NEXT-GENERATION SAFETY-CRITICAL SYSTEMS
ON MULTI-CORE COTS PLATFORMS

BY

RENATO MANCUSO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Marco Caccamo, Chair
Professor Lui R. Sha
Professor Tarek F. Abdelzaher
Professor Björn B. Brandenburg, Max Planck Institute for Software Systems
ABSTRACT

Multi-core platforms represent the answer of the industry to the increasing demand for computational capabilities. In fact, multi-core platforms can deliver large computational power together with minimum costs, compact size, weight and power usage. Multi-core architectures however are shaking the very foundation of modern real-time computing theory, i.e. the assumption that worst case execution time (WCET) can be calculated on individual tasks to compute the schedulability of the complete system when tasks are running together. This fundamental assumption has been broadly accepted by classic scheduling theory for the past three decades; unfortunately, it is not even true in an approximate sense in a modern multi-core chip, and this leads to a lack of composability. Shared hardware resources like caches, main memory, and I/O buses are all sources of unpredictable timing behavior and temporal dependencies among real-time tasks running in parallel. As a result, certifying systems deployed using multi-core platforms is significantly more challenging compared to single-core implementations.

In this work, we tackle the challenge of restoring the constant WCET assumption for real-time tasks deployed on multi-core systems. While predictability and performance determinism are of paramount importance in safety-critical applications, cost containment and time-to-market are dominant factors for the large-scale adoption of novel technologies. Hence, our work proposes solutions that can be adopted with commercially available components, also known as commercial-off-the-shelf (COTS) components. In order to achieve deterministic performance on COTS multi-core platforms, we propose software-level techniques to enforce usage control over shared hardware resources. We also demonstrate that when proper enforcement is performed, real-time analysