use in industries. The measurement-based approach is a good complement to the static approach by searching the worst case interference of co-running tasks to estimate the upper bound. However, compared to the static approach, there is a lack of formal proofs to support the measurement-based approach. In addition, measured execution times might not be reproduced in the next measurement because of the variability. Therefore, the measurement usually has to be repeated to cover the worst-case scenario.

A.2.4 Performance Evaluation

The purpose of the performance evaluation is to monitor how an application behaves on an architecture. This monitoring information allows us to identify possible bottlenecks of performance which guide us to maximize the performance through various optimizations. In addition, it is also useful for characterizing workloads, allowing us to quantify hardware resource utilization at application level and providing some clues to better understand the runtime variability. There are broadly two distinguished profiling techniques: *sampling* and *instrumentation*.

Sampling Techniques
In the sampling approach, the measurement tool stops the target application in fixed intervals or at specific events and samples the program counter as the application progresses to measure performance aspects statistically by triggering relevant interrupts. At this point, additional information about hardware and program state can also be recorded. By unwinding the stack-frames, the tool is then able to roughly pinpoint the corresponding source position from the debug information.

The SimPoint tool [31, 25] is an automatic technique relying on sampling techniques to find and exploit program phase behavior to guide program analysis, optimization and simulation. During the phase analysis, different metrics of the program are captured within each fixed interval or variable-sized interval to classify the intervals within the programs execution that have similar behavior/-metrics as the same phase.

Instrumentation Techniques
In the instrumentation approach, the target program is augmented with measurement points, called probes which are mostly installed at all entries and exists of a given function, to gather the necessary information. For example, the inserted hooks allow the profiler to maintain a shadow stack at runtime. The stack stores the time at which a function is entered, which is subtracted from the time when the function returns. The accumulated differences precisely capture how much time was spent in each function.

There are several well-known program profiling tool based on the instrumentation. Valgrind [35] provides a generic infrastructure for supervising the execution of programs by providing a way to instrument programs in very precise ways, making it relatively easy to support activities such as dynamic error detection and profiling. The most used tool in Valgrind is *memcheck* which inserts extra instrumentation code around almost all instructions, which keeps track of the memory state, lile its validity and addressability. In addition, Joint Test Action Group (JTAG) which is propably the most widely used debug interface applies

the processor instrumentation to examine and modify the internal and external states of a system's registers, memory and I/O space.

Both sampling and instrumentation are in principle capable of gathering the same information, though each of them has proper advantages and drawbacks. We can easily control measurement dilation under sampling by adjusting the sampling frequency. However, as sampling is performed outside of the scope of the observed application, important events occurring in between sampling points may potentially be missed. In contrast, the instrumentation approach where an event will always be observed provides deterministic and reproducible results, but automatic placement of event-probes is difficult to control and has tendencies to fail for current C++ codes due to immense overhead.

In the thesis, we propose methodologies characterizing architecture and applications, and computing the runtime upper bound based on the performance evaluation using *hardware monitors*. Hardware monitors are essentially a group of special purpose registers implemented in most recent architectures include some special on-chip hardware to count hardware related activities. The collected hardware monitors information provides performance information [32] on the applications, the operating system, and the underlying hardware behavior.

The special purpose registers available from the micro-architecture instruction-set are confined to count on-die related events. In some architectures, it would imply not being able to gather some events related to the last cache level, as well as the interconnect, and the DDR controller. As contention on these resources can actually be the bottleneck of the architecture, we cannot afford not collecting this information.

However for recent embedded architectures, the integrators are proposing some monitoring facilities at platform level. These monitoring features are most of the time dedicated to debugging through proprietary hardware probes but allow to count events at all the platform levels, including the number of DRAM refresh, page switch, and so on. This ability is a proof of the existence of some additional (most of the time undocumented) platform-level hardware counters that could be exploited through some reverse-engineering.

A.2.5 Conclusion

There are two complementary timing analysis methods for single-cores: the static and the measurement-based method. Although the static analysis is able to provide a sound and safe execution time bound, the processor-behavior analysis can not be achieved for some complex processors with out-of-order executions which

produce timing anomalies. In addition, the static analysis supposes that it has a complete and detailed behavior description of the analysed architecture. Unfortunately, this is not the case of COTS architectures which typically only provide a high level description. The measurement-based analysis is an alternative method to estimate the WCET by replacing the processor-behavior analysis with performing measurements on the target hardware or a detailed simulator. However, the measurement-based analysis would offer an unsafe bound if only a subset of execution paths and states were considered.

Multi-core architectures are increasingly considered as execution platforms for safety-critical systems since they offer a good energy-performance tradeoff. However, they produce the increased performance variability that should be analysed and controlled in the safety-critical domain. Considering the increasing complexity of timing analysis caused by the interference on shared resources in multi-cores, the static and measurement-based analysis for single-cores are no longer usable for multi-cores. For multi-cores, both approaches are not scalabale: the static analysis has to face a state explosion which is impossible to be all covered during modeling, and the measurement-based analysis has to suffer sequential consistency problems. To achieve timing analysis in multi-cores, two approaches have been advocated: the in-house approach and the COTS approach.

For the in-house approach, the projects (PREDATOR [26], MERASA [33] / parMERASA [34] and PRETS [7]) propose new architectures dealing with the inter-task interferences on WCET to finally reduce the performance variability. The advantage of the approach is that it directly addresses temporal and spatial isolation in hardwares, and frequently requires few or no modifications to softwares. However, the hardware modifications can make the new architecture prohibitively costly and long TTM.

In contrast to the in-house approach, the COTS approach has lower NRE cost and shorter TTM while leaving the timing analysis difficult due to its complex hardware behavior. The timing analysis using COTS can be splitted into two categories: the static analysis and the measurement-based analysis. The static analysis concentrates on deriving analytic and formal models on accessing shared resources in the co-running context, which provides an upper bound capable of proof. The measurement-based analysis applies resource stressing benchmarks to quantify the possible maximum co-running performance slowdown which is used to bound the execution time of real co-running applications. The measurement-based analysis is, at the most time, too pessimist. The slowdown an application experiences with co-running resource-stressing benchmarks is significantly higher compared to with co-running real applications [28].

Compared to the static analysis, the measurement-based analysis is more straitforward and can be always evaluated through a series of experiments directly in hardware platforms, while the static analysis is always highly complex

due to the large number of potential features to be considered and thus often tested via simulations with several unrealistic assumptions.

In this thesis, we propose a measurement-based approach to estimate the runtime variability of co-running safety-critical applications on a COTS multi-core architecture. The contribution is described in Chapter A.5.

Chapter A.3

Target Platform - QorIQ P4080

The selection of processors for future safety-critical applications depends on some essential factors related to the manufacturer and the processor design. For example, avionic industries select a manufacturer which has experience in the avionic domain and is involved in the **certification process**. In addition, the manufacturer is expected to have a sufficient life expectancy and ensure a long-term support, including willing to exchange the information related to the processor design, debug and test. In the side of the processor design, the industries will evaluate the desired candidate according to a series of technical critiria aiming at identifying undesirable features and correlated mitigation means.

According to above selection rules, the **Freescale's QorIQ P4080** is a potential candidate of the next generation processor for future avionic applications, because Freescale is a long-term provider in the avionic domain and it launched a project MCFA (Multi-Core For Avionics) collaborating with avionics manufacturers to facilitate their certification of systems using multicore processors. In this context, airbus keeps a long term relationship with Freescale about PowerPC based systems. Moreover, predecessors of the QorIQ line have been used in multiple aircraft applications.

Therefore, we selected the P4080 as the target platform to evaluate the experiments, and we present the P4080 structure and features in this chapter.

A.3.1 Structure of the P4080

The P4080 Development System [12] is the 8-core Freescale's QorIQ platform. It belongs to the P4 series which is a high performance networking platform, designed for backbone networking and enterprise level switching and routing. As shown in Figure A.3.1, the system is composed of eight **e500mc PowerPC cores** coupled with private L1 data+instruction caches and a private L2 unified cache.

A shared memory architecture including two banks of L3 cache and two associated DDR controllers is built around the CoreNet fabric. Additionally, several different peripherals are implemented around the CoreNet fabric. The CoreNet fabric is a key design component of the QorIQ P4 platform. It manages full coherency of the caches and provides scalable on-chip, point-to-point connectivity supporting concurrent traffic to and from multiple resources connected to the fabric, eliminating single-point bottlenecks for non-competing resources. This eliminates bus contention and latency issues associated with scaling shared bus/shared memory architectures that are common in other multi-core approaches.

We concentrated on the e500mc cores and the shared memory architecture without using other perpherals. The features of the e500mc and the shared memory are depicted in Table A.3.1.

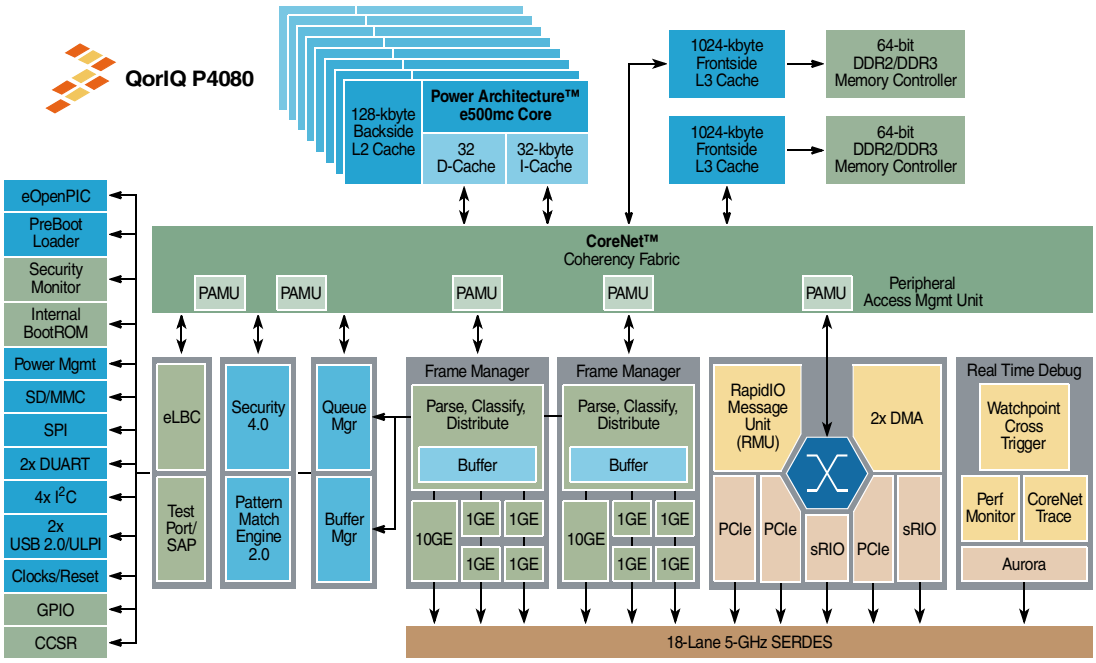


Figure A.3.1: *Freescale P4080 block diagram*

A.3.2 Hardware Monitors in the e500mc Core

Each e500mc PowerPC core provides a special set of registers for the **performance monitoring** called Performance Monitor Registers (PMR). The performance monitor provides the ability to count predefined core related events, for example cache misses, mispredicted branches, or the number of cycles an exe-

Core	8 Power Architecture e500mc at 1.5GHz
Pipeline	7-stage pipeline, superscalar, out-of-order
Distributed L1 caches	32kb, 8-way associative, 64-byte line, PLRU
Distributed L2 cache	128kb, 8-way associative, 64-byte line, PLRU
Performance monitors	162 available events 4 performance monitor counters per core
Shared L3 cache (x2)	1MB, 32-way associative, 64-byte line, PLRU
Memory controller	Two DDR memory controllers
Interconnect	CoreNet Coherency Fabric

Table A.3.1: *Freescale P4080 specifications*

cution unit stalls. According to the manual of e500mc, there are totally 162 available events. Each core has ability to collect four of these 162 events at a given time.

In the P4080, the PMRs can be classified in three types:

- The Performance Monitor Counter registers (PMC) which are used to count selected software-selectable events.
- The Performance Monitor Global Control register (PMGC) which controls the counting of performance monitor events. It is mainly used to freeze/unfreeze all the counters.
- The Performance Monitor Local Control registers (PMLC) which controls each individual performance monitor counter, including freeze/unfreeze each counter, select desired events for each counter, enable/disable the performance monitor interrupt for each counter and etc.

Each PMR has its own PMR number (PMRn) and comes in two versions, one that is read- and writable in the supervisor mode and one that is only readable from the user mode. Table A.3.2 lists all the PMRs and their PMRn in the supervior mode and the user mode. The first step to use PMRs for monitoring is to configure the PMGC0 and PMLCs to control the performance counters and to select desired events. However, writing to the PMGC0 and PMLCs is only legal in the supervisor mode, which means that we can realise the write operations either on the bareboard of the P4080 or using a kernel module on the Linux environment implemented on the P4080. Considering that using kernel modules is more complicated and may produce unexpected overhead during monitoring, we finally used the bareboard method to avoid these drawbacks.

In order to interact with the PMRs, the P4080 provides a pair of assembly instructions *mtpmr/mfpmr* to move to and from the PMRs. The instructions are shown in Table A.3.3.

Name	Supervisor		User	
	Abbreviation	PMRn	Abbreviation	PMRn
Performance monitor counter 0	PMC0	16	UPMC0	0
Performance monitor counter 1	PMC1	17	UPMC1	1
Performance monitor counter 2	PMC2	18	UPMC2	2
Performance monitor counter 3	PMC3	19	UPMC3	3
Performance monitor local control a0	PMLCa0	144	UPMLCa0	128
Performance monitor local control a1	PMLCa1	145	UPMLCa1	129
Performance monitor local control a2	PMLCa2	146	UPMLCa2	130
Performance monitor local control a3	PMLCa3	147	UPMLCa3	131
Performance monitor local control b0	PMLCb0	272	UPMLCb0	256
Performance monitor local control b1	PMLCb1	273	UPMLCb1	257
Performance monitor local control b2	PMLCb2	274	UPMLCb2	258
Performance monitor local control b3	PMLCb3	275	UPMLCb3	259
Performance monitor globam control 0	PMGC0	400	UPMGC0	384

Table A.3.2: *e500mc Performance Monitor Registers*

Name	Mnemonics	Syntax
Move From Performance Monitor Register	mfpmr	rD, PMRn
Move To Performance Monitor Register	mtpmr	PMRn, rS

Table A.3.3: *Instructions for reading and writing the PMRs*

Event selection is specified through an event-select field in the PMLCan registers listed in Table A.3.2. Table A.3.4 describes event types used in Table A.3.5. Table A.3.5 lists some encodings for the selectable events to be monitored and establishes a correlation between each counter, events to be traced, and the pattern required for the desired selection.

Event type	Label	Descriptions
Reference	Ref:#	Shared across counters PMC0PMC3. Applicable to most microprocessors.
Counter	Com:#	Shared across counters PMC0PMC3. Fairly specific to e500 microarchitectures.
Counter-specific	C[0:3]:#	Counted only on one or more specific counters. The notation indicates the counter to which an event is assigned. For example, an event assigned to counter PMC2 is shown as C2:#.

Table A.3.4: *Event types*

A.3.2.1 Performance Monitoring Example Using PMRs

We provide a simple example to explain how to use PMRs in this section. We first show the flowchart of using PMRs in Figure A.3.2. The example is a simple benchmark where we traverse all the elements of an array. For this benchmark, we want to monitor the processor cycles and data L1 cache misses using per-

Numbers	Events	Spec/ Nonspec	Count Description
General Events			
Ref:0	Nothing	Nonspec	Register counter holds current value
Ref:1	Processor cycles	Nonspec	Every processor cycle
Ref:2	Instructions completed	Nonspec	Completed instructions 0,1 or 2 per cycle
Com:3	Micro-ops completed	Nonspec	Completed micro-ops
Com:4	Instructions fetched	Spec	Fetched instructions 0,1,2,3 or 4 per cycle (instructions written to the IQ)
Com:5	Micro-ops decoded	Spec	Micro-ops decoded
Com:6	PM_EVENT transitions	Spec	0 to 1 transitions on the *pm_event* input
Com:7	PM_EVENT cycles	Spec	*Processor cycles* that occur when the *pm_event* input is asserted
...			
Load/Store, Data Cache, and Data Line Fill Buffer (DLFB) Events			
Com:27	Loads translated	Spec	cacheable loads micro-ops translated
Com:28	Stores translated	Spec	cacheable stores micro-ops translated
...			
Com:41	Data L1 cache reloads	Spec	Counts cache reloads for any reason. Typically used to determine data cache miss rate (along with loads/stores completed)
...			
Bus Interface Unit (BIU) Interface Usage Events			
Com:67	BIU master requests	Spec	Master transaction starts (number of Aout sent to CoreNet)
Com:68	BIU master global requests	Spec	Master transaction starts that are global (M=1)(number of Aout with M=1 sent to CoreNet). For e500mc Rev 1.x and Rev 2.x this event is not supported
Com:69	BIU master data-side requests	Spec	Master data-side transaction starts (number of D-side Aout sent to CoreNet)
...			
Snoop Events			
Com:72	Snoop requests	N/A	Externally generated snoop requests (number of Ain from CoreNet not from self)
...			

Table A.3.5: *Some performance monitor event selection of the e500mc*

formance counters. From the description of the example, we know that we can monitor these two events at the same time using two performance counters. Here we select the PMC0 (PMR16) and the PMC1 (PMR17) listed in Table A.3.2 respectively for the processor cycles and data L1 cache misses. Accordingly, we should configure the PMLCa0 (PMR144) and the PMLCa1 (PMR145) to respectively select the event processor cycles whose identification number is 1 and the event L1 cache misses whose number is 41 as shown in Table A.3.5. After determining the performance counters to be used, we begin to configure all the necessary PMRs following the procedures in the flowchart in Figure A.3.2. The source code of the example is shown in Listing A.3.1.

Listing A.3.1: *Example code of using PMRs*

```
/*Freeze counters*/
lis 14, 0x8000;     //set general register r14 to
                    //the value 0x80000000
mtpmr 400, 14;      //PMGC0=r14=0x80000000 to
                    //freeze all the counters

/*Initialize counters*/
```

```
li 14, 0;           //set r14 to zero
mtpmr 16, 14;       //initialize PMC0 to zero
mtpmr 17, 14;       //initialize PMC1 to zero

/*Select desired events*/
lis 14, 1;          //set r14 to 1 which is the number
                    //of the event processor cycles
mtpmr 144, 14;      //configure PMLCa0 to select
                    //the processor cycles
lis 14, 41;         //set r14 to 41 which is the number
                    //of the event data L1 cache misses
mtpmr 145, 14;      //configure PMLCa1 to select
                    //the data L1 cache misses

/*Enable counters*/
li 14, 0;           //set r14 to zero
mtpmr 400, 14;      //PMGC0=r14=0x0 to unfreeze/enable
                    //all the counters

{benchmark executes...}

/*Freeze counters*/
lis 14, 0x8000;
mtpmr 400, 14;

/*Read counters to get the event values*/
mfpmr 14, 16;       //read the value of the PMC0 to r14
mfpmr 15, 17;       //read the value of the PMC1 to r15
```

A.3.3 Hardware Monitors in the Platform P4080

Besides the core-related performance monitoring, the P4080 has the ability to monitor the platform related activities. In the P4080, an infrastructure is integrated to support run control, performance monitoring and tracing. This infrastructure is called Advanced QorIQ Platform Debug Architecture (APDA). The functionality of the APDA is modular.

The P4080 has five functional areas and each of them has its own function-specific debug logic that forwards information to cross-functional facilities to coordinate run control, performance monitoring, and tracing operations.

Five function areas are illustrated below:

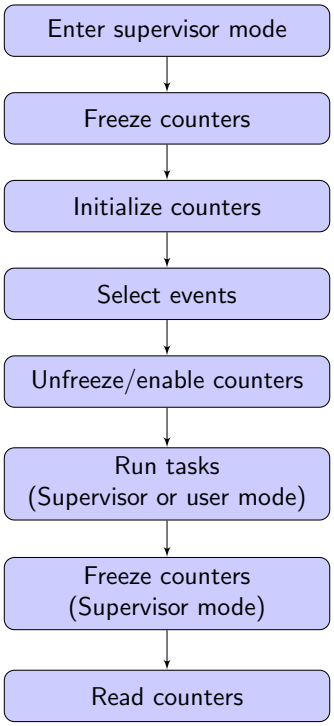


Figure A.3.2: *The flowchart of using PMRs*

- CoreNet - platform coherency fabric and platform cache
- DDR - external memory
- Data Path (Frame Manager and Queue Manager) - network packet processing and queuing
- OCeaN (PCI-Express and Serial RapidIO) - interface to high-speed serial peripherals
- e500mc processor cores

The cross-functional debug components are as follows:

- Event Processing Unit (EPU) - event selection, combining, counting, cross-triggering, and run-control interface
- Nexus Port Controller (NPC) - central ingress and egress point for Nexus commands and trace data
- Nexus Concentrator (NXC) - trace filtering, arbitration, and message formatting

- Nexus Aurora Link (NAL) - high-speed serial debug link over SerDes lanes

The NPC, NXC and NAL are not covered in the thesis, but the EPU is briefly presented as it is the heart of the APDA. The EPU houses the platform performance monitoring function and enables the performance characterization through a set of event counters and their associated performance counter control registers. It allows the user to select, control performance events by configuring corresponding registers similarly with the core-related performance monitors PMRs. There are a maximum of 2048 event signals that can be routed to the 32 platform performance counters. Some events can be combined with each other to produce a new event. It thus provides a high flexibility to users for performance monitoring.

We concentrated on three function areas: the e500mc, the CoreNet and the DDR. The performance monitoring for the e500mc is to use PMRs located in the e500mc as presented in Section A.3.2. To monitor performance activities of the CoreNet and the DDR, we configure relevant registers of debug components to select desired events. Unfortunately, the CoreNet area debug is not documented in the reference manual. As a consequence, we have no ability to monitor the activities related to the CoreNet and the L3 cache. We finally used APDA to monitor DDR concerning activities.

A.3.3.1 P4080 Memory Map Overview

Before explaining how to use platform hardware monitors, we first present the P4080 memory map which will be used for platform hardware monitors configurations. In the P4080, there are several address domains within the device, including:

- Logical, virtual, and physical (real) address spaces within the Power Architecture core(s)
- Internal local address space
- Internal Configuration, Control, and Status Register (CCSR) address space, which is a special-purpose subset of the internal local address space
- Internal Debug Control and Status Register (DCSR) address space, which is another special-purpose set of registers mapped in the internal local address space
- External memory, I/O, and configuration address spaces of the serial RapidIO link

- External memory, I/O, and configuration address spaces of the PCI Express links

The MMU in the core handles translation of logical (effective) addresses, into virtual addresses, and ultimately to the physical addresses for the local address space. The local address map is defined by a set of 32 Local Access Windows (LAWs). Each of these windows maps a region of the local address space to a specified target interface, such as the DDR controller, DCSR, CCSR or other targets. Each LAW is defined by a pair of base address registers that specify the starting address for the window, and an attribute register that specifies whether the mapping is enabled, the size of the window, a coherency subdomain, and the target interface for that window. The concerning registers are described as below:

- LAWn base address register high (LAW_LAWBARHn): It identifies bits 0∼3 of the 36-bit base address of local access window n.
- LAWn base address register low (LAW_LAWBARLn): It identifies bits 4∼23 of the 36-bit base address of local access window n.
- LAWn attribute register (LAW_LAWARn): It is composed of four main fields:
  - EN: Enable this window.
  - TRGT_ID: Target identifier identifying the target interface when a transaction hits in the address range defined by this window.
  - CSD_ID: Coherency subdomain identifier identifying a group of one or more partitions that have access to the address space defined by this access window.
  - SIZE: Identifies the size of the window from the starting address.

A.3.3.1.1 Local Address Map Example

Figure A.3.3 shows a typical local address map example, and Table A.3.6 shows the setting of LAWs corresponding to the each address segment in the example.

A.3.3.2 The Use of Platform Hardware Monitors

To control the debug components of APDA for monitoring, we configure dedicated registers in the Configuration, Control and Status Register map (CCSR), and the Debug Configuration Control and Status Register map (DCSR) presented in Section A.3.3.1. The DCSR and CCSR both belong to the internal local address

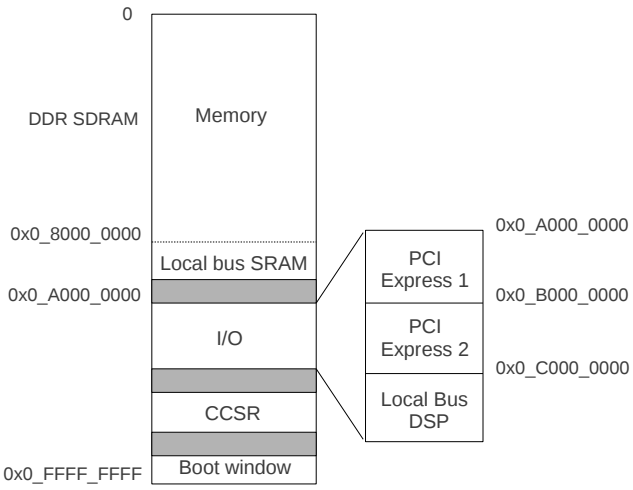


Figure A.3.3: *Local address map example*

Window	Base Address	Size	Target Interface
0	0x0_0000_0000h	2Gbytes	DDR
1	0x0_8000_0000h	1Mbytes	Local bus controller (eLBC)-SRAM
2	0x0_A000_0000h	256Mbytes	PCI Express 1
3	0x0_B000_0000h	256Mbytes	PCI Express 2
4	0x0_C000_0000h	256Mbytes	Local bus controller (eLBC)-DSP
5-9	Unused		

Table A.3.6: *Local Access Window Setting Example*

space in the P4080. Usually, the CCSR address range has been defined in the initialization file of the P4080, but the DCSR is not specified. Therefore, to access to registers mapped in the DCSR, the overall mechanism is shown as the following:

1. Open a LAW for the DCSR by configuring LAW registers presented in Section A.3.3.1. The target ID in LAW_LAWARn should be set to select the DCSR, and the size of the DCSR is 4Mbytes.

2. Configure appropriate MMUs to allow the access from e500mc cores to the DCSR address range.

3. Configure relevant registers in the defined DCSR map to monitor the desired platform events. The flowchart of this procedure is the same with using PMRs shown in Figure A.3.2.

For example, to monitor the event *DDR1 read*, we should configure relevant registers of the EPU and of the DDR debug component. If we want to further

specify the event *DDR1 read issued from core1*, we should additionally configure the registers of the NXC to indicate the source of the DDR read.

We succeeded in monitoring DDR events with the help of the reference manual of the platform debug. But the reference manual is not publicly available. It is provided only to authorized debug hardware and software vendors. Therefore, we can not present the detail of register configurations for the event monitoring.

A.3.4 P4080 Configurations

Most multi-core COTS provide some amount of configurability to adapt users' requirements. For example, caches help to reduce data transfer time between cores and the main memory to improve the overall system performance, but they also bring the performance variability when they are shared among co-running applications. To improve the predictability, most embedded architectures allow to simply disable caches, some provide some degree of hardware partitioning, and few allow the caches to be configured as SRAM memories. However, most applications from Thales today do not allow us to completely disable the cache for the performance reason, so we concentrate on the cache partitioning to increase the predictability.

A.3.4.1 Cache Partitioning in the P4080

The P4080 platform provides us with two shared L3 caches each connected to a dedicated DDR controller. Beyond allowing us to enable one or both of the L3 cache / DDR controller pairs, the P4080 L3 cache offers some hardware partitioning support, allowing us to shift from a cache fully shared by the cores to a cache where each core has a dedicated pre-allocated memory space.

All the configurations of L3 caches is set by programming CoreNet platform cache (CPC) registers which are mapped in the CCSR area. The CPC partitioning is thus achieved through specific configuration control registers in this area. The CPC registers used for paritioning are described as below:

- CPC partition ID register n (CPCPIRn): It indicates the partition ID which equates to the CSD_ID assigned by the LAWs.
- CPC partition allocation register n (CPCPARn): It controls the CPC allocation for different data and instruction transactions. For example, it decides whether data store transactions that miss in the CPC will attempt to allocate in one of the ways defined by the corresponding CPCPWRn.

- CPC partition way register n (CPCPWRn): It decides which partition way(s) can be allocated to transactions that match the PID defined in the corresponding CPCPIRn.

The overall partitioning mechanism involves CPC registers and LAW registers presented in Section A.3.3.1, which is concluded as follows:

1. A transaction matches a LAW whose target ID is the memory complex 1 or 2 (DDR controller1 or controller2).
2. LAW_LAWAR[CSD_ID] identifies the coherency sub-domain.
3. The selected CSD_ID is searched through CPC partitioning control registers (CPCPIRn, CPCPARn, CPCPWRn) to find which way(s) of the CPC has been allocated to this transaction.

CPC Partitioning Example
We provide a simple example to demonstrate how to map a DDR area of 256M (0x000000000 0x00FFFFFFF) to eight ways 0∼7 of the CPC1 (L3 cache 1). The example is depicted in Listing A.3.2.

Listing A.3.2: *CPC1 partitioning example*

```
# 1. LAW28 matches DDR1-256M 0x00000000~0x0FFFFFFF
# Bits 0~3 of 0x00FFFFFFF is all 0
LAW_LAWBARH28 = 0x00000000
# Bits 4~23 of 0x000000000 is all 0
LAW_LAWBARL28 = 0x00000000
# EN=1, TRGT_ID=ID of DDR1, CSD_ID=1, SIZE=256M
LAW_LAWAR28 = 0x8100101B

# 2. Partition1 8 ways 0~7 of CPC1 for LAW28
# PID=CSD_ID=1
CPC1_CPCPIR0 = 0x40000000
# All the types of transactions can allocate 8 ways
   defined by CPC1_CPCPWR0
CPC1_CPCPAR0 = 0xFFFFFFFF
# Allocate 0~7 ways to LAW28
CPC1_CPCPWR0 = 0xFF000000
```

A.3.4.2 Compromise of Different Configurations

Different configurations may result in different overall performance and different performance variability. However, a configuration can hardly satisfy all the

desired requirements, which makes us to compromise in terms of some less significant demand. For example, we have two different configurations of P4080 as below:

- Config1: we enable one L3 cache and its associated DDR controller. The L3 cache is not partitioned, which means that all the cores share the same L3 cache.
- Config2: we enable one L3 cache and its associated DDR controller. The L3 cache is partitioned into eight identical segments and each of them is allocated to one core. Therefore, each core has its proper L3 cache storage.

If we want to test the overall performance and the variability of an application when it runs with other applications independently in different cores under the Config1 and the Config2 in P4080, we can predict in design time that:

- the Config1 will in principle provide better overall performance by better fitting asymmetric load balancing scenario while providing more performance variability because of the shared cache line evictions due to interferences of other cores on the L3 cache.
- the Config2 will provide less performance variability, namely better predictability by preventing cache line evictions due to other cores, because each core has its private pre-allocated L3 cache. However, the Config2 may degrade the overall performance by providing less allocated cache area due to the partition.

Considering the impact of configurations on the behavior in the co-running context, we evaluated different configurations to identify the most suitable one for safety-critical applications. Table A.3.7 lists the selected configurations tested in the thesis. The configuration details and relevant results are shown in Section C.3.1.3 and Section C.5.3.

A.3.5 Conclusion of Target Platform

As the next generation processor candidate in the avionic domain, the P4080 is an eight-core platform based on PowerPC developped by Freescale. In the P4080, there are a series of hardware monitors implemented in e500mc cores and in the platform. The core related hardware monitors allow us to collect 162 events, including *CPU CYCLES, L1 cache misses...*, while platform related monitors capturing DDR events, like *DDR read, DDR write, DDR page closing...*, by configuring relevant registers within cores or in the CCSR and DCSR internal

configuration	level 3 cache		#ddr controllers
	size	associativity	
single controller non-partitioned	1MB, shared by 8 cores	32-way	1
single controller partitioned	128KB, per core	4-way	1
dual controller non-partitioned	2×(1MB, shared by 4 cores)	32-way	2
dual controller partitioned	256KB per core	8-way	2

Table A.3.7: *Four configurations of P4080*

address space. Since there are only four hardware monitors in each core, we are able to collect a maximum of four events per core at the same time. For some event which can not be counted by only one 32 bit monitor, we have to combine two monitors as a chained counter to count the overflow of the event. In this case, the number of events which can be simultaneously collected is less than four. In addition to the core-related events, the platform-related events which can be simultaneously collected are also limited due to the limited number of counters in the EPU and the limited number of components (i.e. comparators, filters ...) in the NXC. As a consequence, we should make a compromise among simultaneously collected events.

In addition, the P4080 supports different hardware configurations adapting to users' requirements. We can enable one or both of the L3 cache / DDR controller pairs and further partition shared L3 caches through configuring CPC and DDR concerning registers along the CCSR space.

Although there are many peripherals implemented around the CoreNet in the P4080, we only concentrated on P4080's e500mc cores and the shared memory-path resources: the CoreNet, L3 caches and DDRs throughout the thesis. To facilitate the use of hardware monitors and eliminate unexpected overheads during the events collection, we realised all the experiments on the bareboard of P4080 without any operating system.

Chapter A.4

Software Environment - CodeWarrior

The CodeWarrior [11] is an integrated development environment (IDE) for the creation of software that runs on a number of embedded systems. In the CodeWarrior toolset, the Editor, Compiler, Linker, Debugger, and other software modules operate within IDE. The IDE oversees the control and execution of the tools.

We used CodeWarrior to develop our applications and benchmarks and we downloaded the binary files .elf into the P4080 using CodeWarrior USB TAP which enables P4080 debugging via a JTAG port while connected to our host computer via USB.

We present, in this chapter, different development stages in the CodeWarrior to implement applications into the P4080.

A.4.1 Creating Projects in CodeWarrior

The CodeWarrior provides a project manager to organize all the files related to the project. The first step of CodeWarrior development process is to create a new project using the project wizard. The CodeWarrior Project wizard presents a series of pages that prompt you for the features and settings to be used when making your program. This wizard also helps you specify other settings, such as whether the program executes on a simulator rather than actual hardware.

The users can create all the projects according to their experimental setup. For example, throughout the thesis, we run one application per core in barebone P4080, so we generate a new project with main settings described below:

- Processor type: P4080 - target hardware.

- Processing model: AMP (one project per core) - Select this option to generate a separate project for each selected core.
- Tool chain: Bareboard: GCC EABI e500mc - Select to execute the program on the bareboard.
- Debugger connection type: Hardware - Select to execute the program on the target hardware.
- Trace configuration: Disable trace and profile for the project to avoid the interference noise.

All the features and settings selected in the above example during the project creation are graphically illustrated in Figure A.4.1.

For each new project, the CodeWarrior automatically configures the P4080 by generating an initialization file and a memory configuration file. We can directly compile a new project and then run/debug it on the P4080. The default configuration set by the CodeWarrior is the one in the first row of Table A.3.7. However, this stationery project is designed to get you up and running quickly with the CodeWarrior for Power Architecture Development Studio. We can absolutely modify the initialization file to adapt to our own demands. For example, to get different configurations of the P4080 described in Table A.3.7, we re-configured registers related to L3 caches and DDR controllers in the initialization file.

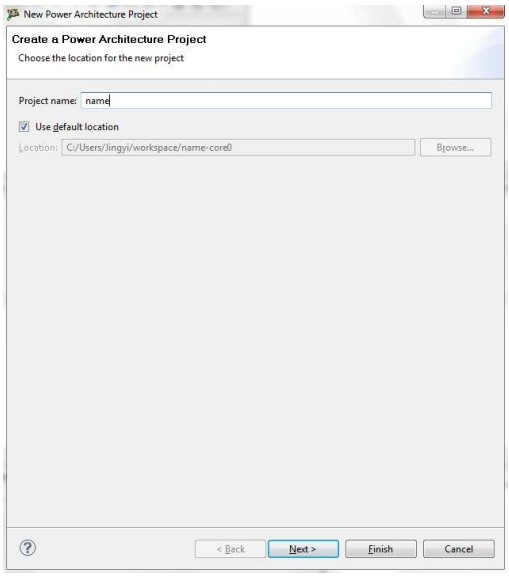
According to the readme.txt file generated in each new project in AMP model, the first core's project (core 0) is responsible for initializing the board. The projects for other than the first core only initialize core specific options, not the whole platform. Therefore launching projects for other cores requires that the core 0's project is running.

A.4.2 Building Projects in CodeWarrior

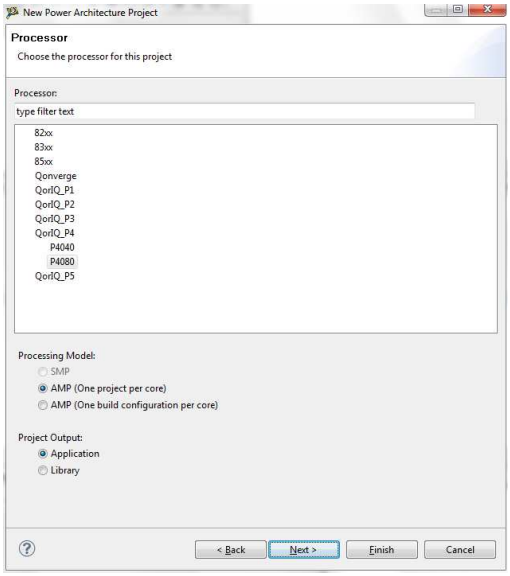
There are two modes of building projects in CodeWarrior: the graphic interface mode and the command line mode.

Graphic Interface Mode

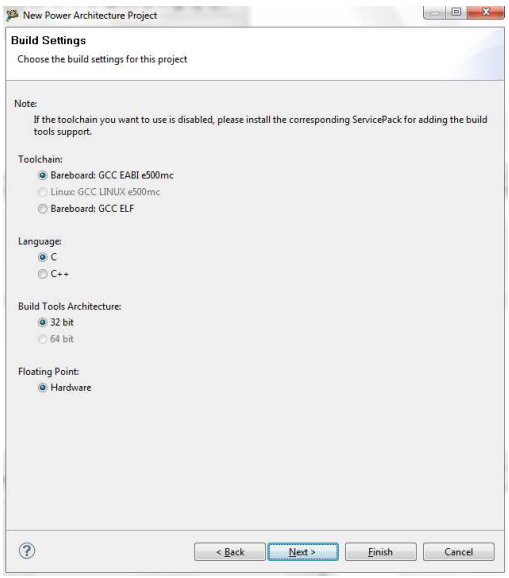
In the CodeWarrior, we can build all the projects in a workspace using option *Build All* in *Project* menu from the CodeWarrior IDE menu bar. Considering that building the entire workspace can take a long time and often there are only a few projects that really matter to a user at a given time, we can also build only the selected projects by right-clicking on the selected project in the CodeWarrior Projects view and selecting *Build Project* from the context menu.


New Power Architecture Project
Create a Power Architecture Project
Choose the location for the new project
Project name: name
Use default location
Location: C:/Users/Jingyi/workspace/name-core0
Browse...
< Back
Next >
Finish
Cancel

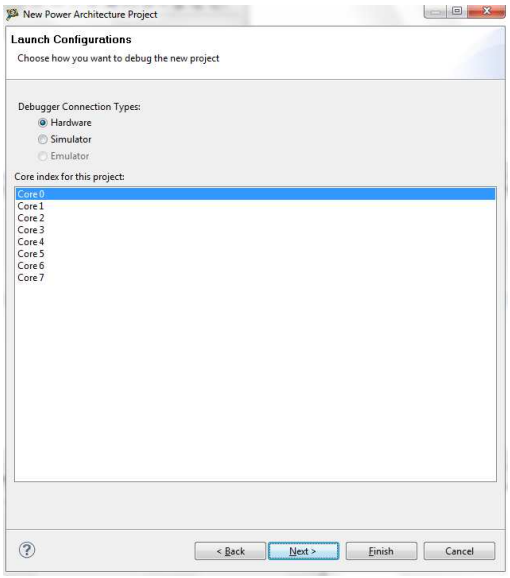
(a)


New Power Architecture Project
Processor
Choose the processor for this project
Processor:
type filter text
82xx
83xx
85xx
Qonverge
QorIQ_P1
QorIQ_P2
QorIQ_P3
QorIQ_P4
P4040
P4080
QorIQ_P5
Processing Model:
SMP
AMP (One project per core)
AMP (One build configuration per core)
Project Output:
Application
Library
< Back
Next >
Finish
Cancel

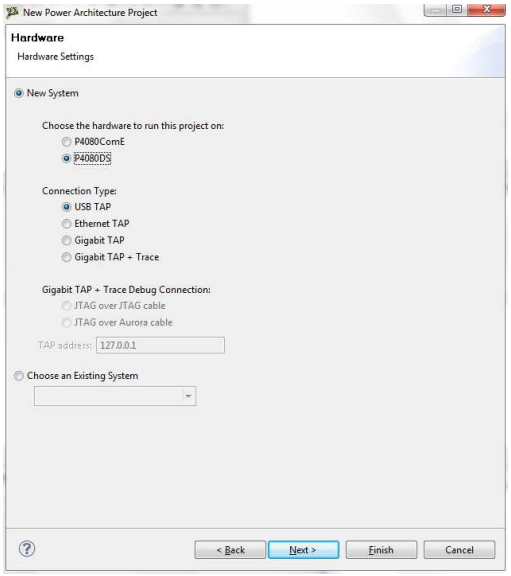
(b)


New Power Architecture Project
Build Settings
Choose the build settings for this project
Note:
If the toolchain you want to use is disabled, please install the corresponding ServicePack for adding the build tools support.
Toolchain:
Bareboard: GCC EABI e500mc
Linux: GCC LINUX e500mc
Bareboard: GCC ELF
Language:
C
C++
Build Tools Architecture:
32 bit
64 bit
Floating Point:
Hardware
< Back
Next >
Finish
Cancel

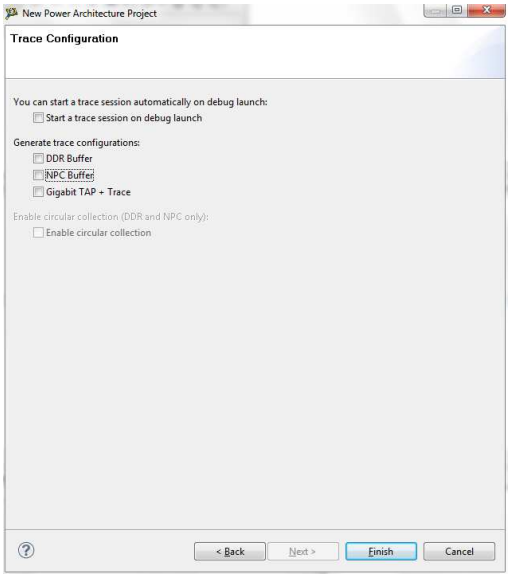
(c)


New Power Architecture Project
Launch Configurations
Choose how you want to debug the new project
Debugger Connection Types:
Hardware
Simulator
Emulator
Core index for this project:
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
< Back
Next >
Finish
Cancel

(d)



(e)



(f)

Figure A.4.1: *Various pages that the CodeWarrior Project wizard displays. (a) Project name and location page, (b) Processor page, (c) Build settings page, (d) Launch configuration page, (e) Hardware page and (f) Trace configuration page.*

Command Line Mode

A new command line tool, ecd.exe, is installed along with the CodeWarrior installation that allows you to run build commands. The command line mode allows us to build projects through a script from outside of the CodeWarrior, which helps automate our experiments. The command **build** is described in following Table A.4.1.

build
Builds a set of C/C++ projects. Multiple-project flags can be passed on the same command invocation. The build tool output is generated on the command line, and the build result is returned by ecd.exe return code, as 0 for success, and -1 for failure
Syntax
ecd.exe -build [-verbose] [-cleanAll] [-project path [- config name | -allConfig] -cleanBuild]
Parameter
-cleanBuild
The -cleanBuild command applies to the preceding -project only
-cleanAll
The -cleanAll command applies to all -project flags
-config
The build configuration name. If the -config flag isn't specified, the default build configuration is used

Table A.4.1: *Description of ecd.exe tool command* **build**

A.4.3 Debugging Projects in CodeWarrior

There are also two modes of debugging projects similar to building projects modes: the graphic interface mode and the command line mode.

Graphic Interface Mode
The CodeWarrior Project wizard sets the debugger settings of the project's launch configurations to default values. We can change these default values based on our requirements from the CodeWarrior IDE menu bar by selecting *Run -> Debug Configurations*. To launch a debugging session and select a debugging operation, like *Step into*, we can click relevant options from the *Run* menu.

Command Line Mode
The CodeWarrior supports a command-line interface to some of its features including the debugger. We can use the command-line interface together with the TCL scripting engine. The **Debugger Shell** view is used to issue command lines to the IDE. For example, you type the command *debug* in this window to start a debugging session. The window lists the standard output and standard error streams of command-line activity. Figure A.4.2 shows the Debugger Shell view. The command line mode helps us automate the experiments.

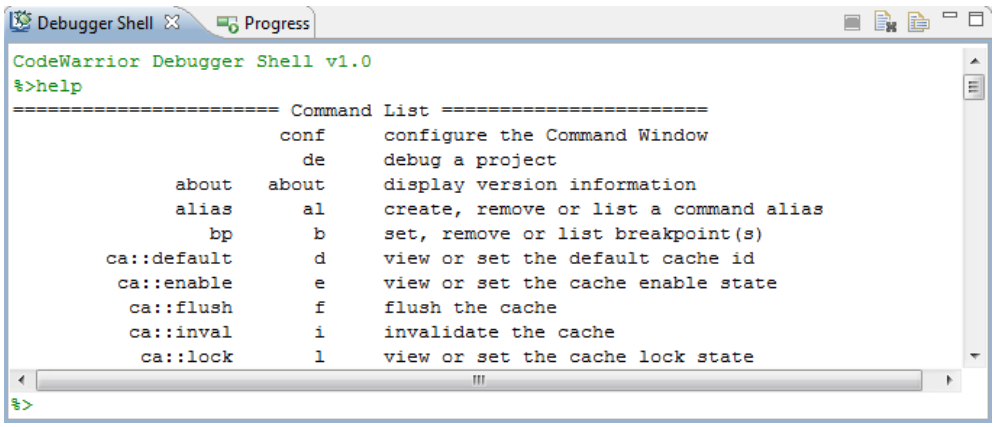


Figure A.4.2: *The Debugger Shell view.*

We will show how to use the CodeWarrior for software developments in Section C.4.3.

Chapter A.5

Contribution

The overall objective of the thesis is to estimate the runtime variability of co-running safety-critical applications on a COTS multi-core architecture. Our proposal will allow us to

- first learn undocumented features of the underlying architecture and applications relatively to shared hardware resources.
- second compute a runtime upper bound of an application co-running with a set of pre-determined applications with gathered shared resource concerning information.

Our proposed approach to achieve the objective is the measurement-based analysis using COTS according to the classification concluded in the state of the art A.2.5. Similar with [28, 22], we also use **resource stressing benchmarks** to quantify the co-running performance slowdown. However, compared to [28] where Radojković et al. only rely on measured execution time to derive the resource sensitivity of an application, we are using **hardware monitors** to capture the shared resource related activities of an application, which allows us not only to observe the sensitivity on shared resources, but also to learn the application's utilization of each shared resource. In addition, Radojković et al. estimate the upper bound of a target application while co-running with other real applications by co-running the target application with resource stressing benchmarks without taking account of the behavior on shared resources of the real co-running applications. Similarly, the authors [22] use resource stressing benchmarks to derive the performance slowdown caused by concurrent accesses to the network and memory under different hardware configurations, and consider the tested maximum slowdown as the worst case slowdown that real applications will experience. However, there is neither a proof nor a formal argument why a specific resource stressing benchmark puts maximum load on the resource. In case of

several possible co-runners, these arguments become even more difficult because of the combined interferences. Another concern demonstrated via experimental results in [28, 22] is that the estimated upper bound is too pessimist compared with the longest measured execution time of co-running real applications. In our approach, *hardware monitors* can provide a proof showing if a resource stressing benchmark or co-running resource stressing benchmarks saturate the resource. Besides, the utilizations of shared resources of each co-running application can help us to design stressing benchmarks simulating the worst case interference on shared resources, which results in a safer and tighter upper bound estimate.

To clearly present our methodologies and concerning experiments, the remaining thesis is organized as below:

- Part B - Quantifying Runtime Variability: Demonstrate the variability that an application may experience while co-running with others due to the contention on shared hardware resources in multi-cores. To better understand the organized experiments, we also present our used applications and stressing benchmarks.
- Part C - Architecture and Application Characterization: Present first the characterization methodology, and second the measurement techniques, and third experimental setup with all the experiment designs and implementations, and fourthly the experimental results which are splitted into the architecture section and the application section.
- Part D - Alternative Technique of WCET: Present first the methodology, and second the measurement techniques, and third the two detailed estimation methods with experimental results.
- Part E - Conclusion: Conclude the thesis and propose some future work based on our estimation results.

Part B

Quantifying Runtime Variability

Chapter B.1

Overview

In Chapter A.1 we explained that using multi-core COTS may bring about unsustainable **runtime variability** for co-running applications because of contention on shared hardware resources, which might not be accepted in safety-critical context. To better estimate this variability, we designed a group of experiments to quantify the variability, which shown us upto which extent the co-running performance can be slown down.

The concept of the quantification is to capture the runtime variability on a target application when it runs with a set of stressing benchmarks on the P4080. We first present applications under analysis for the experiments in Chapter B.2, and then introduce dedicated stressing benchmarks used as co-runners in Chapter B.3.Finally, we explain the experimental setup and analyse the results in Chapter B.4.

Chapter B.2

Applications under Study

B.2.1 Applications from Mibench Suite

As a proxy for various independent applications co-running on a safety-critical system, we used a subset of the MiBench benchmark suite [14], a set of embedded benchmarks from various domains of the embedded market: automotive, consumer, office, networking, security and telecommunication. Considering that we run the experiments on the bareboard P4080, we have to modify the Mibench benchmarks to be portable for barebone testing by eliminating operating system requirements such as system calls and dynamic memory management. We finally selected a subset of 7 Mibench benchmarks: ADPCM, CRC32, FFT, blowfish, SHA, patricia, and susan which are easier to be ported for barebone testing on the P4080 hardware platform. Although these seven benchmarks are not real-time as expected for the safety-critical domain, it is not an issue to apply them for characterizing the runtime variability.

- ADPCM (telecommunication): Adaptive Differential Pulse Code Modulation (ADPCM) is a variation of the well-known standard Pulse Code Modulation (PCM) that varies the size of the quantization step, to allow further reduction of the required bandwidth for a given signal-to-noise ratio.
- CRC32 (telecommunication): This benchmark performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are is an error-detecting code commonly used in data transmission.
- FFT (telecommunication): Fast Fourier Transform (FFT) on an array of data. Fourier transform converts digital signal from time space to frequency space to find the frequencies contained in the signal.
- blowfish (security): This benchmark is a symmetric block cipher with a variable length key.

- SHA (security): Secure Hash Algorithm (SHA) is a family of cryptographic hash functions used in the secure exchange of cryptographic keys and for generating digital signatures.

- patricia (network): Patricia tries are used to represent routing tables in network applications. This benchmark provides functions for inserting nodes, removing nodes, and searching in a Patricia trie designed for IP addresses and netmasks.

- susan (automotive): It is an image recognition package. It was developed for recognizing corners and edges in an image.

Figure B.2.1 shows the instructions (load/store, branches, integer and floating point) distribution of each Mibench benchmark, which can guide us to pre-understand their features before use them for our own purpose. For example, the benchmarks with low portion of load/store instructions will be not suitable for analysing the impact on the memory-path resources.

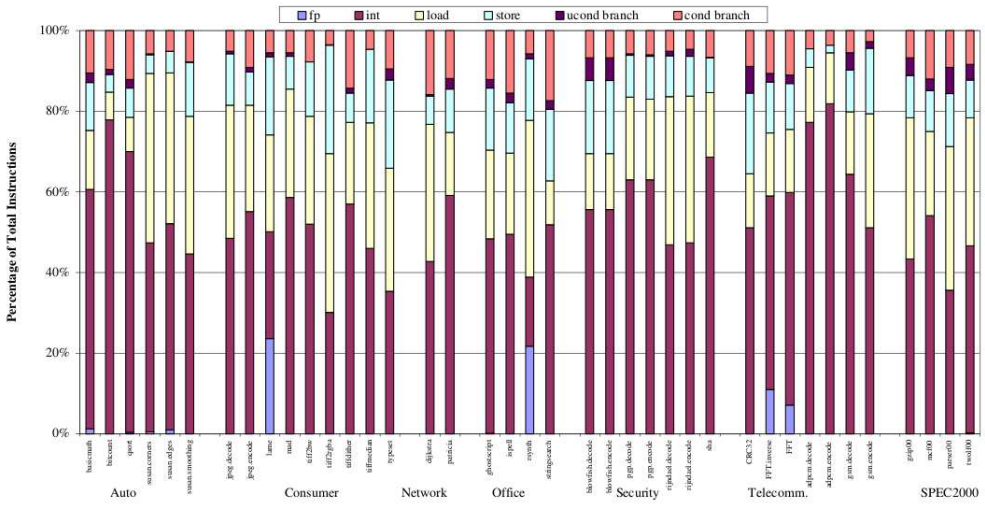


Figure B.2.1: *Distributions of main classes of instructions for each Mibench benchmark.*

B.2.2 Industrial Applications

We completed above suite of small benchmarks with two larger industrial-level applications developed internally at Thales including additional hard real-time constraints:

- an airborne radar application embedded in planes based on the Space-Time Adaptive Processing (STAP) algorithm [37] to detect the position and radial speed of another flying target despite the presence of ground-based or flying jamming devices.
- a pedestrian detection application based on the Viola & Jones shape recognition algorithm [36] to detect pedestrian on the security camera footage.

These applications were used in all the experiments throughout the thesis.

Chapter B.3

Resource Stressing Benchmarks

To evaluate the runtime variability caused by contention on shared hardware resources, we ran an application under monitor with a set of stressing benchmarks. The concept of stressing benchmark design is derived from article [28] that we stated in Section A.2.3. We designed stressing benchmarks which aim at stressing a single hardware resource by putting a high load onto it. However, stressing a single resource is sometimes not practically possible (like stressing the L3 cache without impacting the CoreNet in P4080), we are obliged to stress several resources at a time in this circumstance. Co-running an application with a particular resource stressing benchmarks allows us to produce a significant performance slowdown on the application due to conflicts on this resource.

Stressing benchmarks are directly written in assembly code to 1) let us maximize the master of the instructions to be executed and 2) prevent a compiler to change the main purpose of benchmarks by performing optimizations on them. We can thus stress a target resource in our expected way.

There were several different types of stressing benchmarks proposed in article [28], like the intra-core resources (e.g. pipeline level resources, private caches) and the inter-core resources (e.g. the last level cache, the interconnect). As we presented in Chapter A.3, we concentrated on e500mc cores and the shared memory architecture without using other perpherals in P4080. In addition, we insisted on mapping one benchmark into one core to keep co-running benchmarks totally isolated at the core-level. Accordingly, the major source of the performance variability for co-running applications comes from the contention on the shared memory-path, which allows us to restrict stressing benchmarks to stress shared memory-path resources: the CoreNet, the L3 caches and the DDR controllers.

The memory-path resource stressing benchmarks consist of a sequence of load-/store instructions that access to different cachelines in one L3 cache. Also, L3 cache access implies stressing access to the CoreNet. When the sum of the size of the array that co-running benchmarks traverse exceeds the L3 cache, the data

sets of all the co-runners do not fit in the L3 cache, which causes L3 cache misses forcing access to DDRs. In this circumstance, DDR controllers are stressed.

In addition to producing a fixed high load into one resource, we are able to tune the load by introducing some parameters in the stressing benchmark to modify resouce access pattern and frequency. These more specified stressing benchmarks are not necessary for variability quantification, but they are needed to characterize the architecture in the following part C. We thus present their details in chapter C.2.

Chapter B.4

Quantifying Runtime Variability

Before trying to estimate the runtime variability, we try to quantify this variability and the impact on the over margin.

B.4.1 Experimental Scenario

We used the platform P4080 to evaluate the runtime variability by co-running target applications respectively with stressing benchmarks on the bareboard P4080 without any operating system to minimize the runtime variability and to facilitate the use of hardware monitors as presented in Chapter A.3. In addition, we eliminated the preemption which has to be strictly controlled for safety-critical systems, by having each core running a unique benchmark. With this setup the contention on the shared memory-path becomes the major source of the runtime variability. The settings are used for all the experiments throughout the thesis.

B.4.2 Representing Runtime Variability Using Violin Plots

We are considering safety critical applications that requires to control their runtime variability to ensure that their worst execution time is below the hard real-time deadlines.

The runtime variability however, is not only characterized by a minimum and a maximum runtime. The statistical distribution of the runtimes also provides some useful information such as the rarity of the worst case, the distribution relatively to the median runtime, ...

To represent this distribution of runtimes, we relied on violin plots [17]. Violin plots are a method of plotting numeric data. They show the statistical distri-

bution of the data, which are a quick way of observing one or more sets of data graphically. One violin plot example is depicted in Figure B.4.1.

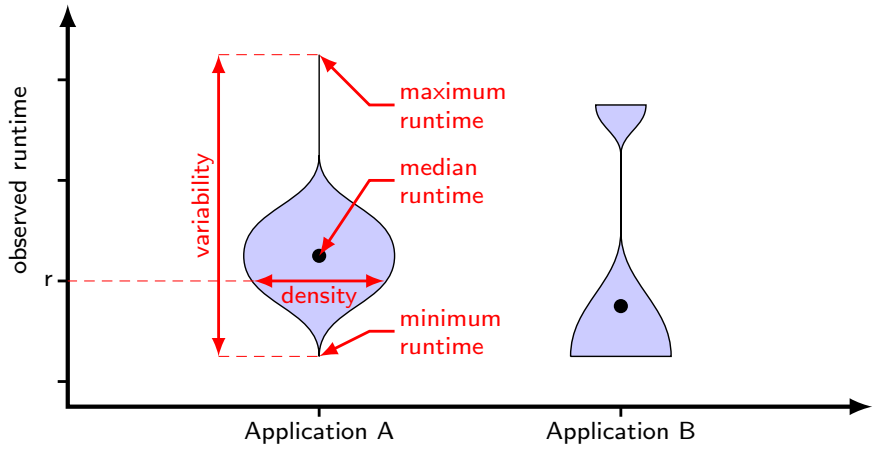


Figure B.4.1: *Example of violin plot to represent runtime distribution of two different applications*

Figure B.4.1 is composed of two violin plots providing some information about the runtime distribution of two applications A and B. For each application the variability is characterized by the bottom-most point and the top-most point that correspond to the minimum and maximum observed runtimes. The black dot in the violin represents the median runtime. Finally the width of the plot for a particular runtime (r in the figure) is proportional to the density of the runtime population with such a runtime.

We apply the violin plots to establish the distribution of measured execution times to graphically show the quantified variability.

B.4.3 Quantification Using Stressing Benchmarks

To quantify the runtime variability that a target application may experience due to the collision in shared resources, we designed experiments as below:

1. Measure the runtime of this target application using hardware monitors when it runs in isolation $\mathrm{RT}_{isolation}$. To capture the runtime variability, we repeat this measure several iterations to get a set of $\mathrm{RT}_{isolation}$, and we pick the median value of this set of $\mathrm{RT}_{isolation}$ as this application's standalone runtime.

2. Measure the runtime of the target application when it runs simultaneously with a set of stressing benchmarks $\mathrm{RT}_{corunning}$ in several iterations, and then

normalise all the obtained $RT_{corunning}$ by $RT_{isolation}$ to get a set of values of the performance slowdown.

3. Plot a violin plot using the set of values of the performance slowdown to graphically show its distribution, including the variability.

We quantified the runtime variability on nine applications presented in chapter B.2 when they respectively ran with the memory-path stressing benchmarks on the bareboard P4080.

Figure B.4.2(a) illustrates, with distribution violin plots, the runtime variabilty for nine applications running standalone on the P4080. To mimic a single-core configuration, the other cores are idle. In such a configuration the runtime variability represented by the height of the violin plots remains very low (below 0.1%).

However, when introducing 2 co-running stressing benchmarks, the runtime variability of each application around the previously computed standalone median runtime is increasing rapidly as depicted in Figure B.4.2(b). The impact on the average runtime is not significant, however, for the worst-case, we observe an average variability of 71% and a maximum variability of 361% for airborne radar.

Increasing the number of co-runners upto eight cores as depicted in Figure B.4.2(c) furthermore degrades the runtime variability, up to an average variability of 87% on the worst case and a maximal variability of 396% for airborne radar. Additionally, the average and minimal runtimes start to be impacted as well, with a variability of 54% on the average runtime.

Another study [22] from EADS also exhibits that using actual WCET analysis techniques for multi-cores would force the industry to multiply the WCET by a value close to the number of cores being used, providing no performance benefits over single-cores.

As considering the impact on such a large runtime variability would lead to unsustainable WCET margins far above the performance benefits of multi-core systems, it is critical to control this variability by providing a detailed characterization on the contention mechanism of the shared hardware resources and how each co-running application is behaving relatively to the shared hardware resources. Therefore, we proposed a methodology to achieve the architecture and application characterization in the next part, Part C.

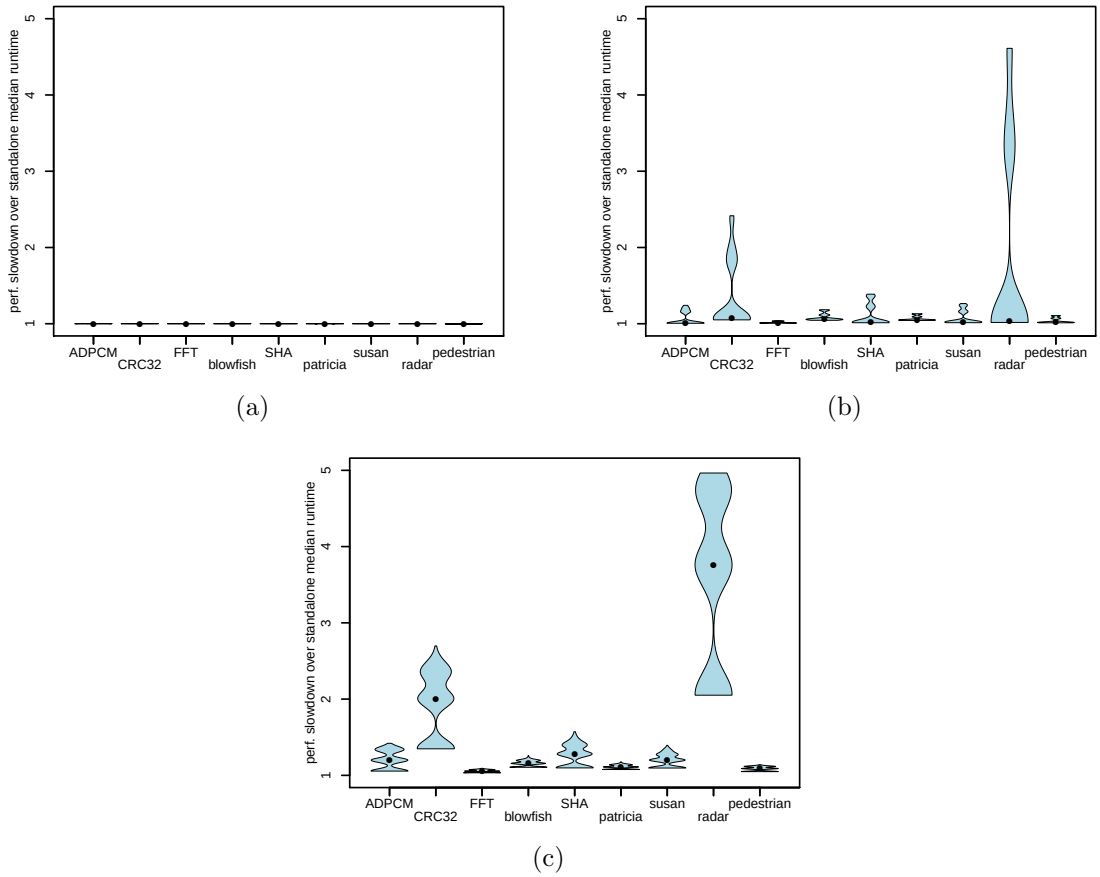


Figure B.4.2: *Runtime variability over 600 iterations of reference applications running (a) standalone, (b) concurrently with 2 benchmarks stressing the shared memory path, and (c) concurrently with 7 benchmarks stressing this resource.*

Part C

Architecture and Application Characterization

Chapter C.1

Characterization Methodology

To be able to control the variability quantified in Part B, we proposed a methodology characterizing the underlying architecture and applications which both behave like a gray- or black-box as we presented in Chapter A.1. Figure C.1.1 illustrates the concept of methodology. The charaterization consists on two steps: 1) Architecture characterization and 2) Application characterization.

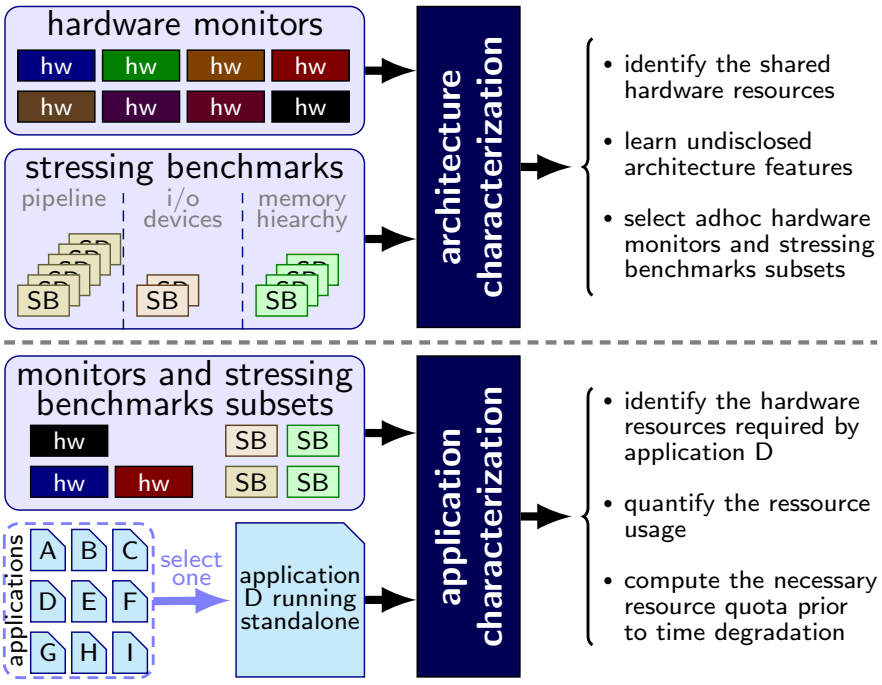


Figure C.1.1: *Overview of the analysis process*

Architecture Characterization

In the architecture characterization, we run different resource stressing bench-

marks in parallel to produce and tune a load on hardware resources under analysis. To observe the variation of performance according to different resource loads, we monitor and collect relevant events using hardware monitors during the execution. Thanks to stressing benchmarks and hardware monitors, this step is able to identify shared hardware resources, undisclosed features, like the contention mechanism of a shared resource, and select useful stressing benchmarks and hardware monitors which are responsible for the runtime variability.

Application Characterization
The purpose of the application characterization analysis is to understand why a particular application performance, and thus its execution time, varies when the application is running with other applications in the same multi-core. Therefore, we first run an application with a set of shared resource stressing benchmarks which have been selected in the previous architecture characterization to learn which shared resources such application is sensitive to. Second, we monitor and collect the events related to sensitive shared resources during the standalone run of the application to compute each shared resource usage of the application.

In the remaining of the part, we first present two measurement techniques - selected hardware monitors and designed stressing benchmarks. Second, we explain how to organise experiments using these two techniques to achieve characterization objectives. Third, we present the implementation in detail, especially how to automate experiments. Fourth, we demonstrate and analyse the experimental results. Last, we make a conclusion about the characterised features of the architecture and the applications.

Chapter C.2

Measurement Techniques

To evaluate the methodology, we used two measurement metrics: *Hardware Monitors* and *Stressing Benchmarks*.

C.2.1 Hardware Monitors

We used the setup presented in Section B.4.1 for the characterization, which means that all the performance slowdown should be mainly caused by interferences of co-running applications on shared hardware resources - the CoreNet, the L3 caches and the DDR controllers. So instead of monitoring all the available events in the P4080, these three shared resources concerning events are necessary to characterize the architecture and applications.

Table C.2.1 lists the main events used during the characterization and corresponding descriptions. As we explained in Section A.3.3, the CoreNet (including the CPC (L3 cache)) monitoring is not documented in the reference manual. We are thus not able to monitor the concerning events.

Event	Description
CPU CYCLES	Core related event counting the number of processor cycles. Event number is *Ref:1* in Table A.3.5.
BIU master requests	Core related event counting the number of CoreNet transactions. Event number is *Com:67* in Table A.3.5.
DDRr	Platform related event counting the number of DDR read transactions.
DDRw	Platform related event counting the number of DDR write transactions.

Table C.2.1: *The main events used in the characterization*

C.2.2 Stressing Benchmarks

To better characterize concurrent accesses to shared hardware resources, a large set of stressing benchmarks are defined, each dedicated at stressing a particular potentially shared hardware resource. Since the memory-path is the main source of the runtime variability, stressing benchmarks are restricted to produce a high load to the memory-path resources - the CoreNet, the L3 caches and the DDR controllers.

The overall concept of stressing benchmarks has been presented in chapter B.3. In this section, we present the detail of stressing benchmarks used for the characterization. The sample code listed in Listing C.2.1 shows a general framework of memory-path stressing benchmark with its parameters **TABLESIZE**, **STRIDE**, **OPERATION**, **NOP** and **UNROLLED**. We shown the source code of a stressing benchmark example in the Appendix (see Chapter F).

Listing C.2.1: *General framework of memory-path stressing benchmarks*

```
LOOP (i=0;i<TABLESIZE-1;i+=STRIDE*UNROLLED)
{
        ACCESS_TABLE(i,OPERATION)
        NOP
        ACCESS_TABLE(i+STRIDE,OPERATION)
        NOP
        ...
        ACCESS_TABLE(i+(STRIDE*(UNROLLED-1)),OPERATION)
        NOP
}
```

The following describes the parameters in Listing C.2.1:

- **TABLESIZE**
  It defines the size of the table that the benchmark traverse within one loop. We regulate this parameter to decide which resource along memory-path will be stressed. For instance, on the P4080, a benchmark with TABLESIZE less than private L2 cache 128KB will not stress the CoreNet, the L3 caches and the DDR. However, if we set TABLESIZE between the L2 cache size and the L3 cache size, we will stress both CoreNet and L3 cache without touching DDR. To stress one DDR controller, we increase TABLESIZE upto more than one L3 cache.

- **STRIDE**
  It defines the distance of elements between two consecutive table accesses. This parameter is essential for cache usage. We regulate STRIDE to modify the number of accesses hitting the same cache line, which changes the number of accesses physically communicating with the interconnect and the

main memory. The worst case for cache locality is to use each cache line only once, which maximizes the physical access to the interconnect and the memory. For example, the cache line size in P4080 is 64 bytes and we have a table of type integer which is 4 bytes. If we hit different cache line for each access, STRIDE should be set to more than 16. Figure C.2.1 shows the cache usage with different STRIDE.

- **OPERATION**
  It defines table access type, either read or write. Different combination of two operations makes different accessing latency and traffic load.

- **NOP**
  It defines some idle time between two consecutive table accesses. This parameter is primarily used to regulate the access frequency, namely to regulate the traffic workload on the interconnect.

- **UNROLLED**
  The loop is unrolled into a number of accesses defined by this parameter. This loop unrolling helps to reduce the frequency of branches and loop maintenance instructions, which accordingly increases the frequency of table access.

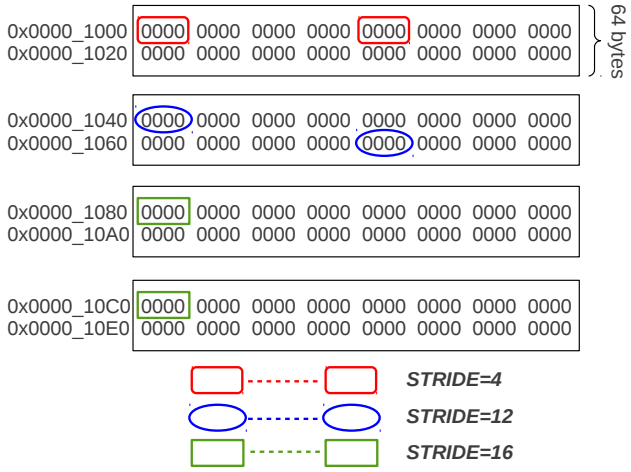


Figure C.2.1: *The cache access pattern with different STRIDE (one cache line=64bytes)*

Chapter C.3

Experimental Setup

To evaluate the proposed characterization methodology, we designed a set of experiments using the measurement techniques presented in Chapter C.2. This chapter presents how we organized the experiments to achieve corresponding characterization objectives. In addition, we quantified the size of the design space according to the methodology, and explained how to reduce it. The experimental setup is the same as we presented in Section B.4.1.

C.3.1 Architecture Characterization

C.3.1.1 Identifying Shared Hardware Resources

To identify shared hardware resources, we run a particular resource stressing benchmark on the P4080 in isolation to collect the runtime represented by CPU CYCLES using hardware monitors. This stressing benchmark is then replicated up to the number of available cores to collect the runtime again. Comparing these runtimes allows to confirm if this stressed particular resource is truely shared (a runtime slowdown implicates a shared resource), while reproducing this study for every potentially shared resource enables to identify each effectively shared resource. During the identification, useful hardware monitors and stressing benchmarks can be selected according to truely shared resources. This study is an useful complement to the information appearing in the user manual (cache size and sharing level). The results are presented in Section C.5.1.

C.3.1.2 Identifying Undisclosed Features and the Shared Resource Availability

After identifying shared hardware resources, we learn undisclosed architecture features relatd to shared resources by varying the mapping of co-running stressing benchmarks and the parameters of stressing benchmarks presented in Section C.2.2.

Identifying Undisclosed Features
Our experimental P4080 platform features a complex CoreNet interconnect to connect all the cores to two L3 platform caches, each connected to a DDR controller. The undisclosed topology of this interconnect has a significant impact on how the memory traffic of one core will interfere with the traffic of other cores, which means that different core mappings will bring different runtime variability. Therefore, varying the mapping of co-running stressing benchmarks and monitoring the performance under different mappings allows us to reveal this phenomenon. The results are presented in Section C.5.2.

Quantifying the Shared Resource Availability
To learn the CoreNet availability, we modify the parameter NOP of co-running stressing benchmarks to tune the CoreNet access load allowing to study correlation between the performance and the CoreNet load, which deducts the maximum throughput and the saturation behavior of the CoreNet that are not clearly documented. Furthermore, mapping the CoreNet stressing benchmarks into different cores allows to learn the CoreNet topology. We get the CoreNet load by monitoring the event *BIU master requests* in all the co-running cores. The concept of the DDR quantification is the same with the CoreNet, but with the DDR monitoring to get DDR accesses. The results for the CoreNet and DDR evaluations are presented in Section C.5.5.

C.3.1.3 Identifying the Optimal Configuration

The P4080 supports different configuration modes as explained in Section A.3.4. Considering that different configurations may result in different performance, the optimal configuration for the safety-critical context is thus identified to respect two criteria: 1) predictability, ensuring low performance variability, and 2) sufficient minimal performances assuring the worst case execution time will be below the application deadlines of the hard real-time system. However, it is not obvious to intuitively infer the most suitable configuration. For instance, partitioning can lower the runtime variability while taking a risk of degrading overall performance below acceptable throughput as we explained in Section A.3.4, and activating the

second L3 cache / DDR controller only makes sense if it does not compromise the predictability of the interconnect.

We have selected four configuration candidates described in Table A.3.7 in Section A.3.4 to be challenged for the performance and predictability. Figure C.3.1 details each configuration structure. The experiments realized to determine the optimal configuration is described in Section C.5.3.

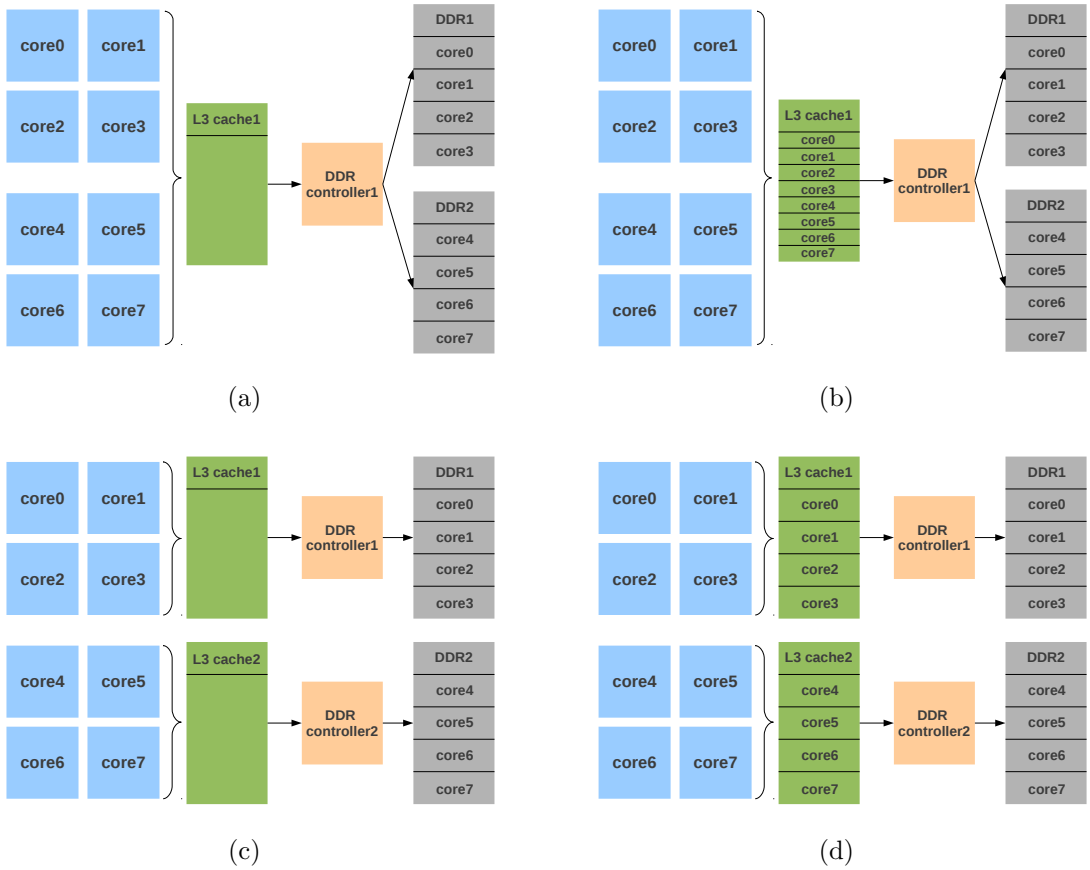


Figure C.3.1: *Selected configurations of P4080 (a) single controller non-partitioned, (b) single controller partitioned, (c) dual controller non-partitioned and (d) dual controller partitioned.*

C.3.2 Application Characterization

As the runtime variability is mostly due to the conflict on concurrent shared hardware resources accesses, it is critical to identify which of such resources each

application requires, as well as to quantify the amount of the resource usage by the application.

C.3.2.1 Identifying Sensitive Shared Resources

To identify which of the shared resources an application is sensitive to, we run each application with every shared resource stressing benchmark from the subset identified by the architecture analysis, gathering the runtime information using hardware monitors. Application performance slowdowns allow us to identify to which resources this application is sensitive as well as the sensitivity of the application to the resources. The results are presented in Section C.6.2.

C.3.2.2 Capturing the Shared Resource Usage

A resource usage allows to evaluate the minimal share of this resource the application requires. To compute each shared resource usage, we monitor sensitive shared resource concerning events for a standalone application and normalise the value of each resource event by maximum throughput of this resource quantified in the architecture characterization. The obtained percentage value is the usage of this resource. The results are presented in Section C.6.3.

C.3.2.3 Determining Possible Co-running Applications using Resource Usages

Finally, repeating the above processes on every application would allow us to identify applications likely to run concurrently on the system without exceeding the maximum throughput of sensitive shared resources, which means that individual worst-case execution times would not significantly degrades. The results are shown in Section C.6.4.

C.3.3 Design Space

Considering the available number of hardware monitors (~ 200), the total number of stressing benchmarks (~ 1000), and the total number of possible mappings on an 8-core architecture, the experimental space of the methodology presented in Chapter C.1 can be quite large.

Let A be the number of applications to characterize, S the number of stressing benchmarks, M the number of available hardware monitors, C the number of cores, I the number of iterations of repeating each experimental scenario, and N

the number of monitor each core is able to measure at once. The total number of possible experiments is:

$$\frac{1}{N}(AMC(S+1)^{(C-1)} + SMC(S+1)^{(C-1)})I$$

Considering that average execution time of a single experiment to be 100ms, and the order of magnitude for the values presented in Table C.3.1, it would requires us 10^{19} years to exhaust such a design space.

Applications	A	9
Stressing benchmarks	S	1000
Hardware monitors	M	200
Cores	C	8
Iterations	I	100
Performance monitor registers	N	4

Table C.3.1: *Order of magnitude of the design space*

To deal with such a design space we setup an automatic framework which is explained in Section C.4.3.2. As depicted in Figure C.1.1 we perform an overall architecture characterization prior to performing the application characterization.

Beyond the characterization aspect presented in Section C.3.1, the architecture characterization phase allows us to filter out unnecessary hardware monitors and stressing benchmarks by identifying those that are not related to the performance variability. As we explained in Chapter C.2, we only used the hardware monitors and stressing benchmarks related to the CoreNet, the L3 cache and the DDR controllers taking account of our scenario restriction (one application per core and restricted on shared memory-path without the use of other peripherals) and documented resource features (L1 and L2 caches are both private).

We can learn from the design space formula that the decisive factor of the order of magnitude is the number of possible mappings. We will show how to reduce the design space in terms of the mapping and also in other respects in Chapter C.5.

Chapter C.4

Implementation

C.4.1 Measurement Framework

We have presented in Chapter A.3 how to use PMRs in the e500mc and platform debug facilities to respectively monitor core-related and platform-related activities. In this section, we only presented the measurement framework according to the flowchart of using hardware monitors shown in Figure A.3.2.

Listing C.4.1 shows the measurement framework using hardware monitors.

Listing C.4.1: *Measurement framework using hardware monitors*

```
LOOP (i=0;i<N;i++)
{
        FREEZE_Counter();
        INITIALIZE_Counter();
        CONFIGURE_Counter();
        UNFREEZE_Counter();
        {Benchmark execution...}
        FREEZE_Counter();
        COLLECT_Counter();
}
```

In Listing C.4.1, all the procedures from FREEZE_Counter to COLLECT_Counter respect the flowchart A.3.2 except that the loop indice N is a new parameter that we have not seen. The parameter N limits the iterations of the measurement repeated to capture the variability of collected events.

C.4.2 Synchronization of Multi-cores Using the Interprocessor Interrupt (IPI)

To quantify the CoreNet availability during the architecture characterization, we need to collect the CoreNet transactions represented by the event *BIU master requests* in all the co-running cores to get the total load of the CoreNet. Considering that each core runs independently/asynchronously with others, we can not make sure that each core stops simultaneously to store hardware monitor counters. Therefore, we need a synchronization mechanism to manage hardware monitor information collection.

There are three methods to realise the intercore interrupt in the P4080: the interprocessor interrupt (doorbell) method, the messaging-interrupt method and the shared-message, signaled-interrupt method. We used the interprocessor interrupt method to achieve the synchronization. The interprocessor interrupt approach is preferred when a cluster of cores works asynchronously and one core may signal the other core(s) with unpredictable timing. The interprocessor interrupt is managed by the Multicore Programmable Interrupt Controller (MPIC) which is responsible for receiving hardware-generated interrupts from different sources (both internal and external), prioritizing them in the context of interrupts that are generated from within the MPIC (such as messaging, timer, and interprocessor interrupts), and delivering them to the appropriate destination for servicing.

The doorbell method can generate an event from a single core to a single core or from a single core to multi-cores. Figure C.4.1 depicts two example diagrams of using doorbell in these two contexts.

C.4.2.1 Use of the Interprocessor Interrupt

The use of the interprocessor interrupt is achieved by configuring relevant registers in the MPIC map. There are several common registers for all the intercore interrupt methods described in the following:

- Processor core Who Am I (WHOAMIn): There is one WHOAMI per core. It can be read by a processor core to determine its physical connection to the MPIC. The value returned when reading this register may be used to determine the value for the destination masks used for dispatching interrupts.
- Global Configuration Register (GCR): It controls the MPIC's operating mode, and allows the software to reset the MPIC. There are three available modes to be selected, the mixed mode should be set for the intercore interrupt.

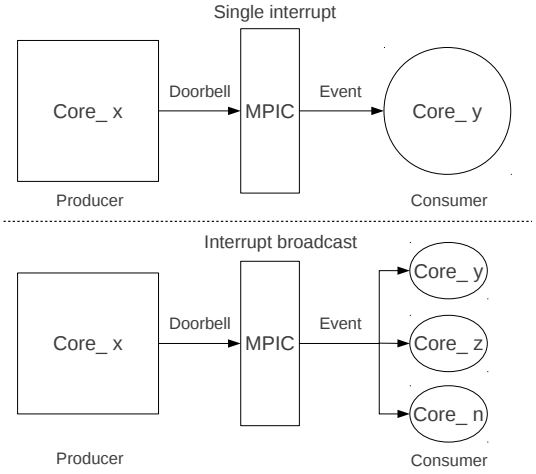


Figure C.4.1: *Examples of using doorbell. Top: single core to single core; Bottom: single core to multi-cores*

- Processor core Current Task Priority Register (CTPR): There is one CTPR per core It allows each core to specify the priority of task that will be allowed to interrupt. The MPIC uses this value for comparison with the priority of incoming interrupts. Given several concurrent incoming interrupts, the highest priority interrupt is asserted to the core.

- Processor core Interrupt Acknowledge registers (IACKn): Each processor core has an IACK register assigned to it. In the mixed mode, when the MPIC causes *int* to be asserted, the external interrupt service routine acknowledges the request by reading that cores processor core IACK, which at this point holds the 16-bit vector value for the interrupt source that generated the request. Reading IACK negates the *int* signal to that core, making it possible for another interrupt source to signal an external interrupt to the core.

- End-Of-Interrupt (EOIn): There is one EOI per core. It is a write-only register that notifies the MPIC that servicing of the currently-active interrupt is complete.

In addition to these common registers, there are registers specific to the doorbell method:

- Interprocessor Interrupt Vector/Priority Registers (IPIVPRn): IPIVPRs contain the interrupt vector and priority fields for the four interprocessor interrupt channels. There is one vector/priority register per channel. One of them must be programmed with the interrupt vector and priority.

- Interprocessor Interrupt Dispatch Registers (IPIDRn): There are four IPIDRs, one for each interprocessor interrupt channel. Writing to an IPIDR with a bit set causes an interrupt to the target-core device. These registers are accessible through the per-CPU private address or by external devices, through the core-specific, global address allowing external bus masters to use simple mechanisms to generate interrupts to a selected core. Reading the chosen core(s) IACK register clears the interrupt so that when interrupts are re-enabled the same IPI is not seen again.

The procedure of using the interprocessor interrupt to signal an interrupt from the core0 to core1 is explained below:

1. Core0 reads WHOAMI0 and Core1 reads WHOAMI1 to identify their role.
2. Core0 writes GCR to program the mixed mode.
3. Core0 writes IPIVPR0 to unmask interprocessor interrupt 0 and to program a priority and vector ID.
4. Core1 writes CPTR1 to adjust the interrupt priority task to a value less than IPIVPR0, which lets the incoming interrupt occur (this can be done by Core0 or Core1).
5. Core0 writes IPIDR0 to signal the interrupt to Core1.
6. Core1 reads IACK1 to negate *int* and get the interrupt vector ID.
7. Core1 writes zero to EOI1 to clear the IPI and signal the end of the interrupt service.

C.4.2.2 Framework of Synchronizing Hardware Monitor Collections Using IPI

Let take the core0 and core1 as example to show the framework synchronizing hardware monitor collections. The core0 signals the core1 with the collection timing. The framework is illustrated in Figure C.4.2.

The core0 dispatches the IPI signal to the core1 to inform the timing of collecting hardware counters. Once the latter receives the IPI signal, it calls the IPI interrupt routine where it freezes the hardware counters for collecting them and then initializes them for the next iteration.

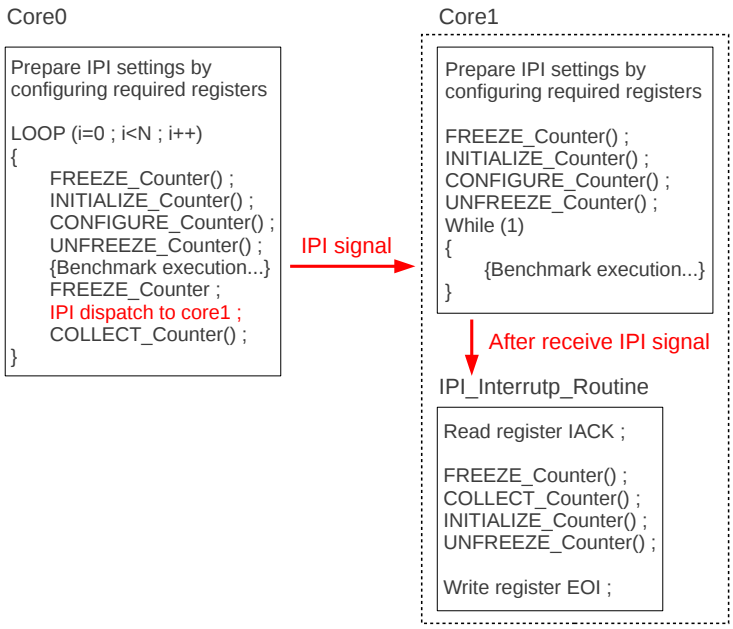


Figure C.4.2: *The framework of synchronizing hardware monitor collections.*

C.4.3 Software Development Using CodeWarrior

In Section C.3.3, we stated the huge design space that we may experience during experiments. Therefore, to cope with this huge space, we developed a framework automating experiments. The automatic framework is composed of two steps: 1) automate the execution of a single experiment within the CodeWarrior. 2) automate a series of experiments outside the CodeWarrior. In Chapter A.4, we presented the command line mode for compilation and debugging in the CodeWarrior. The command line mode provides us the opportunity to achieve the automatisation.

C.4.3.1 Automating the Debugging Session for a Single Experiment within CodeWarrior

As all the experiments were run on baremetal P4080, there was no file system to support functions related to file operations in C programming, which prevented us to write the data directly in a destination file in C. To overcome this limitation, we wrote all the hardware monitor data in the P4080's memory for temporary storage during the application execution and then exported the data from the memory

into destination files in the end of execution using the debugger shell command *save*. To export the data, we debugged the target application and saved the data from a specified memory range into .txt files by stopping the debugging once the execution finished. To automatically detect the end of application execution when all the data are stored in the memory, the application displayed a keyword "over" in the end of execution in a hyper terminal (we used PuTTY) in our host computer. We configured the PuTTY to make the keyword written into a log file with overwrite mode once it appeared in the hyper terminal. Therefore, we were able to detect the end of execution by keeping the log file under surveillance.

To automate this detection and save processes during the debugging session, we used the debugging commands and TCL script supported in the CodeWarrior. The framework of the script is described in Figure C.4.3.

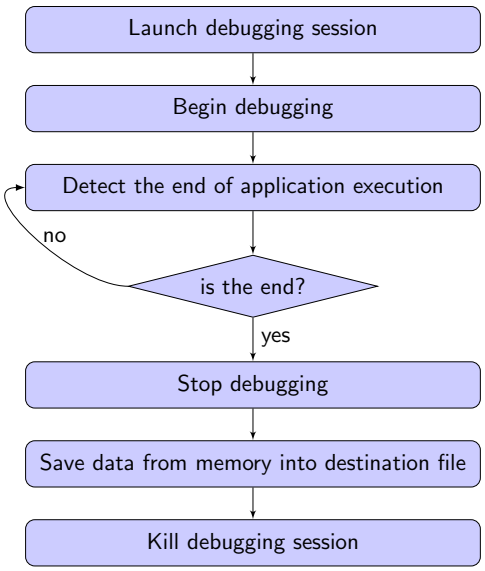


Figure C.4.3: *The flowchart of automating debugging session of a single experiment within CodeWarrior.*

This debugging framework is an important part of the whole automatical framework that we present in Section C.4.3.2. The source code of a TCL script example is shown in Appendix (see Chatper F).

C.4.3.2 Automating Experiments outside CodeWarrior

The automatical framework is shown with a flowchart in Figure C.4.4 which includes the above automatical debugging session. We used the Python script language to program the framework in the thesis. The source code of a Python script example is shown in Appendix (see Chapter F).

As shown in Figure C.4.4, we had three main processes in the automatization: the generation process, the configuration process and the execution process.

C.4.3.2.1 Generation Process

The objective of this process is to generate experimental scenarios that we want to automatically execute in one script. The scenarios are organized by cores as an example illustrated in Listing C.4.2.

Listing C.4.2: *Example of framework of scenario generation process*

```
scenario[i] =
{'core0': {'BM':          'true',
           'APP':         'true',
           'NAME_APP':    APP,
           'SAVE':        APP_APP+1SB},
 'core1': {'BM':          'true',
           'APP':         'false',
           'TABLESIZE':   tablesize,
           'STRIDE':      stride,
           'OPERATION':   write,
           'NOP':         nop,
           'SAVE':        SB_APP+1SB},
 'core2': {'BM':          'false'},
 'core3': {'BM':          'false'},
 'core4': {'BM':          'false'},
 'core5': {'BM':          'false'},
 'core6': {'BM':          'false'},
 'core7': {'BM':          'false'}
}
```

Each scenario is composed of eight cores which represent eight cores in the P4080. In each core, we designed a series of parameters specifying some features of the benchmark mapping in this core. These features would be used to configure the scenario in the following configuration process. The parameters appearing in Listing C.4.2 are explained below:

- BM: It defines if there is a benchmark in the core. 'false' means there is no benchmark in the core, so the core is idle. 'true' means there is a benchmark in it, either an application or a stressing benchmark.
- APP: If BM is set to be 'true', we use APP to indicate if the existing benchmark is an application. 'true'-application, 'false'-stressing benchmark.

- NAME_APP: It gives the name of the application running in the core. It will be set only if APP is set to be 'true'.
- SAVE: It defines the name of the destination file where we will save the scenario's output data, namely hardware monitor information.
- TABLESIZE, STRIDE, OPERATION and NOP: They are all stressing benchmark related parameters that we have explained in Chapter C.2. These parameters will be set only if APP is set to be 'false'. We will obtain a desired stressing benchmark by modifying the source files according to these parameters in the configuration process.

Listing C.4.2 is just an example of scenarios generation. We can add other parameters in each core to further specify the configuration of benchmarks.

C.4.3.2.2 Configuration Process

The objective of this process is twofold: to modify source files and to compile modified projects. The generated scenarios behave as input arguments of the configuration process, which allows us to use each scenario's pre-defined parameters to modify source files in corresponding core's project which has been created prior to the automatization. After projects' modifications, we compiled all the modified projects using command *build* provided by command tool *ecd.exe* which is located in CodeWarrior installation folder as we introduced in Section A.4.2. The command is shown below:

ecd.exe -build -verbose -project PATHofProject -cleanBuild

where PATHofProject is the project that we want to compile.

All the modified source files and compilation output files were additionally saved in a pre-selected path in our host computer, which helped trace backward in case we got an incorrect experimental result.

C.4.3.2.3 Execution Process

The objective of this process is to automate debugging scenarios as we explained in Section C.4.3.1. The first thing that we should do is to create a TCL debugging script according to each scenario's pre-defined parameters. For example, to automatically debug the scenario[i] defined by Listing C.4.2, we need to create a TCL script to first debug a project named APP in core0 and a project of stressing benchmark in core1 which has been already configured and compiled in the configuration process, and second to save the hardware monitor data from core0 and core1 into destination file APP_APP+1SB.txt and SB_APP+1SB.txt, and

finally we quit CodeWarrior using the debugger shell command *quitIDE*. After create desired TCL script, we execute this script within the CodeWarrior using command line tool *cwide.exe* located in CodeWarrior installation folder as:

cwide.exe -vmargsplus -Dcw.script="PATHofTCLscript"

where "PATHofTCLscript" is the TCL script we want to execute in CodeWarrior. The TCL script was additionally saved in our host computer, which helped trace backward in case we got an incorrect experimental result.

However, we met an unexpected problem about the use of CodeWarrior. The CodeWarrior IDE sometimes loses the connection with the target platform P4080 without any warning and we can not diagnose and repair this connection error in command line. To not interrupt our automatization, the only solution is to restart the CodeWarrior IDE and repeat the unfinished scenario. To detect the collapse of CodeWarrior, we create a background thread in parallel with the execution process. The thread behaves as a timer configured with a threshold to count the elapsed time from the start of CodeWarrior until it quits. If the elapsed time is more than the threshold, we consider that the CodeWarrior collapsed and should be restarted. We accordingly kill the process CodeWarrior by force and restart the execution of current unfinished scenario. If the elapsed time is less than the threshold, CodeWarrior quits normally and we cancel the thread prior to the execution of the next scenario. We should pay attention to the value of threshold. This waiting time has to be sufficiently long to let every scenario finish properly.

Thanks to the experiment automatization, we are able to easily organize hundreds of or thousands of scenarios and execute them with one click "return". The only thing we have to do is to fetch result data files from the destination folders.

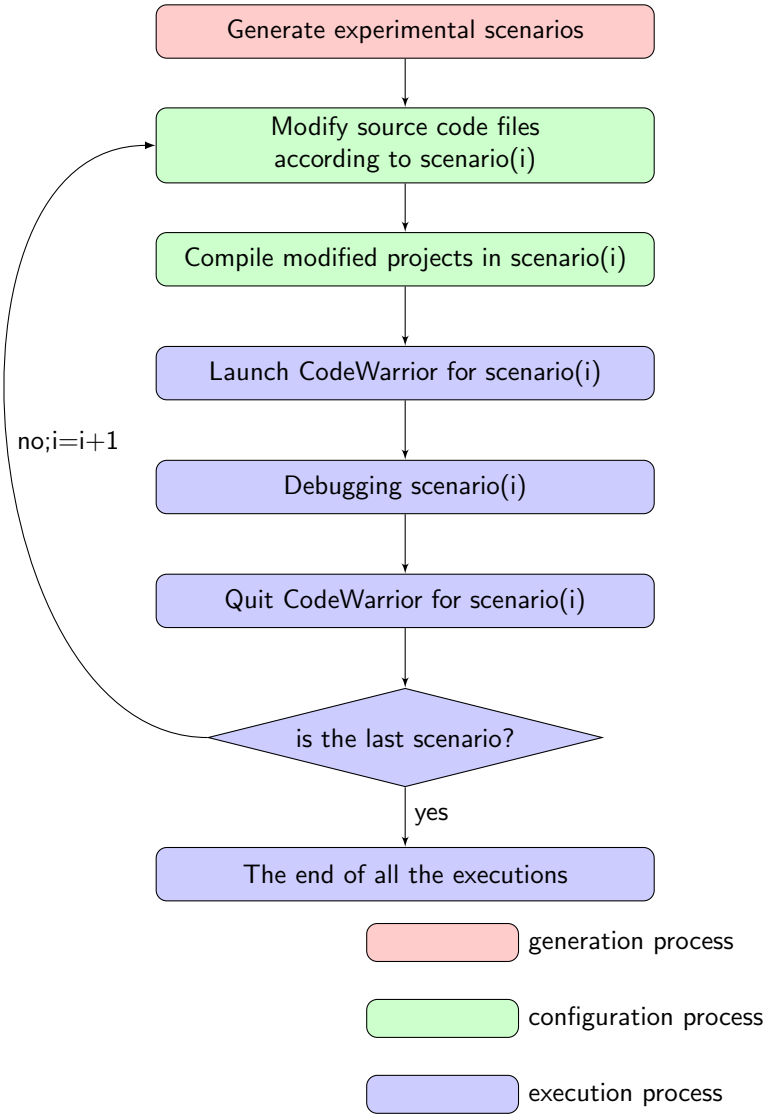


Figure C.4.4: *The flowchart of automating experiments outside CodeWarrior.*

Chapter C.5

Architecutre Characterization Results

We have quantified the potential variability caused by the contention on shared hardware resources among co-running benchmarks on the P4080 in Chapter B.4. Before analysing the contention mechanism of shared resources to control the variability, we first identified which hardware resources are effectively shared in the underlying architecture.

C.5.1 Identifying Shared Hardware Resources

To identify the shared hardware reources, we ran the experiments as presented in Section C.3.1.1. Despite the privacy of L1 and L2 caches, we still co-ran stressing benchmarks related to them to demonstrate the impact of these private resources on the performance variability, which can be used to compare with the impact of the shared resources. We used the default platform configuration appearing in the first row of Table A.3.7 to run:

- first a L1 data cache stressing benchmark in isolation in the core #1, and second a L1 data cache stressing benchmark in the core #1 with seven same stressing benchmarks in remaining cores.
- first a L2 cache stressing benchmark in isolation in the core #1, and second a L2 cache stressing benchmark in the core #1 with seven same stressing benchmarks in remaining cores.
- first a L3 cache stressing benchmark in isolation in the core #1, and second a L3 cache stressing benchmark in the core #1 with seven same stressing benchmarks in remaining cores.

We measured the runtime by monitoring the event CPU CYCLES in the core #1 . For each above experimental scenario, we computed the performance slowdown of the core #1 by normalising the co-running runtime by the standalone runtime as we detailed in Section B.4.3. Figure C.5.1 depicts the performance variability of the core #1 in above three circumstances.

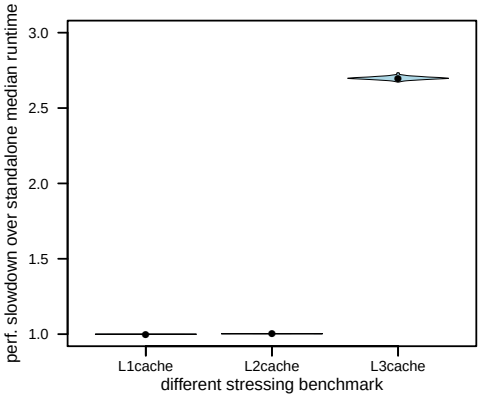


Figure C.5.1: *The runtime variability of the core #1 while 8 co-running L1 data cache stressing benchmarks, 8 co-running L2 cache stressing benchmarks and 8 co-running L3 cache stressing benchmarks.*

We can infer from Figure C.5.1 that co-running L1 data cache and L2 cache stressing benchmarks do not impair the performance of the core #1 while L3 cache stressing benchmark results in a performance slowdown up to 270%. We can thus experimentally confirm that the L1 data cache and L2 cache are effectively private and L3 cache is shared as presented in the P4080 manual.

As we stated in Section C.3.3, we can reduce the design space by filtering out hardware monitors and stressing benchmarks related to L1 data cache and L2 cache which are not responsible for the performance variability.

C.5.2 Identifying Undisclosed Features

The architecture characterization phase is an opportunity to learn about the undisclosed features and mechanisms. More particularly, we identified the impact of the core mapping on the runtime variability as we presented in Section C.3.1.2.

To characterize the core mapping, we used the default platform configuration appearing as first row of Table A.3.7. By running and monitoring various stressing benchmarks within this configuration, we managed to figure out that the eight cores are organized as two clusters of four cores.

The experiment illustrating this cluster effect is depicted in Figure C.5.2. It shows the runtime variability of running three instances of a particular stressing

benchmark, mapping the first instance on core #1, while varying the mapping of the other two instances on the other available cores side by side. We measured the runtime of the core #1 by monitoring the event CPU CYLCES, and computed the performance slowdown for each experimental scenario. The different scenarios were organized as the following:

- Mapping(1,2,3): mapped three stressing benchmarks in core #1, core #2 and core #3.
- Mapping(1,3,4): mapped three stressing benchmarks in core #1, core #2 and core #4.
- Mapping(1,4,5): mapped three stressing benchmarks in core #1, core #4 and core #5.
- Mapping(1,5,6): mapped three stressing benchmarks in core #1, core #5 and core #6.
- Mapping(1,6,7): mapped three stressing benchmarks in core #1, core #6 and core #7.
- Mapping(1,7,0): mapped three stressing benchmarks in core #1, core #7 and core #0.
- Mapping(1,0,2): mapped three stressing benchmarks in core #1, core #0 and core #2.

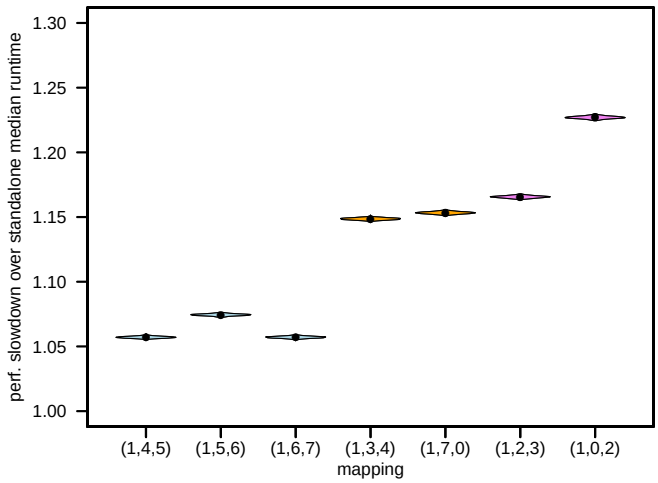


Figure C.5.2: *Runtime variability while mapping three instances of a stressing benchmark on different cores.*

If we classify eight cores into two clusters: Cluster1(core #0 to core #3) and Cluster2(core #4 to core #7), three different distributions can be identified in the figure:

- The first one, corresponding to the first three violin plots, corresponds to mapping the monitored instance alone in Cluster1 while running the two other instances on the other cluster. It exhibits only a small performance slowdown with a maximum of +7.6% and relatively small variance among the three mappings.
- The second distribution, illustrated by the next two violin plots, corresponds to mapping the monitored instance in the same cluster with one of the two other instances and shows a bigger performance slowdown with a maximum of +15.6% and low variability.
- Finally, the third distribution, illustrated by the last two violin plots corresponds to mapping all three instances in the same cluster and exhibits the largest performance slowdown and an important variability (from +16.3% to +23.0%).

We summarize above three types of mapping in Figure C.5.3.

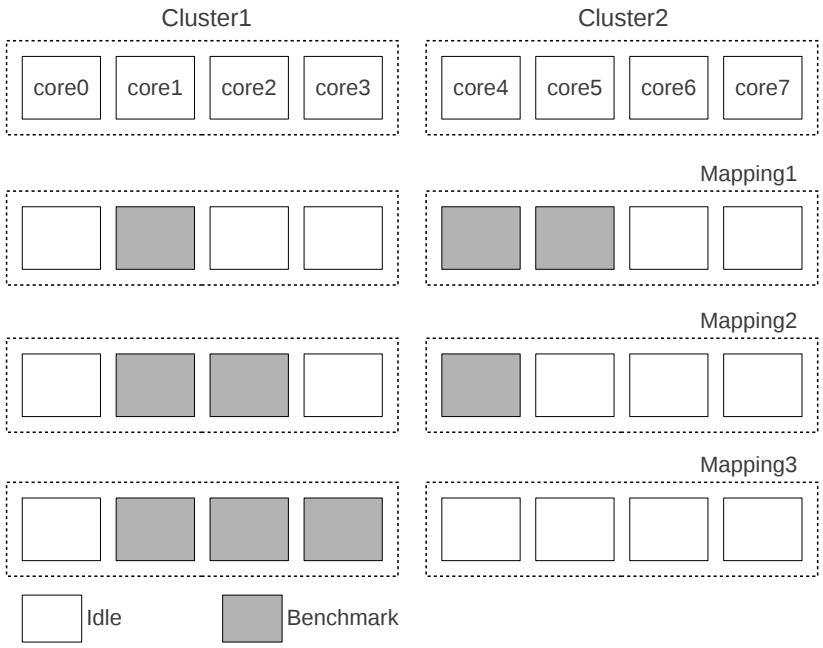


Figure C.5.3: *Three types of mapping under the 4-core cluster effect.*

We generally describe three mappings as below:

- Mapping1: we try to avoid to map the co-running benchmark(s) in the same cluster with the benchmark under monitor.
- Mapping2: we try to balance the total co-running benchmarks in two clusters.
- Mapping3: we try to map the co-running benchmark(s) in the same cluster with the benchmark under monitor.

As a conclusion, due to the cluster effect, the runtime variability of applications is placement dependent for the P4080 platform. Mapping1 brings the smallest performance slowdown and the lowest variability. However within each 4-core cluster the performance does not depend on the placement, enabling us to reduce the number of mapping to be tested, and therefore allowing us to reduce the overall design space.

C.5.3 Identifying the Optimal Configuration

In Section C.3.1.3 we identified several hardware configurations presented in Table A.3.7. Selecting the most appropriate configuration for safety-critical applications is not straightforward as two criteria: the low variability and the sufficient minimal performance, have to be maximized presented in Section C.3.1.3.

To evaluate these different hardware configurations, we designed a set of stressing benchmarks dedicated at stressing the different shared hardware resources along the memory path including the CoreNet interconnect, the L3 caches and the DDR controllers.

We measured the runtime of stressing benchmark running on core #1 by monitoring the event CPU CYCLES, and we varied the number of stressing benchmarks running on the remaining cores. For two configurations in the first two rows of Table A.3.7, we identified Mapping1 bringing the smallest performance slowdown and the lowest variability due to 4-core cluster effect in the previous section. For other two configurations in the last two rows of Table A.3.7, we configured core #0 to core #3 attached to one L3 cache and its associated DDR controller and core #4 to core #7 attached to the other L3 cache and its associated DDR controller, which artificially separated eight cores into two clusters as two former configurations. To fulfill the low variability for safety-critical systems, we selected Mapping1 to map varied number of stressing benchmarks in the remaining cores for four configurations as shown below:

- 1SB: standalone stressing benchmark in core #1.
- 2SB: 2 stressing benchmarks in (core #1, core #4).

- 3SB: 3 stressing benchmarks in (core #1, core #4, core #5).
- 4SB: 4 stressing benchmarks in (core #1, core #4, core #5, core #6).
- 5SB: 5 stressing benchmarks in (core #1, core #4, core #5, core #6, core #7).
- 6SB: 6 stressing benchmarks in (core #1, core #0, core #4, core #5, core #6, core #7).
- 7SB: 7 stressing benchmarks in (core #1, core #0, core #2, core #4, core #5, core #6, core #7).
- 8SB: 8 stressing benchmarks in (core #1, core #0, core #2, core #3, core #4, core #5, core #6, core #7).

Figure C.5.4 shows the distribution of the observed runtime of core #1 while varying the number of co-runners using the above mapping.

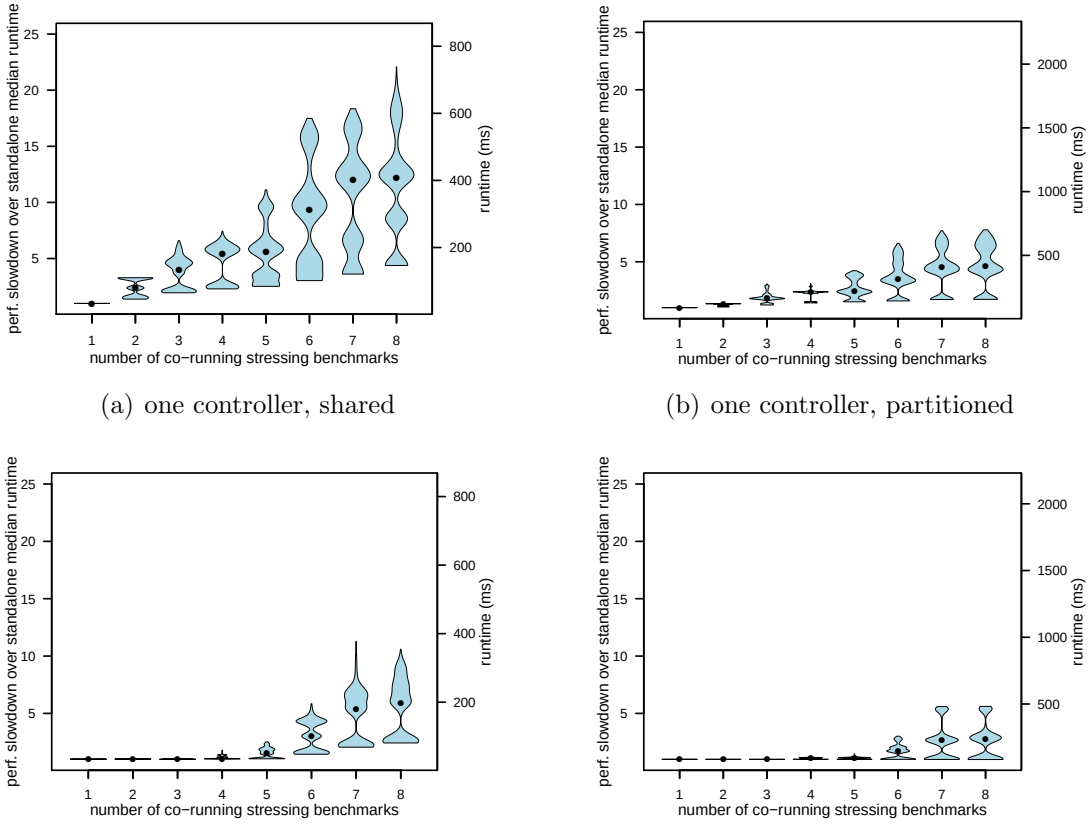
The y-axis on the left shows the observed speed down compared to when running the stressing benchmark standalone for each particular configuration. The y-axis on the right corresponds to the overall runtime in millisecond (ms).

The performance variability of the different configurations can be obtained by comparing the height of the various distribution violin plots relatively to the left y-axis, the tallest plot being the one with the largest variability. The worst performance of the various configurations can be observed with the top-most point of each violin plot. The associated runtime appears on the right y-axis.

The configurations exhibiting the lowest variability are the configurations with partitioned L3 cache(s) appearing in Figures C.5.4(b) and (d). On the other hand, enabling a second L3 cache and associated DDR controller in configurations in Figures C.5.4(c) and (d) also allow the system to reduce the performance variability by providing some load balancing between two clusters, while increasing the overall performances.

To put it simply, on one hand, activating the second L3 cache and associated controller brings both more predictability and more performance. On the other hand, shared L3 caches are providing more performance while partitioned L3 caches are offering more predictability.

To compare these two last configurations, we collected worst execution times of Figures C.5.4(c) and (d) into Table C.5.1. Even by offering a larger variability, the shared configuration provide better overall performance leading to lower worst case runtimes than the partitioned configuration. Therefore the worst case upper bound has the opportunity to be lower for the shared configuration, making the dual-controller shared configuration the most efficient setup for our safety critical system.



(a) one controller, shared

(b) one controller, partitioned

(c) dual controllers, shared

(d) dual controllers, partitioned

Figure C.5.4: *Runtime variability of one of the stressing benchmarks while varying the number of co-running instances.*

	1	2	3	4	5	6	7	8
dual-controller shared	34	34	35	61	84	196	378	355
dual-controller partitioned	86	86	87	102	126	259	481	482

Table C.5.1: *Worst execution times (in ms) for the monitored core while varing the number of running benchmarks.*

Even though we shown that the configuration with two controllers and shared L3 caches could be the most efficient for safety critical systems as allowing a lower upper bound for the worst execution time, avionic applications usually favor partitioning as it allows to minimize interferences, here eliminating L3 cache line evictions due to other co-running tasks. For this reason we focused on the con-

figuration with two controllers but with partitioned L3 caches for the remaining of the thesis.

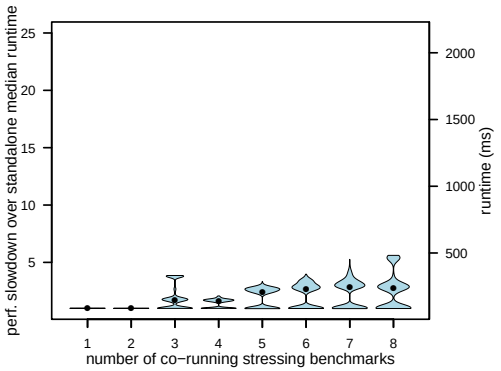
Using this configuration is also in cope with the cluster organization identified in Section C.5.2, and will benefit from the same design space reduction options as applications sharing the same cluster / L3 cache / memory controller will compete on the same shared hardware resources, whereas application placed in different clusters will not.

C.5.4 Selecting the Adequate Mapping

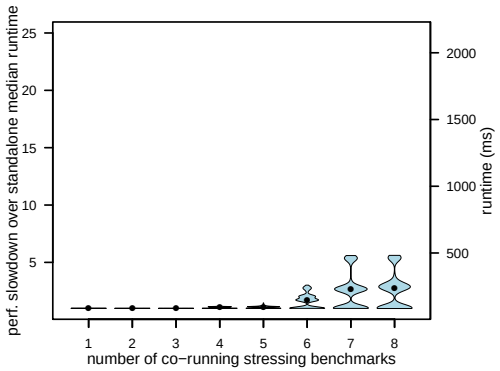
Selecting an optimal mapping with the considered dual-controller partitioned hardware setup is important as the amount of resource competition between tasks will be tied to their core placement, with large competition for tasks running on the same cluster, and with low or even no competition for tasks running on different clusters.

This optimal mapping largely depends on the application to be considered. In a system only running critical blackbox tasks with the same level of criticality, a fair load balancing of the tasks between the cluster should be privileged.

In a mixed critical system running a few very high-critical applications together with some lower-critical ones, designers may want to keep one cluster for the high-critical tasks, and the other cluster for the low critical tasks. This way, the possible impact of low-critical tasks on high-critical ones is minimized by reducing the sources of resouce competition between them.



(a) fair load balancing between tasks with the same criticality level



(b) load balancing minimizing tasks running with the high-critical one

Figure C.5.5: *Comparing the runtime variability of different balancing techniques.*

Figure C.5.5 presents these both setups. We placed the benchmark with highest criticality level in the first cluster, and then augmented the number of

co-running benchmarks. In Figure C.5.5(a) new benchmarks were placed to ensure fair load balancing as Mapping2 in Figure C.5.3, in Figure C.5.5(b) new benchmarks, considered of lower criticality were placed first on the other cluster as Mapping1 in Figure C.5.3.

While Figure C.5.5(a) corresponding to the fair load balancing exhibits similar and more regular variabilities, for Figure C.5.5(b) the performance variability starts to be impaired only when the number of co-runners do not fit anymore into the second cluster (from 6 co-running benchmarks). As a consequence, a mixed-critical system should better not try to use all the available resources to ensure that the low-critical traffic does not impact the high-critical one. In a word, a mixed-critical system should choose the mapping in Figure C.5.5(b) to keep the high-critical task non-disturbed.

C.5.5 Quantify the Shared Resource Availability

With a hardware setup selected, the last step of the hardware characterization is to identify the available amount of each hardware resource, quantifying the maximum throughput, available number of concurrent accesses and so on.

The most important resource to be characterized (and the least documented) is the CoreNet interconnect connecting the core clusters to their dedicated L3 cache and DDR controller, as well as to the other off-chip resources.

Evaluation of the CoreNet Interconnect.
Very few details on the topology of the CoreNet interconnect are available, and the only information available in the P4080 reference manual is the 0.8 Tbps coherent read bandwidth.

To better characterize this interconnect, we need to figure out its maximum throughput corresponding to the amount of traffic needed to saturate the interconnect. Remaining strictly below this saturation value would allow us to minimize the variability due to the interconnect, while getting closer to the saturation value would mean a significant increase of the runtime.

We also need to figure out how this available bandwidth is distributed among the clusters. Is the total bandwidth shared by all the cores, or is this bandwidth partitioned per-cluster?

We defined the CoreNet load as the number of CoreNet transactions per CPU CYCLE by monitoring the event *BIU master requests* and *CPU CYCLES*. To evaluate the correlation between the runtime and the CoreNet load, we set up a dedicated stressing benchmark only stressing the CoreNet and performing misses

in the L1 and L2 caches to maximize possible CoreNet load by setting the parameter **TABLESIZE** to fit the data in the partitioned area of L3 cache and the parameter **STRIDE** to use each L1/L2 cache line only once, and tuned the CoreNet load by regulating the parameter **NOP**.

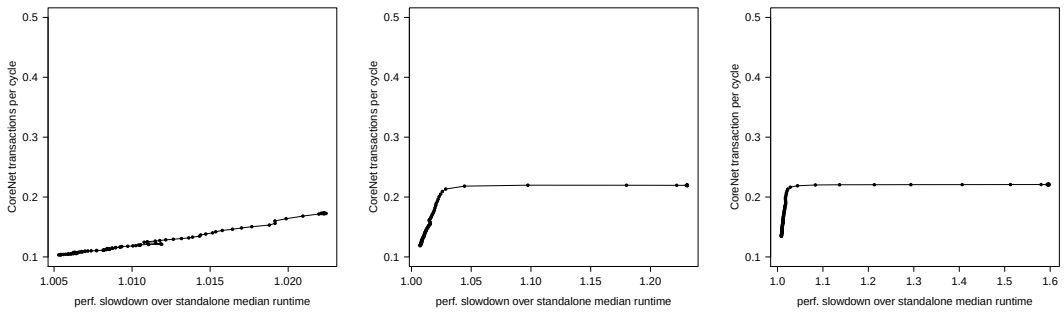
To learn the **maximun throughput** and the **saturation behavior** of the CoreNet, we organized the experiment as below:

- We ran a stressing benchmark in core #1, and measured its runtime by monitoring the event *CPU CYCLES*.
- We ran the same stressing benchmark(s) in the remaining cores in the same cluster with core #1. For every stressing benchmark co-running with core #1, we regulated the value of **NOP** to progressively interfere the execution of core #1.
- We monitored the event *BIU master requests* in all the co-running cores to get the CoreNet transactions. To synchronize the independent co-running cores to collect the event *BIU master requests*, core #1 signaled other cores with the timing of event collection using the interprocessor interrupt as we presented in Section C.4.2.
- We computed the CoreNet load by dividing the sum of *BIU master requests* in all the co-running cores by *CPU CYCLES* of core #1 (*CPU CYCLES* of all the co-running cores are actually the same, because they have been synchronized): sum(BIU)/CPU CYCLES.

The results of the experiment are presented in Figure C.5.6.

Figure C.5.6(a), (b) and (c) respectively depicts the correlation between the performance variability and the CoreNet load while running two, three and four instances of our stressing benchmark on the first cluster. In Figure C.5.6(a), the performance stays nearly unchanged (upto 2.5% of performance slowdown), which means that two co-running stressing benchmarks are not able to saturate the CoreNet. We can infer from the (b) and (c) that the CoreNet begins to be saturated from three co-running stressing benchmarks. The performance remains nearly unchanged below 0.219 CoreNet transactions per CPU cycle, but suffers a brutal degradation when reaching this value. This inflection point corresponds to when the CoreNet becomes saturated just before reaching the maximum bandwidth 0.221 CoreNet transactions per CPU cycle.

After identifying the maximun bandwidth and the saturation behavior of CoreNet, we organized a group of experiments to identify the CoreNet topology between two clusters under the optimal hardware configuration. The experiment was based on the previous experiment, but with different core mappings to discover how the CoreNet bandwidth is distributed between two clusters.



(a) (b) (c)

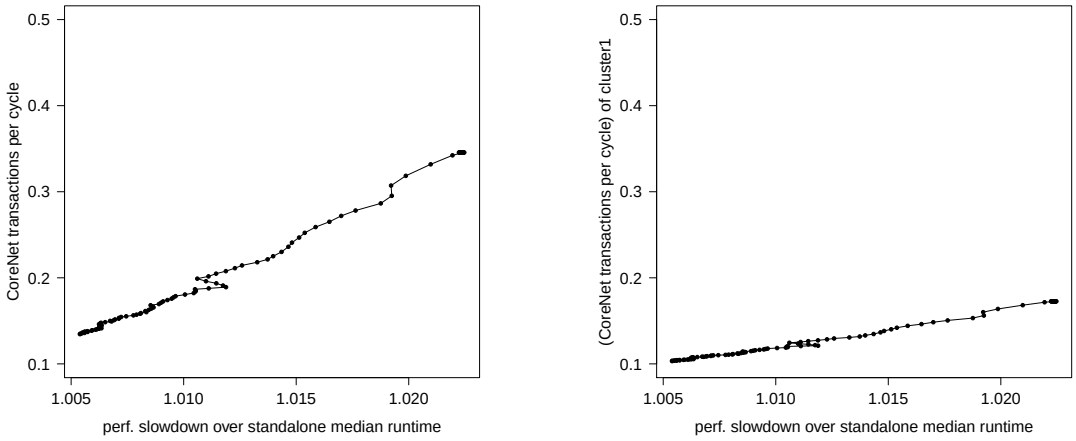
Figure C.5.6: *Performance slowdown versus CoreNet load to identify CoreNet maximum bandwidth and saturation behavior. (a) Total CoreNet load while running 2 co-runners in the 1st cluster, (b) Total CoreNet load while running 3 co-runners in the 1st cluster, (c) Total CoreNet load while running 4 co-runners in the 1st cluster*

- We ran four co-running benchmarks with half of them mapped in Cluster1 and the remaining half of them mapped in Cluster2.
- We ran six co-running benchmarks with half of them mapped in Cluster1 and the remaining half of them mapped in Cluster2.
- We ran eight co-running benchmarks with half of them mapped in Cluster1 and the remaining half of them mapped in Cluster2.

Figure C.5.7, Figure C.5.8 and Figure C.5.9 respectively show how the performance varies according to the CoreNet load in above three different mappings.

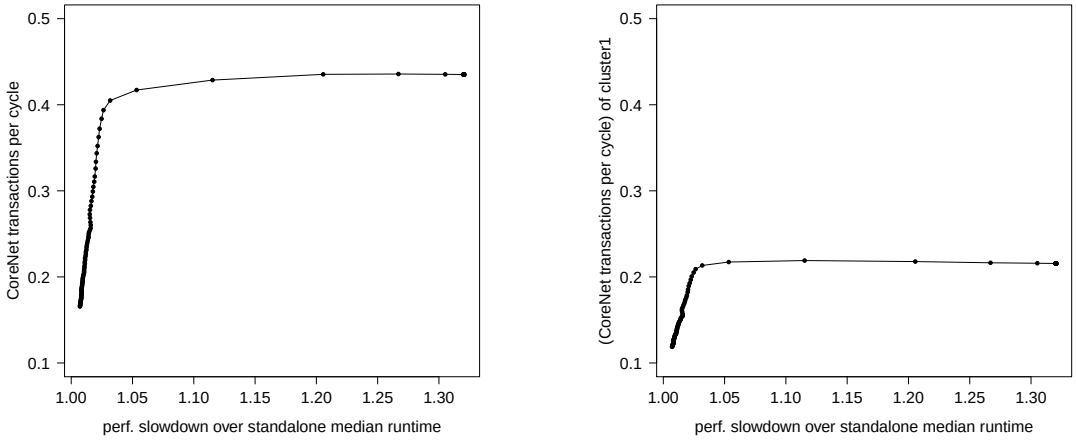
Figure C.5.7(a) describes the correlation between the performance of core #1 and total CoreNet load while running four instances of the stressing benchmark in two clusters. The maximum value of CoreNet load is 0.346 transaction per CPU cycle which is doubled compared with Figure C.5.6(a). Figure C.5.7(b) describes the correlation between the performance of core #1 and CoreNet load of Cluster1, which demonstrates the same behavior with Figure C.5.6(a). The doubled maximum value and the same behavior of Cluster1 means that 1) using the second cluster, namely enabling the second L3 cache, triggers the other part of the CoreNet bandwidth. 2) when two clusters are both below the CoreNet saturation 0.219 transaction per CPU cycle, the CoreNet bandwidth is evenly distributed between to both clusters without any inter-cluster interference.

Figure C.5.8 and Figure C.5.9 describes the same correlation when running six and eight instances of the stressing benchmark. Figure C.5.8(a) and Fig-



(a) (b)

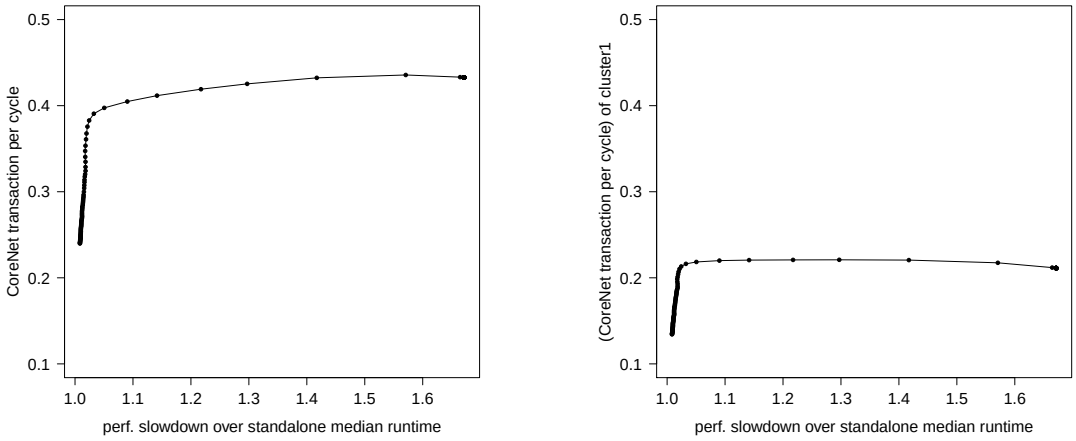
Figure C.5.7: *Performance slowdown versus CoreNet load while 4 co-runners balanced in two clusters to identify CoreNet topology. (a) Performance slowdown versus total CoreNet load, (b) Performance slowdown versus CoreNet load of Cluster1.*



(a) (b)

Figure C.5.8: *Performance slowdown versus CoreNet load while 6 co-runners balanced in two clusters to identify CoreNet topology. (a) Performance slowdown versus total CoreNet load, (b) Performance slowdown versus CoreNet load of Cluster1.*

ure C.5.9(a) clearly show that the performance degradation occurs once the num-



(a) (b)

Figure C.5.9: *Performance slowdown versus CoreNet load while 8 co-runners balanced in two clusters to identify CoreNet topology. (a) Performance slowdown versus total CoreNet load, (b) Performance slowdown versus CoreNet load of Cluster1.*

ber of CoreNet transactions reaches 0.406 transaction per CPU cycle which is nearly doubled compared to Figure C.5.6(b) and (c). Figure C.5.8(b) and Figure C.5.9(b) depict that the per-cluster CoreNet saturation is 0.219 transaction per CPU cycle which is the same with Figure C.5.6(b) and (c). The nearly doubled value and the same per-cluster saturation means that when two clusters both reaches the per-cluster CoreNet saturation 0.219 transaction per CPU cycle, the total saturated CoreNet (0.406 transactions per cycle) is not sufficient to support the saturated bandwidth of both clusters (2×0.219 transactions per cycle). The same conclusion can be made for the total maximum bandwidth of the CoreNet: it (0.436 transactions per cycle) is not sufficient to support the maximum bandwidth of both clusters (2×0.221 transactions per cycle). In other words, each cluster has its private basic CoreNet bandwidth, but there is still a little concurrent part of CoreNet competed between two clusters. The competition on this concurrent part can be reflected by the tail of the curve in Figure C.5.8(b) and Figure C.5.9(b) which falls a little compared to Figure C.5.6(b) and (c). That's because two clusters both need the maximum per-cluster bandwidth. Since they are not able to both get the maximum at the same time, the Cluster1 has to cede some concurrent bandwidth to the Cluster2. In addition, the competition on the concurrent part can be also observed through the extra performance slowdown upto 9% and 8% in Figure C.5.8(b) (the maximum slowdown is $\times 1.23$) and Figure C.5.9(b) (the maximum slowdown is $\times 1.59$) compared to Figure C.5.6(b)

(the maximum slowdown is ×1.32) and (c) (the maximum slowdown is ×1.67).

In a real-time context, to enforce that the activity of a cluster does not impact the activity of the other one, we therefore need to make sure that this saturation bandwidth is not reached.

Evaluation of the DDR.
Another important shared hardware resource to consider are the DDR controllers and the associated DDR memory.

To similarly evaluate the maximum throughput and saturation value of each DDR controller, we shifted back to the hardware setup with one unique L3 cache and associated DDR controller. This allowed us define a stressing benchmark aiming at stressing the DDR controller with the load of 8 different running cores performing misses in the L3 cache. To identify the maximum throughput and saturation behavior of the DDR controller, we organized the similar experiment with CoreNet evaluation:

- We ran a stressing benchmark in core #1, and measured its runtime by monitoring the event *CPU CYCLES*.
- We ran the same stressing benchmark(s) in the remaining cores with core #1. For every stressing benchmark co-running with core #1, we regulated the value of **NOP** to progressively interfere the execution of core #1.
- We monitored the events *DDRread* and *DDRwrite* in core #1 for all the co-running cores to get the total number of DDR accesses.
- We computed the DDR load by dividing the sum of *DDRread* and *DDRwrite* by *CPU CYCLES* of core #1: (DDRread+DDRwrite)/CPU CYCLES.

Figure C.5.10 shows the correlation between the performance variability of the benchmark monitored in core #1 and the number of accesses to the DDR controller. The figure exhibits again an inflection point when saturating the DDR controller when reaching 0.042 accesses to the controller per CPU cycle.

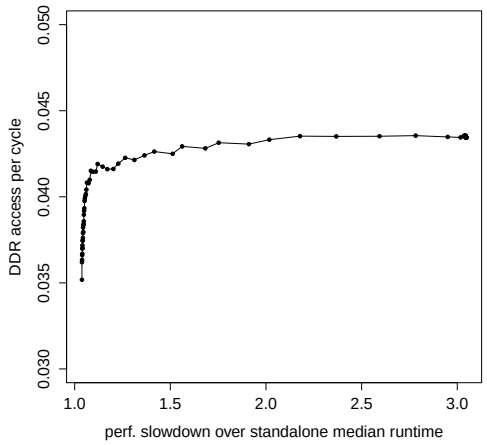


Figure C.5.10: *Runtime variability versus DDR controller accesses to identify each DDR controller maximum bandwidth*

Chapter C.6

Application Characterization Results

In the previous chapter we quantified the different available shared hardware resources. In this section, we focus on identifying the resource requirements of the applications. The application runtime variability while stressing a particular resource will allow to determine the share of the hardware resource that each application requires. We characterized applications under the optimal hardware configuration of P4080.

We start by figuring out the optimal number of iterations required to fully capture the runtime variability of each application, and then quantify the sensitivity of each application to the shared resources and the resource usage of each application. Such an information about resource usage could later be used to determine which applications could run smoothly together.

C.6.1 Optimal Number of Iterations to Capture Variability

To capture runtime variability of a particular application, each experiment involving this application has to be run a large number of times in successive iterations. A large enough number of iterations will be able to capture the whole runtime variability of the application, while a not sufficiently large number will miss the runtime with the rarest distribution. As missing the worst execution time is not an option, we need to figure out what is this optimal number of iterations allowing to fully capture the runtime variability.

To empirically determine this optimal execution iteration number of a particular application, we setup an experiment performing successive executions of this

application concurrently with a resource-stressing environment. Every 100 iterations, we collected the runtime distribution since the beginning of the execution.

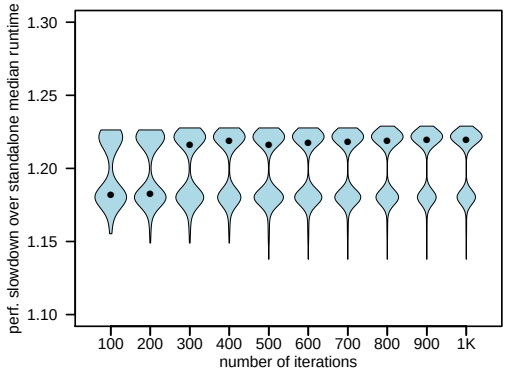


Figure C.6.1: *Runtime variability collected with different number of iterations for application Adpcm.*

Figure C.6.1 shows such runtime distribution results for the Adpcm benchmark co-running with two DDR stressing benchmarks. The shape of a violin plot corresponds to the captured behavior of the application. Therefore, comparing the violin plot shapes enables us to figure out if the optimal iteration value has been reached. For these experiments, stopped the iteration counter when we obtained three consecutive identical violin plots. For Figure C.6.1 the optimal number of iteration is therefore 1000.

Identifying the optimal number of iterations to capture the runtime variability of each application allows us to reduce the overall design space. We applied the same methodology to identify the optimal number of iterations for every experiment.

C.6.2 Identifying the Sensitivity to Shared Resources

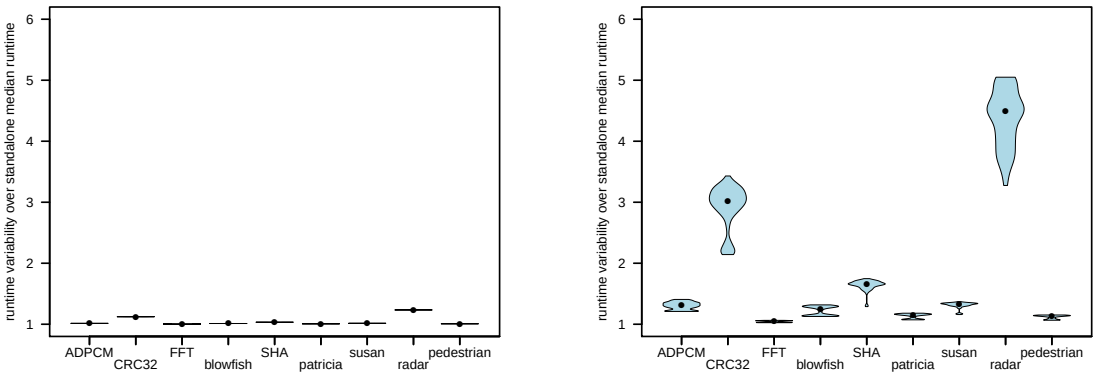
We quantified the performance variability that each application may experience due to the collision on shared resources in Chapter B.4. However, in addition to representing the runtime variability, the performance slowdown of an application can be also used to perform as the sensitivity of the application to the resource(s) stressed by co-running stressing benchmarks. If the performance slowdown is low, the application is not sensitive to the stressed resource(s), which means that the application does not need a significant quota of the resource(s) to complete its execution. On the contrary, if the application experiences a significant slowdown

when it runs with a resource stressing benchmark, the application certainly needs a great usage of the resource showing a high sensitivity to it.

In this section, we used a CoreNet and a DDR stressing benchmark with the worst case cache locality to identify the sensitivity of the applications to the CoreNet and DDR under the optimal configuration. Since we identified two symmetric clusters under the optimal configuration, we restricted our experiment in one cluster to identify the sensitivity. The experiment was organized in the same way with the variability quantification.

- Measure the runtime of a target application when it runs in isolation in core #1 $\mathrm{RT}_{isolation}$.
- Measure the runtime of the target application in core #1 when it runs simultaneously with three CoreNet/DDR stressing benchmarks in the remaining cores in the same cluster $\mathrm{RT}_{corunning}$.
- Normalise $\mathrm{RT}_{corunning}$ by $\mathrm{RT}_{isolation}$ to get the performance slowdown, namely the sensitivity to CoreNet/DDR.
- Repeat the above measurement within the optimal number of iterations to capture the variability and plot a violin plot.

Figure C.6.2 shows the sensitivity of nine applications to the CoreNet and DDR.



(a) (b)

Figure C.6.2: *The sensitivity of applications to (a) the CoreNet, (b) the DDR.*

We can infer from Figure C.6.2 that CRC32, airborne radar are more sensitive to both CoreNet and DDR compared to other applications, and FFT is the most stable application showing the lowest sensitivity. In addition, comparing Figure

C.6.2(a) and (b) allows us to discover that the contention on the DDR is much more destructive than on the CoreNet to the performance.

However, the sensitivity only tells us upto which extent an application is sensitive to a shared resource, but it can not provide the shared resource usage of the application which is required to deduct possible co-running applications without exceeding the resource saturation. We thus capture the shared resource usage of applications in the next section.

C.6.3 Capturing the Shared Resource Usage

Section C.5.5 allowed us to quantify the maximum throughput of shared resources: CoreNet and DDR controller in the architecture. We now want to capture the resource requirements of each standalone application.

To perform this measurement, we ran each application standalone, collecting the event *CPU CYCLES*, *BIU master requests*, *DDRread* and *DDRwrite* to get the CoreNet load and the DDR load of the application by computing *BIU master requests*/*CPU CYCLES*, and (*DDRread* + *DDRwrite*)/*CPU CYCLES*.

Table C.6.1 provides the resource usage information for each application, this usage being computed as a ratio to the previously quantified saturation value: 0.219 requests per CPU cycle for the CoreNet, and 0.042 requests per CPU cycle for the DDR controller.

Application	average CoreNet	peak CoreNet	average DDR	peak DDR
ADPCM	0.86%	9.04%	4.52%	47.13%
CRC32	0.93%	1.30%	4.77%	6.90%
FFT	0.19%	13.38%	0.38%	31.43%
blowfish	0.14%	0.72%	0.74%	3.74%
SHA	0.25%	2.15%	1.33%	11.19%
patricia	0.07%	0.19%	0.35%	0.97%
susan	0.40%	2.96%	2.01%	15.44%
airborne radar	2.23%	3.06%	11.68%	16.19%
pedestrian detection	0.10%	4.29%	0.48%	22.86%

Table C.6.1: *CoreNet and DDR loads of standalone application*

Table C.6.1 lists both the average and peak number of resource usage for the considered applications. We collected the peak value of CoreNet and DDR related hardware counters relying on sampling techniques that we will present in Part D. The maximum average usage remains quite low: 2.23% of the available CoreNet resources, and 11.68% of the DDR resources. The peak usage however is significant with Adpcm using as much as 47% of the available DDR bandwidth.

In addition, we confirm that CRC32 and airborne radar which are more sensitive to shared CoreNet and DDR also have a higher CoreNet and DDR average usage compared to others.

C.6.4 Determining Possible Co-running Applications using Resource Usages

With the resource usage of each benchmark and the total amount of available resources quantified, we can deduct which applications could run together without significantly endangering each other's performance. Considering that the performance remains nearly unchanged below the CoreNet saturation and the DDR saturation, we can deduct that the performance of co-running applications will not significantly degrade if the sum of their peak CoreNet usages and the sum of peak DDR usages both do not exceed the corresponding saturations.

Looking at the peak usage of the DDR resource for the ADPCM application in Table C.6.1, up to two instances of ADPCM should run fine on the same cluster with low performance impact compared to the standalone version; 3 and 4 instances should start to exhibit significant slowdown because their total peak DDR usage exceeds the saturation, namely more than 100%.

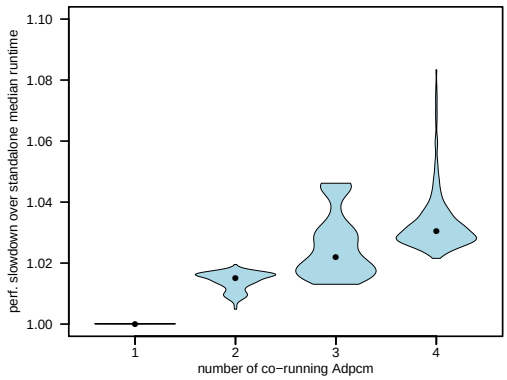


Figure C.6.3: *Performance slowdown with difference number of co-running ADPCM.*

Figure C.6.3 depicts the performance variability while running an increasing number of ADPCM instances on a single cluster. All slowdown are normalized to the standalone execution time of ADPCM. Running two concurrent instances of the ADPCM benchmark produces a slight maximum performance degradation of +2%. This is in cope with the fact that the DDR controller is just below saturation. When running three concurrent instances the maximum increases to +5%, and to +8% when co-running four different instances.

Even though the impact on runtime behavior is not that high, the behavior is correctly captured. The reason why the maximum performance degradation is only 8% is due to the fact that the average DDR controller usage of ADPCM is only 9%, far away from the peak usage of 47%.

Chapter C.7

Conclusion

In this part, we presented a methodology and its associated automatic framework allowing us to characterize both the hardware and the safety-critical software relying on hardware monitors available in multi-core architectures and stressing benchmarks. From the hardware point of view, we successfully

- identified some undisclosed hardware features: 4-core cluster effect in the P4080, which helped reduce the design space in terms of the mapping. The mapping which tries to avoid to map the co-running applications in the same cluster with the application under monitor will provide the lowest variability for this application under monitor.
- identified the most suitable hardware configuration for safety-critical applications. According to two criteria: the low variability and sufficient overall performance, the optimal configuration is dual controller/partitioned L3 cache.
- quantified shared hardware resource availability: the maximum throughput, the saturation behavior and the topology of the CoreNet and the DDR controller, which helped compute the resource usage in the application characterization. Either the CoreNet or DDR controller has a saturation bandwidth below which the performance remains nearly unchanged. Once it is reached the performance will dramatically degrade. Under the optimal configuration, each cluster has a basic proper CoreNet bandwidth but they still share a little part of bandwidth. To enforce that the activity of a cluster does not impact the activity of the other one, we need to make sure that the per-cluster saturation bandwidth 0.219 transactions/cycle is not reached in both clusters.

From the software point of view, we were able to

- get the optimal iteration for capturing the variability, which helped reduce the design space.
- capture the sensitivity of applications to shared resources.
- capture the average and the peak resource usage.
- perform the first prediction on co-running application behavior using resource usage information.

For safety-critical applications, we need an estimated execution time upper bound of co-running applications, and determine if the co-running applications can practically run together by comparing this upper bound with the required deadline. However, our first prediction can not accurately estimate an upper bound of the co-running execution time, which prevents us to guarantee a safe execution without endangering the deadline.

In order to estimate an execution time upper bound, we proposed an alternative technique to estimate the WCET in the next part, Part D.

Part D

Alternative technique to Estimate the WCET

Chapter D.1

WCET Estimation Methodology

As we explained in Chapter C.7, to be able to guarantee a safe co-running execution without any risk of missing deadlines for safety-critical systems, we need a more reliable technique to get a precise execution time upper bound in the co-running context.

We therefore propose a methodology heavily based on worst-case execution/simulation techniques to easily and rapidly estimate the WCET of an application when co-running with a pre-determined set of applications in safety critical systems. The procedures of the methodology are illustrated in Figure D.1.1.

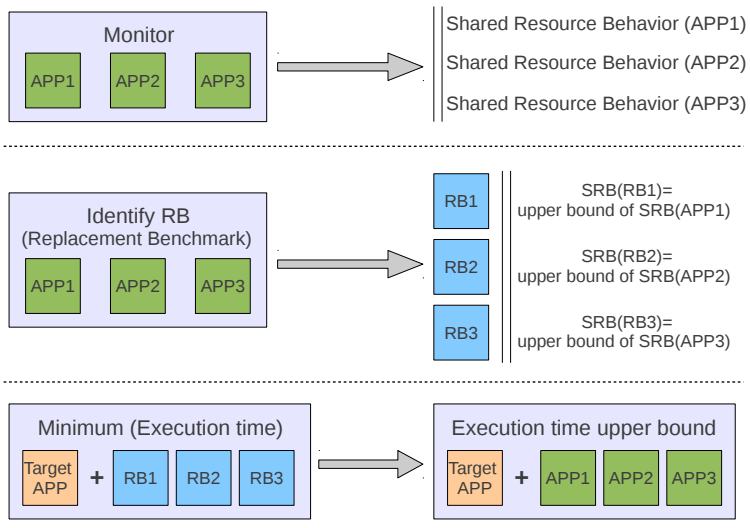


Figure D.1.1: *The upper bound estimation methodology.*

The methodology first monitors the shared resources related behavior of a set of standalone pre-determined applications. Then, it identifies for each of these

applications a replacement benchmark that exhibits an upper bounded behavior over the original application relatively to the monitored shared resources.

Such replacement benchmarks should allow us to simulate the worst-case contention on shared hardware resources caused by the original applications. By co-running these replacement benchmarks together with a target application, we will be able to estimate a safe execution time upper-bound for the target application while co-running with the original applications.

The proposed method can be applied in a black-box context without knowing the full details of the hardware and the software. The avionic industry is facing such a context when integrating third-party software componnents into COTS hardware with undisclosed features.

Chapter D.2

Experimental Setup

D.2.1 Experimental Scenario

During the previous architecture characterization (see Section C.3.1.3 and Section C.5.3), we developed a methodology automatically determining the most suitable configuration of P4080 such that the runtime variability is reduced without significantly degrading the average performance. For that purpose, we performed a per-core partitioning of the L3 caches and the DDR memory space. Potential interferences are therefore restricted to concurrent accesses to the CoreNet, L3 caches and DDR controllers. In addition, we also **discoverd** that the most suitable configuration has a symmetric cluster behavior with two groups of four cores, each cluster sharing a basic part of CoreNet to both L3 caches. So for the sake of simplicity, we restricted our estimation evaluation to identify the potential co-runners of a single cluster under the identified optimal configuration on the bareboard P4080.

D.2.2 Measurement Techniques

We used hardware monitors and stressing benchmarks to evaluate the estimation methodology.

D.2.2.1 Hardware Monitors

The first step of the methodology is to monitor the shared resources related behavior of pre-determined applications that we want to run with a target application. As we did during the architecture and application characterization, we used hardware monitors to collect shared resource related events. We evaluated the estimation methodology under the optimal configuration identified in Part C

on the P4080, so the shared resources are the CoreNet and DDR controllers. We thus only monitored the events listed in Table C.2.1.

To ease comparison of the shared resource behavior among different applications, we normalised CoreNet and DDR events by *CPU CYCLES*:

- BIU/cycle: the number of CoreNet transactions per cpu cycle
- DDRr/cycle: the number of DDR read accesses per cpu cycle
- DDRw/cycle: the number of DDR write accesses per cpu cycle

D.2.2.2 Stressing Benchmarks

The second step of the estimation methodology is to identify a replacement benhcmark for a pre-determined application by taking the upper bound behavior on shared resources. To perform the replacement, we designed stressing benchmarks.

Stressing benchmarks have been introduced for the variability quantification and the characterization as a way to produce a high load on a particular hardware resource. The purpose of these stressing benchmarks was twofold: 1) to actually identify the shared hardware resources and associated contention mechanisms, and 2) to identify the hardware resources each application is sensitive to.

In this part, we designed these stressing benchmarks in the same way described in Section C.2.2, but we extended these stressing benchmarks with the ability to simultaneously stress multiple hardware resources and finely tune the load they are producing to each of them. The stressing benchmark used to replace an application should behave like the original application relatively to the monitored shared hardware resources, while exhibiting upper bounded behavior.

For example, we are able to create a stressing benchmark with 0.01 BIU/cycle, 0.005 DDRr/cycle and 0.001 DDRw/cycle. We will show how the methodology uses these stressing benchmarks in Chapter D.3 and Chapter D.4.

Chapter D.3

Global Signature

D.3.1 Defining Global Signatures

The global signature of an application provides some global information about how the overall application behaves relatively to each hardware resource. The global signature is defined as a vector of normalized hardware counters values collected during an application full standalone run on one core, with the other cores being idle. As previously said, we restrict ourselves to three shared memory path related metrics for the signature elements: BIU/cycle, DDRr/cycle and DDRw/cycle The global signature can be therefore represented as: [BIU/cycle, DDRr/cycle, DDRw/cycle] which gives information about overall CoreNet and DDR behavior.

To capture potential global signature variability, we collected these signatures for hundreds of different iterations, and only kept the maximum values representing the highest observed impact on hardware resources. These collected values, provided in Table D.3.1 exhibited a very small (below 0.1%) variation for these standalone runs.

Application	BIU/cycle	DDRr/10^2cycle	DDRw/10^2cycle
ADPCM	0.001871	0.103959	0.083134
CRC32	0.002029	0.202885	0.000007
FFT	0.000409	0.012522	0.003619
blowfish	0.000308	0.030787	0.000007
SHA	0.000562	0.056187	0.000002
patricia	0.000147	0.014689	0.000004
susan	0.000892	0.058581	0.025376
airborne radar	0.004879	0.484096	0.001111
pedestrian detection	0.000219	0.018304	0.001795

Table D.3.1: *Global signature of target applications*

Hardware monitor collection is performed between each iteration of a full application run, ensuring a non-intrusive monitoring of the application.

D.3.2 Using Global Signatures

Global signatures enable us to select for each original application a stressing benchmark with the closest upper bound signature (we used the euclidean distance). This identified stressing benchmark is likely to behave like the original application relatively to the monitored shared hardware resources, while exhibiting much less variability.

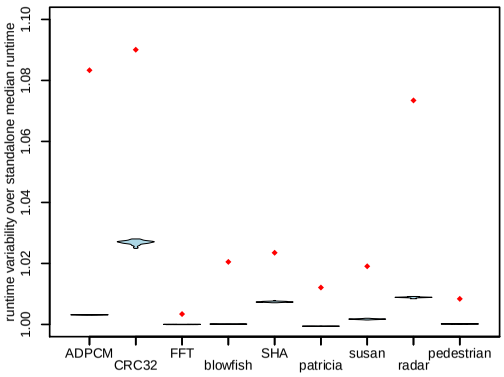
Our assumption is that the stressing benchmarks should exhibit a very close and upper bounded behavior over the original benchmark relatively to these shared hardware resources, while requiring much less iterations to capture the variability on resource usage and runtime.

To validate this assumption, we organized the experiment as:

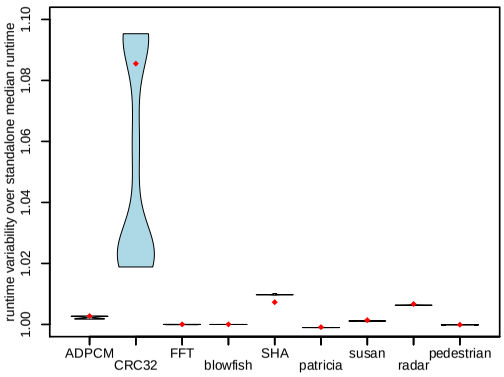
- We ran a target application on core #1 in isolation to get its standalone runtime $\mathrm{RT}_{isolation}$.
- We ran this target application on core #1 with three instances of an original application in the same cluster within the optimal number of iterations to get a set of co-running runtimes $\mathrm{RT}_{co-apps}$.
- We ran this target application on core #1 with three instances of the identified closest stressing benchmark of the original application in the same cluster within the optimal number of iterations to get a set of co-running runtimes RT_{co-sbs}.
- We normalised $\mathrm{RT}_{co-apps}$ and RT_{co-sbs} respectively by $\mathrm{RT}_{isolation}$ to get a set of performance slowdown for these two groups of experiment: $\mathrm{PS}_{co-apps}$ and PS_{co-sbs}.

The observed slowdown is related to the interference among co-running applications on the shared hardware resources, and we expect all the PS_{co-sbs} caused by co-running stressing benchmarks to upper bound the worst case performance slowdown $\max(\mathrm{PS}_{co-apps})$ caused by the co-running original applications, meaning that the stressing benchmark could be used to predict the worst case contention on shared resources caused by the original application in a safety critical context.

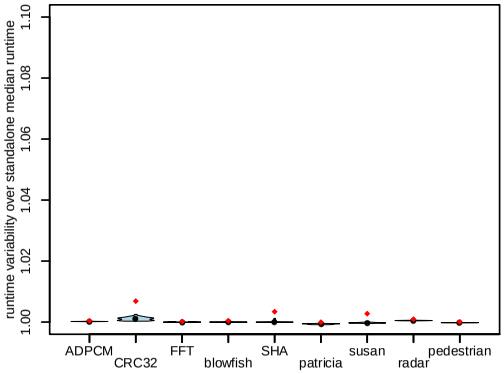
Figure D.3.1 depicts the experimental results: the maximum observed performance slowdown while running with three original applications (ADPCM in (a), CRC32 in (b), FFT in (c), SHA in (d), patricia in (e), susan in (f), airborne radar in (g) and pedestrian detection in (h)): $\max(\mathrm{PS}_{co-apps})$ is presented by a single red mark, while all the values of PS_{co-sbs} obtained by co-running stressing benchmarks are represented by a blue violin plot. If our upper bounding assumption is correct, every violin plot should completely appear above the corresponding red



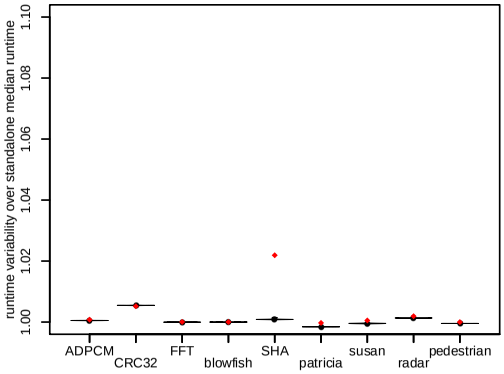
(a) ADPCM



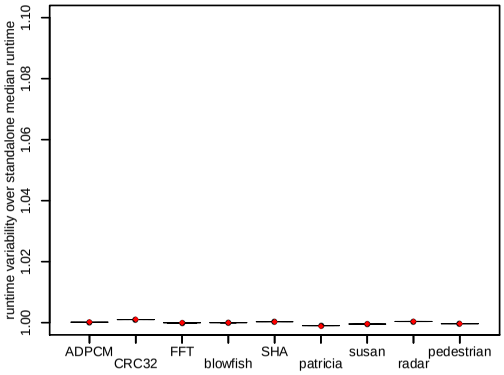
(b) CRC32



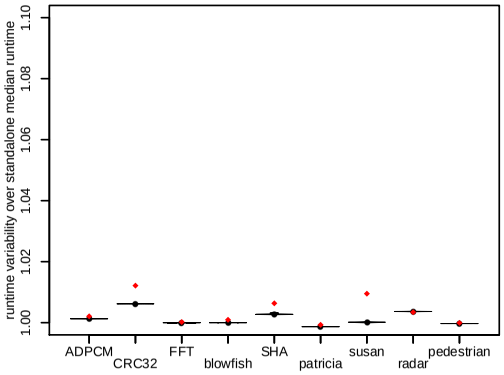
(c) FFT



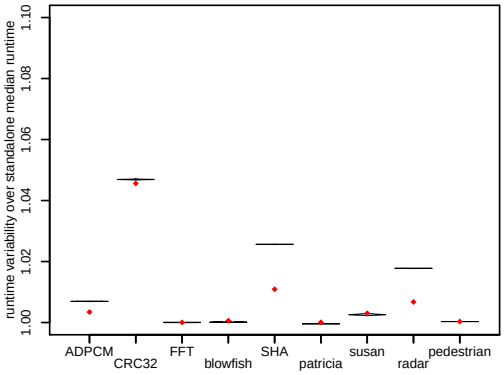
(d) SHA



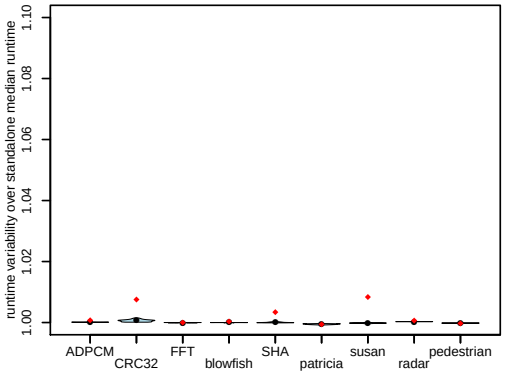
(e) Patricia



(f) Susan



(g) Airborne radar



(h) Pedestrian detection

Figure D.3.1: *Evaluating the global signatures against the performance slowdown induced by co-running with 3 instances of ADPCM, CRC32, FFT, SHA, patricia, susan, airborne radar, pedestrian detection versus their equivalent stressing benchmarks. Blue violin plots represent the runtime variability while co-running with stressing benchmarks. The red marks denote the maximum runtime while co-running with the original applications.*

mark, whereas violin plots closeby the corresponding mark will demonstrate the accuracy of the prediction.

Running these experiments, we observed two categories of applications: For CRC32, FFT, SHA, patricia, susan, airborne radar and pedestrian detection we observed a very accurate prediction, because all the violin plots is very close to the corresponding red mark. However we failed in systematically over bounding the worst case performance slowdown of the target application for some applications. For example, for airborne radar in Figure D.3.1(g), we only succeeded in over bounding the target applications: ADPCM, CRC32, SHA and airborne radar. The over-bounding ability defined as the relative difference between the number of successful cases and the total cases is therefore: 4/9=44.4% for airborne radar. The other category of applications is like ADPCM in Figure D.3.1(a). Most of the violin plots stay far from their corresponding red marks (i.e. 8% for ADPCM and 6% for CRC32). In addition, we failed in over bounding all the target applications, exhibiting over-bounding ability 0%. This second category reveals that the identified stressing benchmark is not able to simulate the shared resources behavior of the original application.

Table D.3.2 lists the over-bounding ability in each subfigure of Figure D.3.1 and its over-margin value which is defined as the average of the disparity between the minimum value of each violin plot and the red mark: average((abs(min(PS_{co-sbs})-max($PS_{co-apps}$))).

Even though providing an accurate prediction with an average over margin of only 0.72% as shown in Table D.3.2, the global signature fails to provide the required systematic upper bound, only succeeding in 24.9% of the cases. This technique therefore proves itself to be irrelevant for safety critical applications.

Application	over-margin value	over-bounding ability
ADPCM	3.69%	0.00%
CRC32	0.79%	11.1%
FFT	0.18%	0.00%
SHA	0.28%	11.1%
patricia	0.00%	100%
susan	0.25%	11.1%
airborne radar	0.36%	44.4%
pedestrian detection	0.24%	22.2%
average	0.72%	24.9%

Table D.3.2: *Evaluating global signature accuracy in terms of over-margin value and upper-bounding ability.*

D.3.3 Limitation of Global Signatures

By collecting hardware monitors for full application runs, we artificially introduced an averaging effect on this behavioral information. We take an example to illustrate the limitation of global signatures caused by this average behavior in Figure D.3.2. Application1 requires 80% of a particular hardware resource during the first 50% of the full run time, the average resource usage is therefore 40%. We identify a stressing benchmark with a regular resource usage 40% to simulate Application1. Synchronized two co-running Application1 will certainly cause a significant contention on the resource since their total requirement exceeds the resource saturation ($2 \times 80\%$). However, co-running two stressing benchmarks may not bring any performance slowdown since their resource demand is always below the resource saturation ($2 \times 80\%$). As a consequence, the stressing benchmark is not able to simulate the worst case contention caused by the peak resource usage 80% that is hidden by the averaging effect introduced in global signatures, which results in failure of over bounding the WCET.

In the following chapter, we present an alternative technique based on local signatures to capture the application's phase behavior of required resources, including the peak resource usage which can be used to simulate the worst case contention.

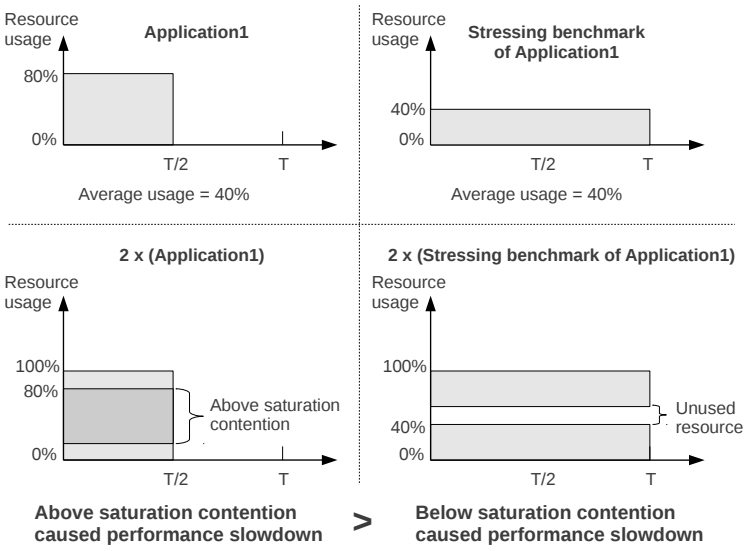


Figure D.3.2: *The example showing limitations of global signatures.*

Chapter D.4

Local Signature

D.4.1 Defining and Collecting Local Signatures

Whereas global signatures are collected once for each full run of the application, local signatures rely on sampling techniques to collect the monitored information on regular basis, many times during each application full run. As a consequence a local signature is not represented by a single vector of hardware monitor values, but by a succession of such vectors.

D.4.1.1 Collecting Local Signatures Using Fixed-Interval Timer (FIT)

The Fixed-Interval Timer (FIT) is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. To perform the regular collection, we implemented an interruption handler triggered by the FIT using the hardware clock. During this interruption, the target application running in isolation is suspended while the hardware counters are collected.

D.4.1.1.1 Implementing the FIT Interrupt

To perform a FIT interrupt, there are a set of registers to configure:

- Time Base (TB): it is composed of two 32-bit registers: the time base upper (TBU) concatenated on the right with the time base lower (TBL). The TB is interpreted as a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the least-significant bit.
- Time Control Register (TCR): it provides control information for the on-chip timer of the core e500mc. There are three fields related to the FIT:

  - TCR[FP] and TCR[FPEXT]: Fixed interval timer period and Fixed interval timer period extention. TCR[FR] concatenates with TCR[FPEXT] to specify one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1.
  - TCR[FIE]: Fixed interval interrupt enable. Set 1 to enable the FIT interrupt.

- Timer Status Register (TSR): it contains status on timer events. The field related to the FIT is TSR[FIS]: Fixed-interval timer interrupt status. TSR[FIS] must be reset by writing 1 to it in order to avoid a redundant fixed-interval timer interrupt.

- Core timebase enable register (RCPM_CTBENR): it provides a mechanism for enabling clocks to any core timebases on the device.

The procedures of the FIT interrupt implementation is:

1. Setting the FIT period by configuring TCR[FP] and TCR[FPEXT].
2. Enabling the FIT interrupt by setting the TCR[FIE].
3. Enabling clocks to current core TB by configuring RCPM_CTBENR.
4. Resetting TSR[FIS] in the FIT interrupt service routine once the FIT interrupt occurs.

D.4.1.1.2 Collecting Local Signatures

The mechanism of collecting local signatures using the FIT interrupt is shown in Figure D.4.1. We configured the FIT related registers to set a fixed time slot T in the beginning of a standalone application. A FIT interrupt occurs every T time slot jumping into the FIT interrupt service routine to execute the routine function where the TSR[FIS] is reset to avoid a redundant interrupt, the clock to current timebase is disabled to freeze the TB counting, hardware counters are collected, the TB is reset to the initial value for the next time slot counting and the clock to the timebase is enabled to unfreeze the TB counting. We finally get N successive vectors of [BIU/cycle, DDRr/cycle, DDRw/cycle].

This local signature collection is therefore more intrusive than for global signatures, but provides detailed information for every execution time slot. The granularity of the information depends on the time slot. A shorter time slot provides more detailed information while leading to more intrusion or overhead. We should therefore select an appropriate time slot to capture the peak resource

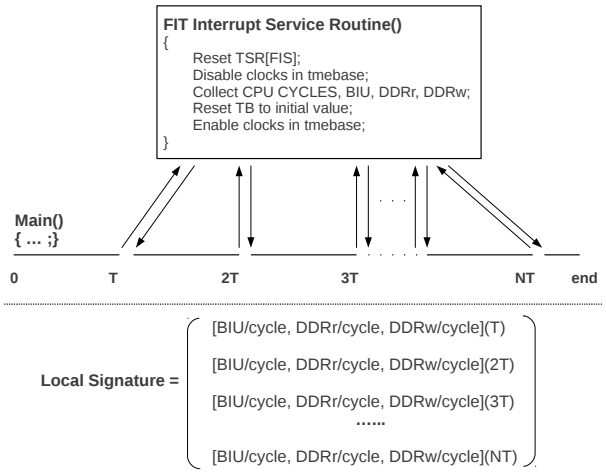


Figure D.4.1: *The mechanism of collecting local signatures using the FIT interrupt.*

usage which is responsible for the worst case performance slowdown but at the same time to guarantee an acceptable intrusion or overhead.

The mechanism of selecting the time slot is described as below:

1. Select a time slot T with magnitude 10^6 CPU CYCLES. If the full runtime is short, we can select a smaller T.

2. Collect local signatures using fixed interval T. After collection, we compute the mean value of each set of normalised counters: $\text{mean}_{BIU/cycle}$, $\text{mean}_{DDRr/cycle}$ and $\text{mean}_{DDRw/cycle}$. We then compare the mean values with their corresponding global values collected in the global signature. If the difference between the mean and the global value: abs(mean-global)/mean is less than 20%, we consider it as an acceptable overhead caused by the interrupt, which means that the time slot T is usable. Otherwise, T is not usable and we should take a larger time slot $2 \times T$ to repeat this step.

3. If T is usable, we reduce T to T/2 and repeat the second step. If T/2 is also usable, we compare the maximum value of each normalised counter collected in the time slot T with corresponding maximums in T/2. If the maximums are similar with a difference below 10%, we take T/2 as an appropriate time slot to capture the peak usage of resources. If the maximums in T/2 are larger than in T, we continue to reduce the time slot to its half value and repeat this step to test if the current time slot is an appropriate time slot.

We show in Figure D.4.2 the results of BIU/cycle of Adpcm as an example to illustrate the impact of time slot on the local signatures.

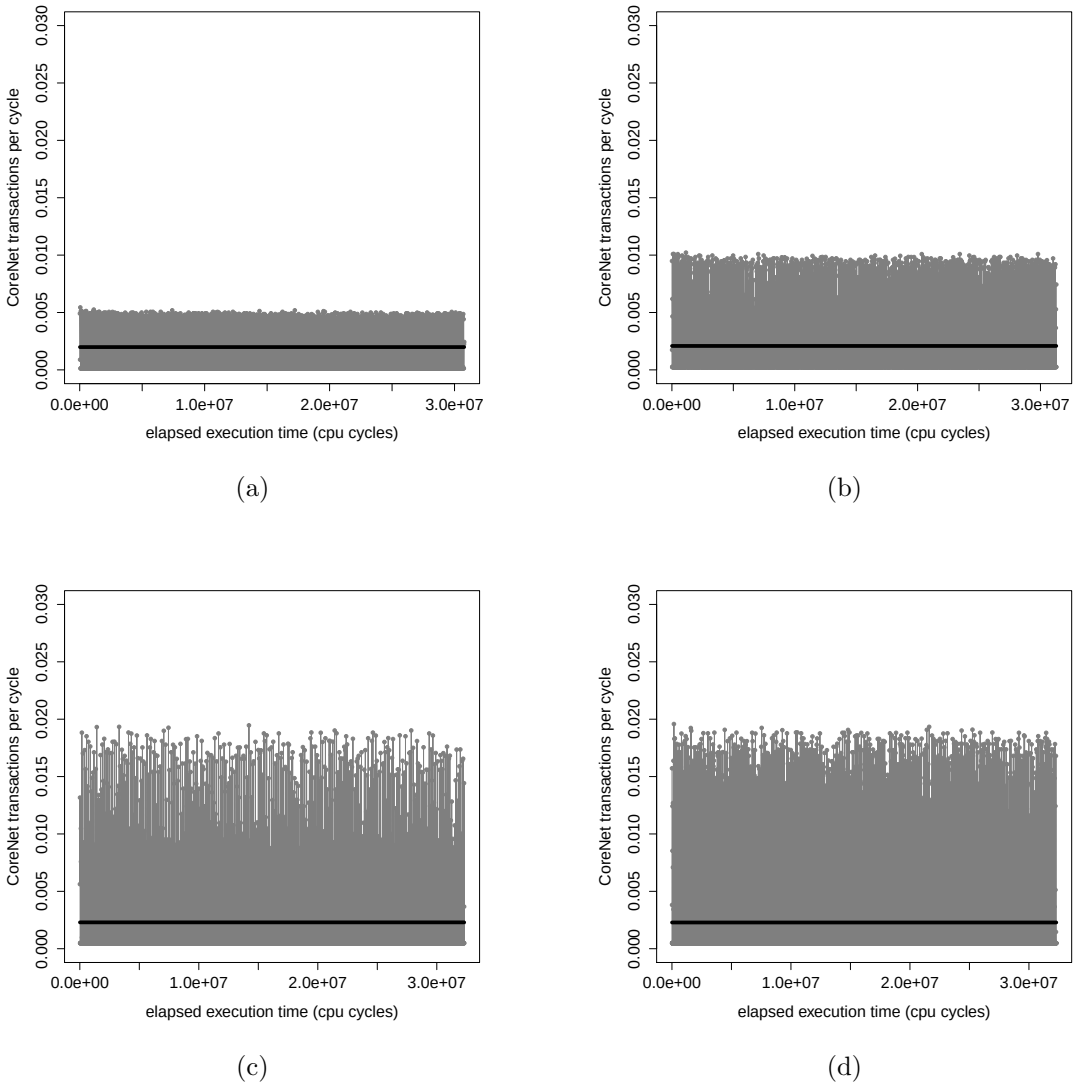


Figure D.4.2: *Variation of the number of CoreNet transaction per cpu cycle during ADPCM full run using time slot (a) T, (b) T/2, (c) T/4 and (d) T/8. The black line denotes the mean value of the CoreNet transaction per cpu cycle.*

In Figure D.4.2, the gray curve refers to the variation of BIU/cycle collected during the full run of ADPCM, and the black line denotes the mean value of BIU/cycle which we used to compare with ADPCM's global BIU/cycle 0.001871 listed in Table D.3.1, allowing us to discover the overhead of interrupt. The mean value in Figure D.4.2(a), (b), (c) and (d) is successively 0.001964, 0.002124, 0.002196, 0.002206 which produced an overhead of 5.0%, 13.5%, 17.4% and 17.9%,

all below unacceptable level 20%. The maximum value in (a), (b) and (c) is nearly doubled every time, and stays similar in (c) with in (d). Accordingly, the appropriate time slot for BIU/cycle of ADPCM is T/8 and the peak BIU/cycle is 0.0198.

In addition, we infer from Figure D.4.2 the advantage of local signatures over global signatures. The local signature is able to capture the chaotic variation behavior of the metric, also the worst case behavior, while the global signature can only capture the average value denoted by the black line.

We used the mechanism of selecting the appropriate time slot for all the metrics of each application to capture their local signature. Figure D.4.3, Figure D.4.4 and Figure D.4.5 respectively depicts the variation of CoreNet transactions per cycle (BIU/cycle), DDR read per cycle (DDRr/cycle) and DDR write per cycle (DDRw/cycle) for applications.

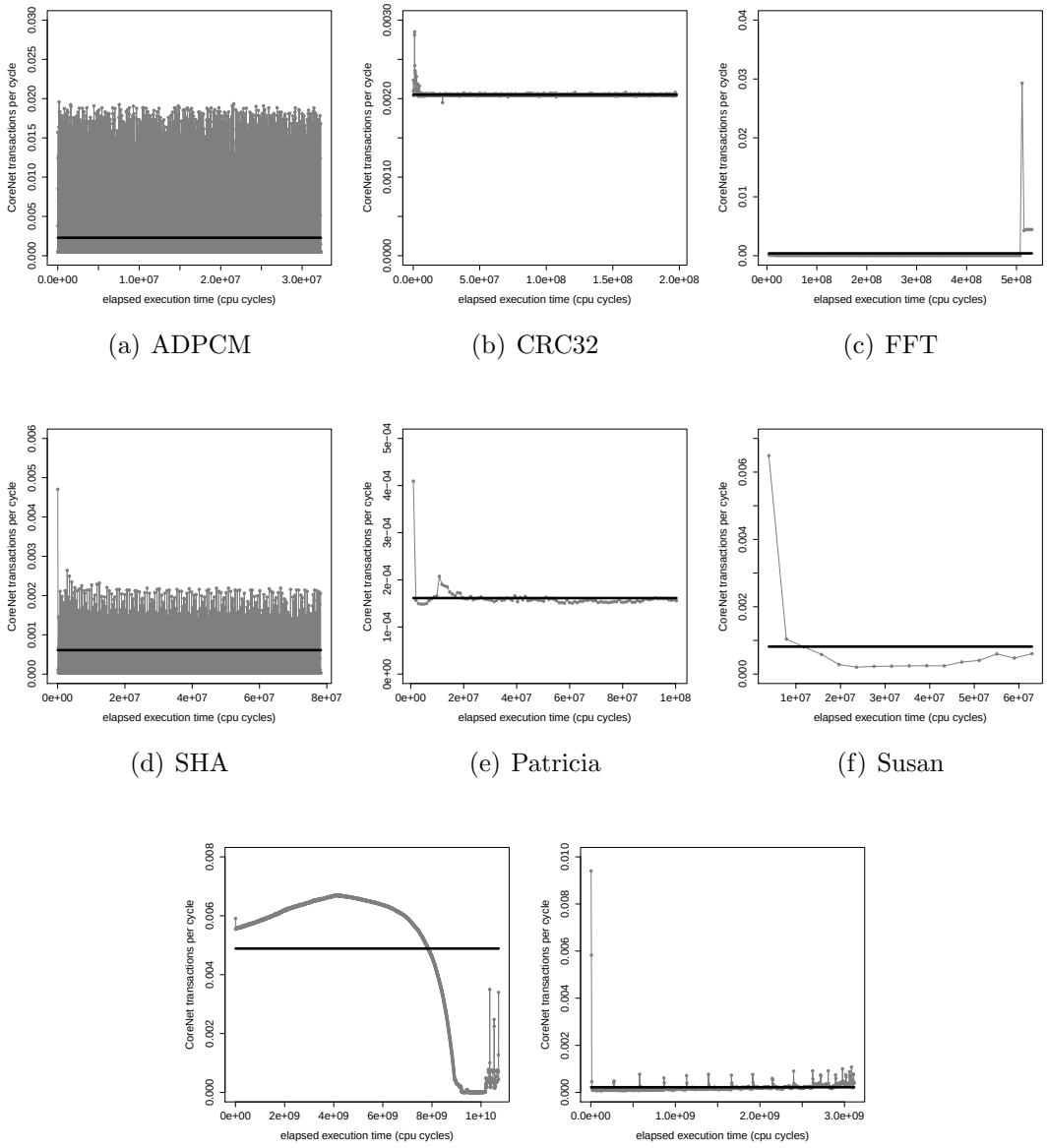
In Figure D.4.3, Figure D.4.4 and Figure D.4.5, the gray curve refers to the variation of normalised metric, and the black line denotes the mean value.

For the behavior on the CoreNet in Figure D.4.3, we can classify the phase behavior into three types: the first type, as ADPCM and SHA, performs a chaotic behavior with a large disparity between the average and most of the peak values (nearly $\times 10$ for ADPCM, $\times 3$ for SHA). The second type, as susan and airborne radar, provides a smoothly changed behavior with the average value not far from most of the local values (about $\times 1.8$ for susan, $\times 1.3$ for airborne radar). The third type, as CRC32, FFT, patricia and pedestrian detection, has a stable or regular behavior around the average line with occasional peak values (i.e. occasional little mountains in the beginning of CRC32 and patricia, a lonely peak just before the end of FFT, a peak in the beginning of pedestrian detection in addition with little hills periodically around the average behavior).

For the behavior on DDRr and DDRw in Figure D.4.4 and Figure D.4.5, all the applications except airborne radar has a behavior on DDRr and DDRw both similar with the CoreNet. Airborne radar has DDRr similar with the CoreNet while a regular and low DDRw with a lonely peak and a noisy mountains in the beginning and in the end of the execution.

This detailed local information can help us understand the failure of over-bounding the WCET in Section D.3.2. We observed two categories of applications according to the estimation in Figure D.3.1: the first category with all the applications without ADPCM and the second with ADPCM. ADPCM failed not only in over bounding the WCET for all the target applications but also in providing an accurate estimation of the WCET, which is charged to its chaotic behavior and upto $\times 10$ gap between the average and the worst case.

In a safety critical context it is critical to successfully capture this worst case behavior. Local signatures allow us both to capture phase behavior of benchmarks while also providing per-metric upper bound for this worst case.

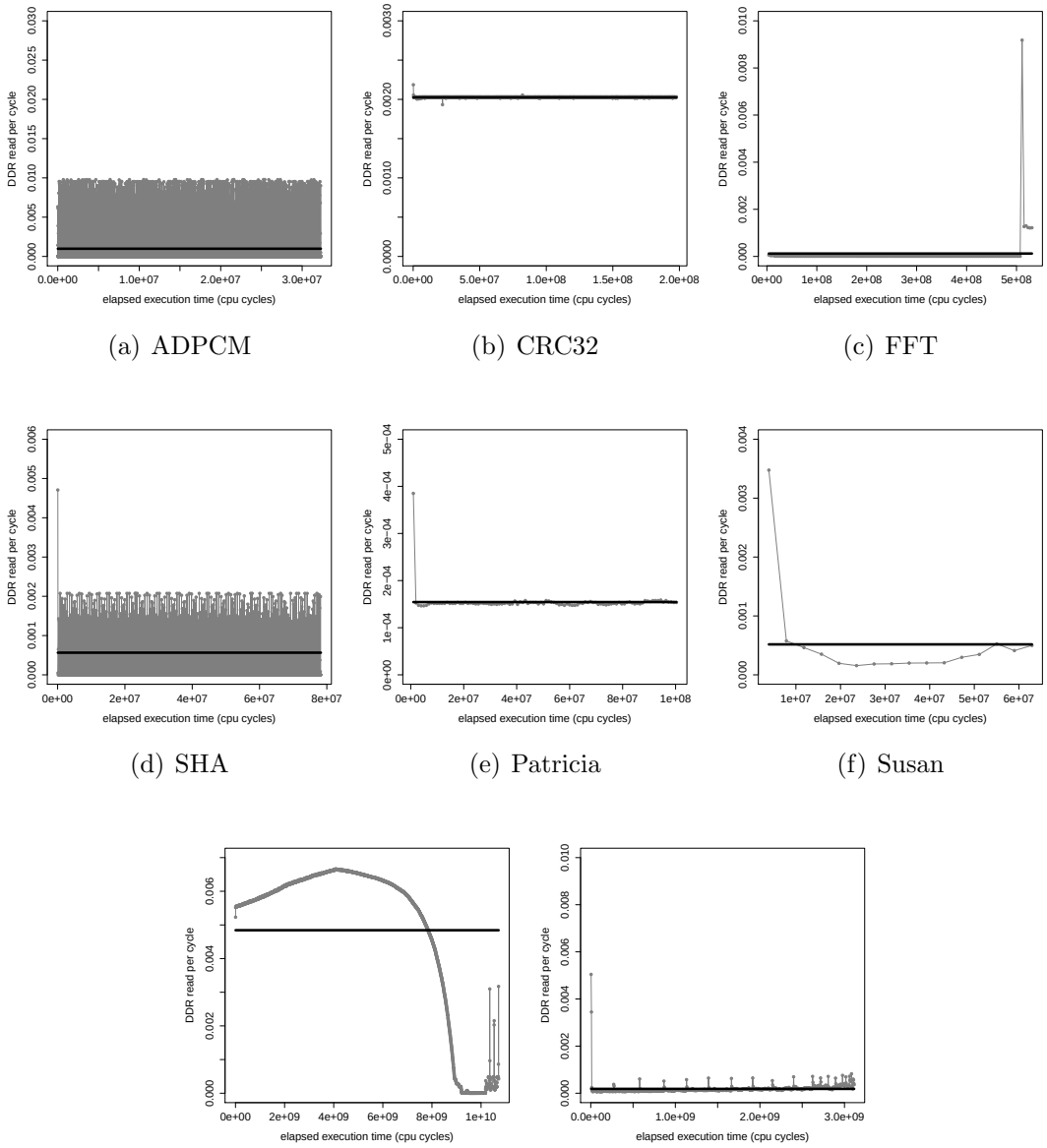


(a) ADPCM (b) CRC32 (c) FFT

(d) SHA (e) Patricia (f) Susan

(g) Airborne radar (h) Pedestrian detection

Figure D.4.3: *Variation of the number of CoreNet transaction per cycle during the full run of (a) ADPCM, (b) CRC32, (c) FFT, (d) SHA, (e) Patricia, (f) Susan, (g) Airborne radar, (h) Pedestrian detection. The black line denotes the mean value of the collected metric.*

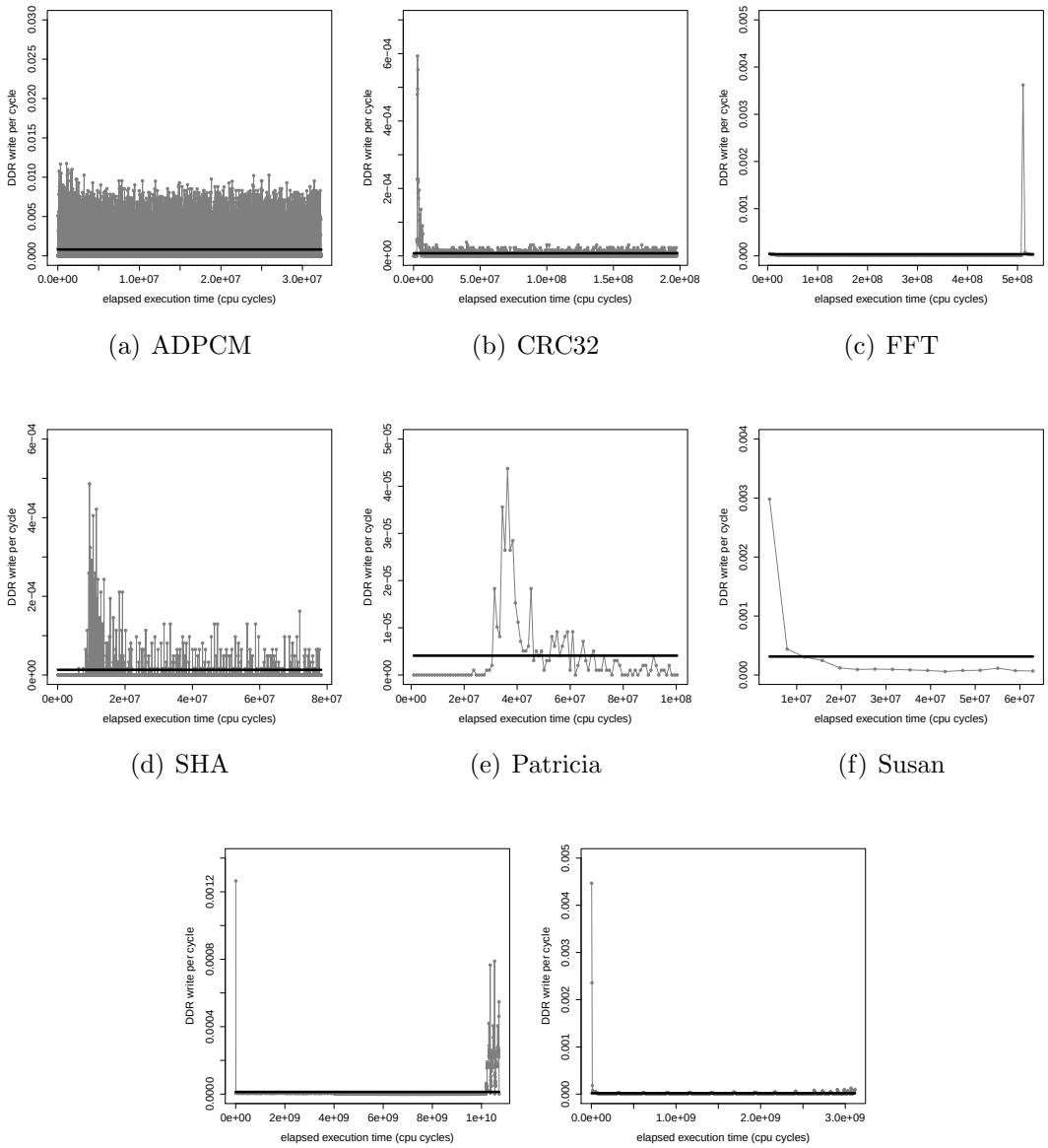


(a) ADPCM (b) CRC32 (c) FFT

(d) SHA (e) Patricia (f) Susan

(g) Airborne radar (h) Pedestrian detection

Figure D.4.4: *Variation of the number of DDR read per cycle during the full run of (a) ADPCM, (b) CRC32, (c) FFT, (d) SHA, (e) Patricia, (f) Susan, (g) Airborne radar, (h) Pedestrian detection. The black line denotes the mean value of the collected metric.*



(a) ADPCM (b) CRC32 (c) FFT

(d) SHA (e) Patricia (f) Susan

(g) Airborne radar (h) Pedestrian detection

Figure D.4.5: *Variation of the number of DDR write per cylce during the full run of (a) ADPCM, (b) CRC32, (c) FFT, (d) SHA, (e) Patricia, (f) Susan, (g) Airborne radar, (h) Pedestrian detection. The black line denotes the mean value of the collected metric.*

D.4.2 Using Local Signatures

While global signatures were only able to capture the average usage of each shared hardware resource during the whole application runtime, local signatures enable us to capture peak hardware resource usage at time slot level by selecting the maximum value independently for each normalised hardware counter among the succession of monitored vector values.

We can then create a signature composed of these maximum values, and identify a stressing benchmark with a similar upper bound signature, as we have done for global signatures. This identified stressing benchmark, appearing in Table D.4.1 which is more pessimistic than results from global signatures in Table **??**, should behave as the original application constantly performing peak access to its shared hardware resource. As a consequence, it is much more likely to be an upper bound of the original application.

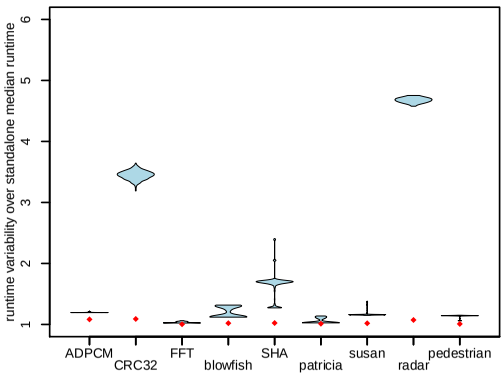
Application	BIU/cycle	DDRr/10^2cycle	DDRw/10^2cycle
ADPCM	0.0198	0.9900	0.9900
CRC32	0.0029	0.2210	0.0690
FFT	0.0300	0.9300	0.3900
SHA	0.0048	0.4272	0.0528
patricia	0.0004	0.0383	0.0044
susan	0.0065	0.3464	0.3027
airborne radar	0.0068	0.5400	0.1360
pedestrian detection	0.0096	0.5126	0.4481

Table D.4.1: *Stressing benchmarks identified with local signatures*

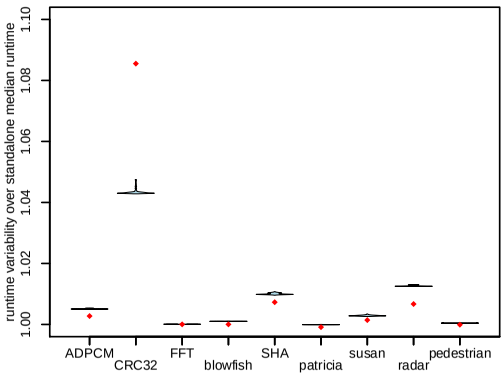
To evaluate this new stressing benchmark candidate, we ran again each target application on the 4-core cluster together with 3 instances of the original application and afterwards with 3 instances of the identified closest stressing benchmark as we have done for global signatures. Figure D.4.6 illustrates both the maximum performance slowdown observed while three co-running original applications with a single red dot, and the different possible slowdowns collected within successive iterations while three co-running identified stressing benchmarks by a blue violin plot.

This time mostly every violin plot completely appear above the mark, indicating a successful upper bounding, at the cost of a larger gap with this upper-bound.

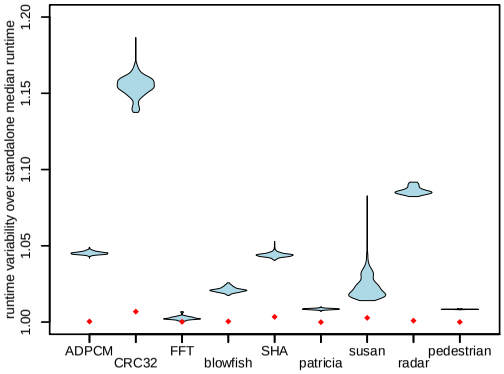
Figure D.4.6(c)(e)(f)(g)(h) are respectively evaluating the results of stressing benchmark candidate of FFT, patricia, susan, airborne radar and pedestrian detection, now all of them displays systematic over bounding of the violin shapes against the maximum observed runtime mark, exhibiting 100% successful over-bounding ability. The safety margin of the over-bound increased compared to global signatures from a maximum of 0.66% to 13% for FFT, 0.02% to 0.21% for patricia, 0.96% to 13.57% for susan, 1.46% to 7.93% for airborne radar and



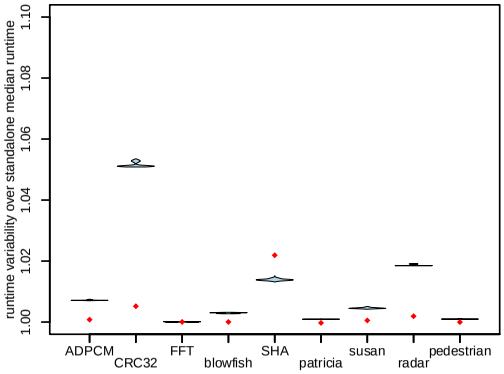
(a) ADPCM



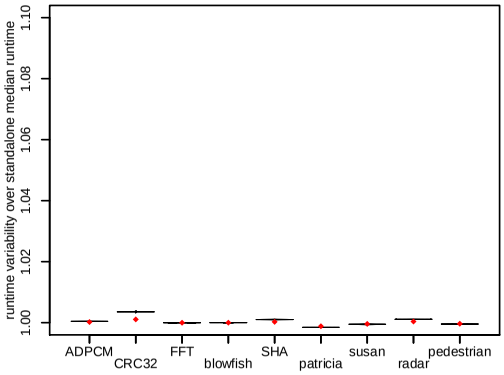
(b) CRC32



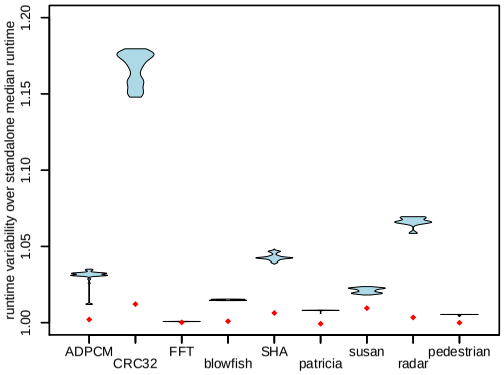
(c) FFT



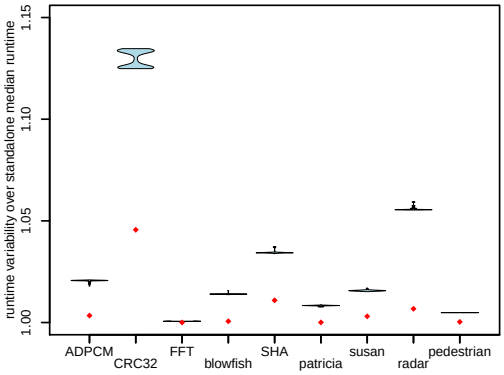
(d) SHA



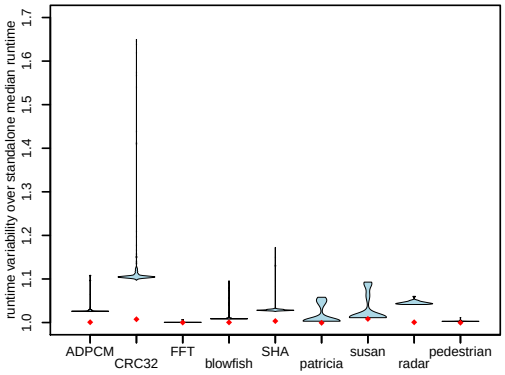
(e) Patricia



(f) Susan



(g) Airborne radar



(h) Pedestrian detection

Figure D.4.6: *Evaluating the local signatures against the performance slowdown induced by co-running with 3 instances of ADPCM, CRC32, FFT, SHA, patricia, susan, airborne radar, pedestrian detection versus their equivalent stressing benchmarks. Blue violin plots represent the runtime variability while co-running with stressing benchmarks. The red marks denote the maximum runtime while co-running with the original applications.*

0.88% to 8.95% for pedestrian detection. However this over-margin remains far below the maximum margin ×5 identified by co-running a target application with resource stressing benchmarks in Figure C.6.2 and the worst case performance slowdown ×4 (the WCET should be multiplied by a value close to the number of cores being used) concluded in [22].

Figure D.4.6(b)(d) are showing similar results for the CRC32 and SHA stressing benchmark. While providing a successful upper bound for 8 out of 9 applications, CRC32 fails to over bound while co-running with the CRC32 application itself with a lower over margin 4.27% compared to 6.67% using global signatures. The SHA stressing benchmark is exhibiting exactly the same behavior also successfully upper-bounding for every application but itself, and it failed also with a lower margin 0.88% compared to previous 2.11%.

Finally, for ADPCM presented in Figure D.4.6(a) which we were previously unable to approximate with the global signatures due to its chaotic behavior, we are now able to systematically over bound its WCET, but with a very large safety margin (between ×3 and ×5) for CRC32 and the airborne radar applications.

Table D.4.2 summarizes evaluation results for every identified stressing applications, indicating first, the accuracy in terms of average safety margin to the over bound, and second, the average over-bounding ability.

Over the 81 evaluations performed, we obtained an average over-margin of 12.1% and managed to successfully upper-bound in 97.5% of the cases.

Application	over-margin value	over-bounding ability
ADPCM	69.7%	100%
CRC32	6.26%	88.9%
FFT	3.71%	100%
SHA	9.51%	88.9%
patricia	0.45%	100%
susan	2.96%	100%
airborne radar	2.26%	100%
pedestrian detection	2.16%	100%

Table D.4.2: *Evaluating local signature accuracy in terms of over-margin value and over-bounding ability.*

Compared to the state of the art, we managed to control the over margin, but haven't yet achieved the ultimate systematic over bounding, though being now very close. One of the reasons that we cannot achieve 100% over bounding is that we restricted ourselves into a few hardware monitors to represent the application's behavior on shared resources, which may not be sufficient to capture the worst case performance slowdown caused by other non-collected activities. The extending future work regarding to this point will be presented in Chapter E.2.

Chapter D.5

Conclusion

In this part, we have presented a technique relying on stressing benchmarks to easily compute the execution time upper bound for an application running on a multi-core alongside a set of pre-determined applications. We created a stressing benchmark to exhibit an upper bound behavior on shared resources of a pre-determined application using:

- global signatures which only capture an average behavior of the application, preventing us to simulate the worst case contention on shared resources caused by the worst case behavior of the original application. Therefore, global signatures are not able to systematically over bound the WCET, but provide a good estimation of the runtime for applications whose behavior are not too chaotic.

- local signatures relying on sampling techniques which capture the phase behavior including the worst case behavior on shared resources of the application. They thus allow us to over bound the WCET thanks to the ability of simulating the worst case interference on shared resources.

This technique enables us to compute a tight WCET margin of a targeted application with a single run on the multi-core, allowing the development of the safety critical system to rapidly and easily determine if a given set of applications can run together while respecting their deadlines. Moreover, the proposed technique is well adapted for the use in an industrial environment, where the combination of different applications is typically limited and controlled.

Our initial evaluation shows that with only three monitored normalised hardware counters: BIU/cycle, DDRr/cycle and DDRw/cycle, we are able to generate representative stressing benchmarks. The proposed technique using the generated stressing benchmarks is able to compute the execution time upper bound with an average over-margin of 12.1%, with the worst case over-margin of 350%. In our

experiments the technique failed to over bound WCETs in two occassions (2.5% of the total number of tests), which we expect to solve in the future work that we present in Chapter E.2.

Part E

Conclusion

Chapter E.1

Conclusion

In the thesis, to achieve the overall objective - estimate the runtime variability of co-running safety-critical applications on a multi-core COTS, we proposed first a methodology to characterize the target architecture and applications, and second an alternative method to estimate the WCET of co-running applications. To better understand the potential runtime variability that co-running applications may experience on a multi-core COTS, we quantified the variability of each target application while co-running with a set of resource stressing benchmarks in the target P4080 platform. The experimental results demonstrated a large variability upto 396% due to the contention on shared hardware resources. To be able to control such variability which would lead to unsustainable WCET margin far above the performance benefits of multi-cores, we proposed a methodology to characterize the architecture and target applications regarding shared hardware resources.

During the characterizations, relying on hardware monitors and stressing benchmarks we successfully

- identified which hardware resources are effectively shared
- discovered undisclosed features of the architecture - two 4-core clusters, which helped select the mapping leading to the lowest variability.
- identified the optimal hardware configuration for safety-critical applications according to two criteria: the low variability and sufficient overall performance.
- identified the adequate mapping for mixed-critical applications and all-critical applications.
- quantified shared hardware resource availability: the maximum throughput, the saturation behavior and the topology of the CoreNet and the DDR

controller, which helped compute the CoreNet and DDR resource usage in the application characterization.

During the application characterization, we relied on the identified optimal hardware configuration to

- identify the sensitivity of each application to each shared resource, which allowed us to understand upto which extent the application is sensitive to each resource and upto which extent the performance of the application may be slown down by the contention on each shared resource.
- compute the resource usage of each standalone application. A shared resource usage can accurately tell us how many percentages of the resource (compared to its saturation value) is required by the application.
- identify possible co-running applications using resource usages. Applications whose total usage of each shared resource keeps below the corresponding saturation value can run together without significantly endangering each other's deadline.

Based on the characterizations, we presented a technique relying on stressing benchmarks to easily compute the execution time upper bound for an application running on a multi-core alongside a set of pre-determined applications. We created a stressing benchmark to exhibit an upper bound behavior on shared resources of a pre-determined application using:

- global signatures which only capture an average behavior of the application, preventing us to simulate the worst case contention on shared resources caused by the worst case behavior of the original application. Therefore, global signatures are not able to systematically over bound the WCET, but provide a good estimation of the runtime for applications whose behavior are not too chaotic.
- local signatures relying on sampling techniques which capture the phase behavior including the worst case behavior on shared resources of the application. They thus allow us to over bound the WCET thanks to the ability of simulating the worst case interference on shared resources.

This technique enables us to compute a tight WCET margin of a targeted application with a single run on the multi-core, allowing the development of the safety critical system to rapidly and easily determine if a given set of applications can run together while respecting their deadlines. Moreover, the proposed technique is well adapted for the use in an industrial environment, where the combination of different applications is typically limited and controlled.

Our initial evaluation shows that with only three monitored normalised hardware counters: BIU/cycle, DDRr/cycle and DDRw/cycle, we are able to generate representative stressing benchmarks. The proposed technique using the generated stressing benchmarks is able to compute the execution time upper bound with an average over-margin of 12.1%, with the worst case over-margin of 350%. Compared to the state of the art, our approach managed to better control the over margin by simulating the worst case contention on shared resources that can be produced by real co-running applications but not only using resource stressing benchmarks to achieve the most pessimistic case [22, 28].

Chapter E.2

Future work

In the experiments, the technique to estimate the WCET failed in over-bounding in two occassions (2.5% of the total number of tests), which can be concluded into one main reason: according to the characterization results on shared resources under the optimal configuration, we intuitively selected three normalised hardware monitors (BIU/cycle, DDRr/cycle and DDRw/cycle) to represent an application's access behavior on the CoreNet and DDR and to identify a stressing benchmark replacing the application, which means that we are only able to estimate the worst case performance slowdown caused by the access contention described by these monitors. However, there may be other hardware monitors being responsible for the performance variability that we didn't take account of, like DDR page closing. For example, for the application CRC32, it has the lowest overall DDR page closing frequency compared to other eight applications through our experiments, which means that it has the most efficient memory access reference pattern to avoid the page switches. When it runs with other applications, this most efficient pattern may be disturbed by memory accesses of co-runners due to the sharing mechanism. In this case, additional increasing page switches may dominate the performance slowdown of the CRC32. Since we didn't apply the event *DDR page closing* to identify the CRC32's replacement stressing benchmark, we are not able to estimate the performance slowdown given by the event.

Therefore, to ameliorate the over-bounding ability of our technique, we should develop a framework automatically detecting the useful hardware monitors responsible for the variability before the estimation. Then, a more representative stressing benchmark can be identified using detected useful monitors to completely simulate the worst case contention on variability sources, which will allow to over bound the WCET in the co-running context.

In addition to 100% over-bounding ability, we are considering to reduce the over margin obtained in our technique. The maximum average over margin from

our experimental results is 69.7% while co-running a target application with several ADPCM. That's because ADPCM's peak CoreNet and DDR usages are both much larger than the global ones due to its too chaotic local behavior and we only take the peak usages to design its replacement stressing benchmark. The stressing benchmark is thus too pessimistic on resource usages, which leads to a large over margin in the estimation. To control the over margin, we can additionally take account of each resource usage distribution and design a stressing benchmark to have a more pessimitic distribution over the original one to provide a tighter upper bound. However, we should admit that making a more pessimitic distribution is inherently difficult for some local behavior without evident phases.

Part F

Appendix

Source Code of Stressing benchmark

Listing 1 demonstrates an example source code of a stressing benchmark traversing a integer table with parameters below:

- STRIDE: 1 element = 4 bytes
- OPERATION: write
- NOP: 1
- UNROLL: 8

Listing 1: *Example source code of stressing benchmark*

```
//r20 and r15 respectively store the beginning and the
   end address of the integer table
//r16 controls the outside forever loop
li 16, 0
//r17 is the pointer of the current table address,
   controls the inside loop (one travers of the whole
   table)
loopforever: mr 17, 20
//STRIDE is 1element=4bytes, UNROLL=8
loop: stw 18, 0(17)
nop
stw 18, 4(17)
nop
stw 18, 8(17)
nop
stw 18, 12(17)
nop
stw 18, 16(17)
```

```
nop
stw 18, 20(17)
nop
stw 18, 24(17)
nop
stw 18, 28(17)
//shift the pointer r17 to the next 8 element
addi 17, 17, 32
//determine if the end of the table
cmplw 17,15
blt loop
// r16=0<1, the loop condition is always true
cmpwi 16,1
blt loopforever
```

TCL Script of Automating Debugging Session

Listing 2 demonstrates an example of TCL script to automate a debugging session where there are projects in core #0, core #1 and core #4. The core #1 is under monitor and send the keyword *over* to inform the execution end and it also send the start and end address of the memory area storing the results data to let us save the data from the memory into the destination file. The blue words in Listing 2 are the keywords of TCL and the red ones refer to the shell commands of CodeWarrior.

Listing 2: *Example TCL script of automating debugging session*

```
# launch a debug session for selected projects in core0,
    core1 and core4
debug
   initialization_2L3Partition-core0_RAM_P4080_Cache_Download

wait 2000
debug stress_bm-core1_RAM_P4080_Cache_Download
wait 2000
debug stress_bm-core4_RAM_P4080_Cache_Download
wait 2000

# start debug
mc::go
wait 1000

# detect the end of the execution of project under
   monitor by surveiling the keyword "over" in the log
   file Flag.txt of the PuTTY
set f [open C:/Flag.txt]
gets $f line
gets $f line
```

```
while {[string match *over* $line]<=0} {
    puts stdout "wait"
    puts $line
    close $f
    wait 1000
    set f [open C:/Flag.txt]
    gets $f line
    gets $f line}

# stop debug
mc::stop

# get the start and end address of data storage area
gets $f saddr
gets $f eaddr

# save the data from the memory into the destination
    file
save -b "p:0x$saddr..0x$eaddr" ./folder/filename.txt -o
puts stdout "successful"

# kill debug session
mc::kill
puts $line
close $f
puts stdout "finish in 5secs"
wait 5000

# quit CodeWarrior IDE
quitIDE
```

Python Script of Automating Experiments

Listing 3 demonstrates an example of Python script to automate the whole experiments mapping one application with two stressing benchmarks in different cores. In the scenarios generation function, we create different scenarios with different mappings, different parameters of stressing benchmarks and different applications under monitor. In the configuration function, we modify the source file and header file of the current stressing benchmark and compile them. The application projects are pre-programmed and compiled statically, there is no need to modify them. In the execution function, we configure all the projects using the configuration function and create TCL script to automate the debugging session within the CodeWarrior. As we presented in Section C.4.3.2, to avoid the influence of the collapse of CodeWarrior, we create a background thread to kill the CodeWarrior by force after waiting a threshold time.

Listing 3: *Example Python script of automating experiments*

```
import os
import signal
import subprocess
import shutil
import thread
import threading
import time

was_killed = False
APP_core = 0

# scenarios generation process function
def generate_scenario():
    scenario = {}
    i = 0
    number_APP = 1
```

```
    name_APP = "CRC32"
    while number_APP<10:
        stride = 1
        while stride <= 16:
            tablesize = 49152
            while tablesize <= 262128:
                scenario[i] = {'core0': {'BM': 'false'},
                               'core1': {'BM': 'true', '
                                 APP': 'true', '
                                 name_APP': name_APP, '
                                 save': str(name_APP)+'
                                 1+2SB_write_'+str(
                                 tablesize)+'_'+str(
                                 stride)},
                               'core2': {'BM': 'true', '
                                 APP': 'false', '
                                 tablesize': str(
                                 tablesize), 'stride':
                                 stride},
                               'core3': {'BM': 'true', '
                                 APP': 'false', '
                                 tablesize': str(
                                 tablesize), 'stride':
                                 stride},
                               'core4': {'BM': 'false'},
                               'core5': {'BM': 'false'},
                               'core6': {'BM': 'false'},
                               'core7': {'BM': 'false'}
                              }
                i = i+1
                scenario[i] = {'core0': {'BM': 'false'},
                               'core1': {'BM': 'true', '
                                 APP': 'true', '
                                 name_APP': name_APP, '
                                 save': str(name_APP)+'
                                 1+2SB_write_'+str(
                                 tablesize)+'_'+str(
                                 stride)},
                               'core2': {'BM': 'false'},
                               'core3': {'BM': 'true', '
                                 APP': 'false', '
```

```
                                    tablesize': str(
                                    tablesize), 'stride':
                                    stride},
                                'core4': {'BM': 'true', '
                                    APP': 'false', '
                                    tablesize': str(
                                    tablesize), 'stride':
                                    stride},
                                'core5': {'BM': 'false'},
                                'core6': {'BM': 'false'},
                                'core7': {'BM': 'false'}
                                }
            i = i+1
            scenario[i] = {'core0': {'BM': 'false'},
                                'core1': {'BM': 'true', '
                                    APP': 'true', '
                                    name_APP': name_APP, '
                                    save': str(name_APP)+'
                                    1+2SB_write_'+str(
                                    tablesize)+'_'+str(
                                    stride)},
                                'core2': {'BM': 'false'},
                                'core3': {'BM': 'false'},
                                'core4': {'BM': 'true', '
                                    APP': 'false', '
                                    tablesize': str(
                                    tablesize), 'stride':
                                    stride},
                                'core5': {'BM': 'true', '
                                    APP': 'false', '
                                    tablesize': str(
                                    tablesize), 'stride':
                                    stride},
                                'core6': {'BM': 'false'},
                                'core7': {'BM': 'false'}
                                }
            i = i+1
            scenario[i] = {'core0': {'BM': 'false'},
                                'core1': {'BM': 'true', '
                                    APP': 'true', '
                                    name_APP': name_APP, '
```

```
                                    save': str(name_APP)+'
                                    1+2SB_write_'+str(
                                    tablesize)+'_'+str(
                                    stride)},
                                'core2': {'BM': 'false'},
                                'core3': {'BM': 'false'},
                                'core4': {'BM': 'false'},
                                'core5': {'BM': 'true', '
                                    APP': 'false', '
                                    tablesize': str(
                                    tablesize), 'stride':
                                    stride},
                                'core6': {'BM': 'true', '
                                    APP': 'false', '
                                    tablesize': str(
                                    tablesize), 'stride':
                                    stride},
                                'core7': {'BM': 'false'}
                               }
            i = i+1
            scenario[i] = {'core0': {'BM': 'false'},
                                'core1': {'BM': 'true', '
                                    APP': 'true', '
                                    name_APP': name_APP, '
                                    save': str(name_APP)+'
                                    1+2SB_write_'+str(
                                    tablesize)+'_'+str(
                                    stride)},
                                'core2': {'BM': 'false'},
                                'core3': {'BM': 'false'},
                                'core4': {'BM': 'false'},
                                'core5': {'BM': 'false'},
                                'core6': {'BM': 'true', '
                                    APP': 'false', '
                                    tablesize': str(
                                    tablesize), 'stride':
                                    stride},
                                'core7': {'BM': 'true', '
                                    APP': 'false', '
                                    tablesize': str(
                                    tablesize), 'stride':
```

```
                                stride}
                            }
                i = i+1
                scenario[i] = {'core0': {'BM': 'true', '
                    APP': 'false', 'tablesize': str(
                    tablesize), 'stride': stride},
                            'core1': {'BM': 'true', '
                                APP': 'true', '
                                name_APP': name_APP, '
                                save': str(name_APP)+'
                                1+2SB_write_'+str(
                                tablesize)+'_'+str(
                                stride)},
                            'core2': {'BM': 'false'},
                            'core3': {'BM': 'false'},
                            'core4': {'BM': 'false'},
                            'core5': {'BM': 'false'},
                            'core6': {'BM': 'false'},
                            'core7': {'BM': 'true', '
                                APP': 'false', '
                                tablesize': str(
                                tablesize), 'stride':
                                stride}
                            }
                i = i+1
                scenario[i] = {'core0': {'BM': 'true', '
                    APP': 'false', 'tablesize': str(
                    tablesize), 'stride': stride},
                            'core1': {'BM': 'true', '
                                APP': 'true', '
                                name_APP': name_APP, '
                                save': str(name_APP)+'
                                1+2SB_write_'+str(
                                tablesize)+'_'+str(
                                stride)},
                            'core2': {'BM': 'true', '
                                APP': 'false', '
                                tablesize': str(
                                tablesize), 'stride':
                                stride},
                            'core3': {'BM': 'false'},
```

```
                                    'core4': {'BM': 'false'},
                                    'core5': {'BM': 'false'},
                                    'core6': {'BM': 'false'},
                                    'core7': {'BM': 'false'}
                                    }
                    i = i+1
                    if tablesize == 49152:
                        tablesize = 65536
                    elif tablesize == 65536:
                        tablesize = 262128
                    elif tablesize == 262128:
                        tablesize = tablesize+1
                if stride == 1:
                    stride = stride*16
                elif stride == 16:
                    stride = stride + 1
            if name_APP == "CRC32":
                name_APP = "Adpcm"
            elif name_APP == "Adpcm":
                name_APP = "Sha"
            elif name_APP == "Sha":
                name_APP = "Blowfish"
            elif name_APP == "Blowfish":
                name_APP = "Patricia"
            elif name_APP == "Patricia":
                name_APP = "Susan"
            elif name_APP == "Susan":
                name_APP = "FFT"
            elif name_APP == "FFT":
                name_APP = "STAP"
            elif name_APP == "STAP":
                name_APP = "Violajones"
            number_APP = number_APP + 1
        return scenario


# configuration process function
def configurate_core(number_core, scenario_experiment, i
    , debug):
    global APP_core
    # if there is a benchmark in the current core
```

```
    if ((scenario_experiment.get(i)).get('core'+str(
       number_core))).get('BM') == 'true':
        # if there is a stressing benchmark in the
           current core
        if ((scenario_experiment.get(i)).get('core'+str(
           number_core))).get('APP') == 'false':
            # there is indeed a stressing benchmark in
               the core, so open the source code file
               and modify it according to the scenario's
                parameters
            f = open("C:\Users\Jingyi\workspace\
               stress_bm-core"+str(number_core)+"\Source
               \main.c", 'w')
            f.write("#include <stdio.h>\n"
                "#include \"P4080_Registers.h\"\n"
                "#include \"global_variables.h\"\n"
                "#define stride "+str(((
                   scenario_experiment.get(i)).get('core
                   '+str(number_core))).get('stride'))+"
                   \n"
                "int32_t mem[tablesize];\n"
                "char *pos_write;\n"
                "char tab[40000];\n"
                "int main()\n"
                "{\n"
                "   int i=0, j=2;\n"
                "   pos_write = tab;\n"
                "   /* enable external interrupts */\n"
                "   asm volatile(\n"
                "   \"mfmsr 0\\n\\t\"\n"
                "   \"ori 0, 0, 0xA000\\n\\t\"\n"
                "   \"mtmsr 0\"\n"
                "   ::: \"r0\"\n"
                "   );\n"
                "   mem[0]=&mem[tablesize-1]+1;\n"
                "   for(i=1;i<tablesize;i++)\n"
                "   {\n"
                "      mem[i] = j;\n"
                "      j++;\n"
                "   }\n"
                "   asm volatile(\n"
```

```
            "   \"li 16, 0x0\\n\\t\"\n"
            "   \"li 18, 0x1\\n\\t\"\n"
            "   \"li 19, 0x0\\n\\t\"\n"
            "   \"li 21, 0x0\\n\\t\"\n"
            "   \"mr 20, %0\\n\\t\"\n"
            "   \"lwzx 15, 16, 20\\n\\t\"\n"
            "   \"stwx 16, 16, 15\\n\\t\"\n"
            "   \"loopforever: mr 17, 20\\n\\t\"\n"
            "   \"loop: stw 18, 0(17)\\n\\t\"\n"
            "   \"stw 18, "+str ((((
               scenario_experiment.get(i)).get('core
               '+str(number_core))).get('stride'))
               *4)+"(17)\\n\\t\"\n"
            "   \"stw 18, "+str ((((
               scenario_experiment.get(i)).get('core
               '+str(number_core))).get('stride'))
               *8)+"(17)\\n\\t\"\n"
            "   \"stw 18, "+str ((((
               scenario_experiment.get(i)).get('core
               '+str(number_core))).get('stride'))
               *12)+"(17)\\n\\t\"\n"
            "   \"stw 18, "+str ((((
               scenario_experiment.get(i)).get('core
               '+str(number_core))).get('stride'))
               *16)+"(17)\\n\\t\"\n"
            "   \"stw 18, "+str ((((
               scenario_experiment.get(i)).get('core
               '+str(number_core))).get('stride'))
               *20)+"(17)\\n\\t\"\n"
            "   \"stw 18, "+str ((((
               scenario_experiment.get(i)).get('core
               '+str(number_core))).get('stride'))
               *24)+"(17)\\n\\t\"\n"
            "   \"stw 18, "+str ((((
               scenario_experiment.get(i)).get('core
               '+str(number_core))).get('stride'))
               *28)+"(17)\\n\\t\"\n"
            "   \"addi 17, 17, "+str ((((
               scenario_experiment.get(i)).get('core
               '+str(number_core))).get('stride'))
               *32)+"\\n\\t\"\n"
```

```
                "    \"cmplw 17,15\\n\\t\"\n"
                "    \"blt loop\\n\\t\"\n"
                "    \"lwzx 19, 16, 15\\n\\t\"\n"
                "    \"addi 19,19,1\\n\\t\"\n"
                "    \"stwx 19, 16, 15\\n\\t\"\n"
                "    \"cmpwi 16,1\\n\\t\"\n"
                "    \"blt loopforever\"\n"
                "    ::\"r\"(mem):\"r15\",\"r16\",\"r17
                    \",\"r18\",\"r19\",\"r20\",\"r21\");\
                    n"
                "}\n"
                )
        f.close()

        # save this modified source file in a pre-
            selected path for tracing backward in
            case an error
        f = open("C:\Users\Jingyi\workspace\
            stress_bm-core"+str(number_core)+"\Source
            \main.c")
        lines = f.readlines()
        main = open("C:\Users\Jingyi\workspace\
            log_experiment\log_scenario"+str(i)+"\
            main_core"+str(number_core)+".c", 'w')
        main.writelines(lines)
        main.close()
        f.close()

        # modify the header file according to the
            scenario's parameters
        f = open("C:\Users\Jingyi\workspace\
            stress_bm-core"+str(number_core)+"\Source
            \global_variables.h", 'w')
        f.write("#include <stdint.h>\n"
                "#define tablesize "+str(((
                    scenario_experiment.get(i)).get('
                    core'+str(number_core))).get('
                    tablesize'))+"\n"
                "extern char *pos_write;\n"
                "extern int32_t mem[tablesize];\n"
                )
```

```
            f.close()

            # save this modified header file in a pre-
                selected path for tracing backward in
                case an error
            f = open("C:\Users\Jingyi\workspace\
                stress_bm-core"+str(number_core)+"\Source
                \global_variables.h")
            lines = f.readlines()
            global_variables = open("C:\Users\Jingyi\
                workspace\log_experiment\log_scenario"+
                str(i)+"\global_variables"+str(
                number_core)+".h", 'w')
            global_variables.writelines(lines)
            global_variables.close()
            f.close()

            #compile this stressing project and save the
                 compilation output information in a txt
                file for tracing backward
            f_compile = open("C:\Users\Jingyi\workspace\
                log_experiment\log_scenario"+str(i)+"\
                stdout_compile"+str(number_core)+".txt",
                'w')
            compile_file = subprocess.call(["C:\Program
                Files (x86)\Freescale\CW PA v10.1.2\
                eclipse\ecd.exe",
                                           "-build", "-
                                               verbose", "-
                                               project", "C
                                               :\Users\
                                               Jingyi\
                                               workspace\
                                               stress_bm-
                                               core"+str(
                                               number_core)
                                               , "-
                                               cleanBuild"
                                               ],
                                           stdout=
                                               f_compile,
```

```
                                                stderr=
                                                    f_compile)
                f_compile.close()

                # add this stressing benchmark project in
                    debug command parameter
                debug = debug + "debug stress_bm-core"+str(
                    number_core)+"_RAM_P4080_Cache_Download;
                    wait 2000;"

            elif ((scenario_experiment.get(i)).get('core'+
                str(number_core))).get('APP') == 'true':
                # add this application project in debug
                    command parameter
                debug = debug + "debug "+((
                    scenario_experiment.get(i)).get('core'+
                    str(number_core))).get('name_APP')+"-core
                    "+str(number_core)+"
                    _RAM_P4080_Cache_Download;wait 2000;"
                APP_core = number_core
        return debug


# execution process function
def execute_experiment(scenario_experiment,
    start_scenario, end_scenario):
    id_scenario = start_scenario
    #configure core0
    global was_killed
    while id_scenario<end_scenario:
        # create a log folder for saving the current
            execution information for tracing backward
        if os.path.exists("C:\Users\Jingyi\workspace\
            log_experiment\log_scenario"+str(id_scenario)
            )==False:
            os.makedirs("C:\Users\Jingyi\workspace\
                log_experiment\log_scenario"+str(
                id_scenario))
        f_log = open("C:\Users\Jingyi\workspace\
            log_experiment\log_scenario"+str(id_scenario)
            +"\log.log", 'w')
```

```
        f_log.write("begin...\n")

        #verify if core0 is only an initialization or
            not
        if ((scenario_experiment.get(id_scenario)).get('
            core0')).get('BM') == 'false':
            id_core = 1
            debug = "debug initialization stress-
                core0_RAM_P4080_Cache_Download;wait 2000;
                "
        elif ((scenario_experiment.get(id_scenario)).get
            ('core0')).get('BM') == 'true':
            id_core = 0
            debug = ""

        #configurate all the cores according to the
            scenario
        while id_core<8:
            debug = configurate_core(id_core,
                scenario_experiment, id_scenario,debug)
            id_core = id_core+1

        #create tcl script for automating debug
        f = open("C:\Users\Jingyi\workspace\execution.
            tcl", 'w')
        f.write(""+debug+"\n"
                "wait 2000\n"
                "mc::go;\n"
                "wait 1000;\n"
                "set f [open C:/Flag.txt];\n"
                "gets $f line;\n"
                "gets $f line;\n"
                "while {[string match *over* $line]<=0}
                    {\n"
                "    puts stdout \"wait\";\n"
                "    puts $line;\n"
                "    close $f;\n"
                "    wait 1000;\n"
                "    set f [open C:/Flag.txt];\n"
                "    gets $f line;\n"
                "    gets $f line;}\n"
```

```
                "mc::stop;\n"
                "gets $f saddr;\n"
                "gets $f eaddr;\n"
                "save -b \"p:0x$saddr..0x$eaddr\" ./
                    results_cluster_cores/((
                    scenario_experiment.get(id_scenario))
                    .get('core'+str(APP_core))).get('save
                    ')+"_scenario"+str(id_scenario)+".txt
                    " -o;\n"
                "puts stdout \"successful\"\n"
                "mc::kill;\n"
                "puts $line;\n"
                "close $f;\n"
                "puts stdout \"finish in 5secs\n\";\n"
                "wait 5000;\n"
                "quitIDE;"
                )
        f.close()
        # save the tcl file in the log folder for
            tracing backward
        f = open("C:\Users\Jingyi\workspace\execution.
            tcl")
        lines = f.readlines()
        f_tcl = open("C:\Users\Jingyi\workspace\
            log_experiment\log_scenario"+str(id_scenario)
            +"\execution.tcl", 'w')
        f_tcl.writelines(lines)
        f_tcl.close()
        f.close()

        #execute this scenario
        # 1. open the hyperterminal PuTTY
        f = open("C:/Flag.txt", 'w')
        f.write("")
        f.close()
        putty = subprocess.Popen(["C:\Program Files (X86
            )\PuTTY\putty.exe", "-load", "BIN"])
        print "putty launched"

        # 2. open the CodeWarrior IDE
```

```
        cwide = subprocess.Popen(["C:\Program Files (x86
            )\Freescale\CW PA v10.1.2\eclipse\cwide.exe",
                "-vmargsplus", "-Dcw.script=\"C:/Users/
                    Jingyi/workspace/execution.tcl\""])

        # 3. create a backgroud thread as a timer
            configured with a threshold 2400seconds with
            kill_CW function
        was_killed = False
        thread_kill_CW = threading.Timer(2400, kill_CW,
            [cwide, id_scenario])
        print "timer created"
        thread_kill_CW.start()

        # 4. if the CodeWarrior quits normally during
            the threshold time 2400s, jump to the next
            scenario;
        # Otherwise, thread_kill_CW will call function "
            kill_CW" to kill the CodeWarrior by force
        print "cwide.wait"
        cwide.wait()
        thread_kill_CW.cancel()
        print "thread.exit"
        print "scenario"+str(id_scenario)+" finished"
        if was_killed:
            print "scenario " + str(id_scenario) + "
                failed (relaunch)"
            time.sleep(60)
        else:
            id_scenario = id_scenario+1
        putty.terminate()
        f_log.write("end...\n")
        f_log.close()
    return 0


# function: kill the CodeWarrior by force
def kill_CW(cwide, id_scenario):
    print "cwide is dead, killing it (scenario " + str(
        id_scenario) + ")"
    global was_killed
```

```
        was_killed = True
        cwide.terminate()
        subprocess.call(["taskkill","/IM","ccs.exe","/F","/T
            "])
        #make a log file to note kill.cwide
        f_close_CW = open("C:\Users\Jingyi\workspace\
            log_experiment\close_CW_in_force\log"+str(
            id_scenario)+".log", 'w')
        f_close_CW.write("Close CW in force in the scenario"
            +str(id_scenario)+"!")
        f_close_CW.close()

# launch all the scenarios
scenario_experiment = generate_scenario()
execute_experiment(scenario_experiment, 0, 378)
```

References

[1] E. Bailey. Study report on anionics systems for the time frame 2007, 2011 and 2020. *European Organisation for the Safety of Air Navigation (EOSA)*, EUROCONTROL, Nov 2004. 7, 10

[2] T. G. Baker. Lessons learned integrating COTS into systems. In *Proceedings of the First International Conference on COTS-Based Software Systems*, ICCBSS '02, pages 21–30, 2002. 18

[3] D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 274–279, New York, NY, USA, 2011. ACM. 17

[4] F. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. PROARTIS: Probabilistically analysable real-time systems, 2012. 18

[5] D. Dvorak and M. Lyu. NASA study on flight software complexity. *Jet Propulsion*, page 264, May 2009. 7, 10

[6] C. Ebert and C. Jones. Embedded software: Facts, figures and future. *Computer*, 42(4):42–52, April 2009. 4, 7, 10

[7] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 264–265, New York, NY, USA, 2007. ACM. 16, 17, 25

[8] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 25–38, Washington, DC, USA, 2007. IEEE Computer Society. 20

[9] C. Ferdinand and R. Heckmann. aiT: worst case execution time prediction by static program analysis. In *IFIP Congress Topical Sessions*, pages 377–384, 2004. 13

[10] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. Program analysis and compilation, theory and practice. pages 12–52. 2007. 8

[11] Freescale. CodeWarrior Development Tools. http://www.freescale.com/webapp/sps/site/homepage.jsp?code=CW_HOME0. [Online]. 41

[12] Freescale. P4080 Product Summary Page. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080. [Online]. 27

[13] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *EMSOFT*, pages 245–254, 2009. 20

[14] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001*, WWC '01, pages 3–14, 2001. 50

[15] R. Heckmann and C. Ferdinand. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Proceedings of the conference on Design, Automation and Test in Europe*, DATE'05, pages 618–619, 2005. 4

[16] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. In *IEEE Proceedings Computers and Digital Techniques*, 2005. 15

[17] J. L. Hintze and R. D. Nelson. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician*, 52(2):181–184, 1998. 55

[18] R. Kirner and P. Puschner. Obstacles in worst-case execution time analysis. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 333–339, 2008. 8, 11

[19] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner. Using measurements as a complement to static worst-case execution time analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. Dec. 2005. 8, 15

[20] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of*

the 2008 international conference on Compilers, architectures and synthesis for embedded systems, CASES '08, pages 137–146, 2008. 17

[21] E. Mezzetti and T. Vardanega. On the industrial fitness of wcet analysis. In *Proceedings of the 11th International Workshop on Worst Case Execution Time Analysis (WCET2011)*. 2011. 8

[22] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. *European Dependable Computing Conference*, pages 42–52, 2012. 21, 46, 47, 57, 123, 130

[23] R. Pellizzoni and M. Caccamo. Impact of peripheral-processor interference on wcet analysis of real-time embedded systems. *IEEE Trans. Computers*, 59(3):400–415, 2010. 19

[24] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 741–746, 2010. 19

[25] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *SIGMETRICS*, pages 318–319, 2003. 23

[26] PREDATOR. Design for predictability and efficiency. http://www.predator-project.eu/. 16, 25

[27] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2/3):115–128, 2000. 4

[28] P. Radojkovic, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *TACO*, 8(4):34, 2012. 21, 25, 46, 47, 53, 130

[29] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: bank privatization for predictability and temporal isolation. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '11, pages 99–108, New York, NY, USA, 2011. ACM. 18

[30] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for tdma arbitration in resource sharing systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224, 2010. 19

[31] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *IEEE PACT*, pages 3–14, 2001. 23

[32] B. Sprunt. The basics of performance-monitoring hardware. *Micro, IEEE*, 22(4):64–71, 2002. 24

[33] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. 16, 25

[34] T. Ungerer, E. Quinones, H. Ozaktas, I. Broster, J. Fernandes, and S. Kehr. parMerasa: Multi-core execution of parallelised hard real-time applications supporting analysability. *Workshop on Advanced Real-time Architectures (ARPA)*, 2012. 16, 25

[35] Valgrind. Valgrind Home Page. `http://valgrind.org/`. [Online]. 23

[36] P. Viola and M. Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001. 52

[37] M. Wicks, M. Rangaswamy, R. Adve, and T. Hale. Space-time adaptive processing: a knowledge-based perspective for airborne radar. *Signal Processing Magazine, IEEE*, 23(1):51–65, 2006. 52

[38] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, T. Mitra, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, I. Puaut, R. Heckmann, F. Mueller, P. Puschner, J. Staschulat, and P. Stenström. The worst case execution time problem, overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, pages 36–53, May 2008. 4, 8, 10, 11, 13, 14, 15

[39] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009. 19

[40] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *ISPASS*, pages 76–86, 2010. 20

[41] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In

Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, PACT '07, page 339, 2007. 20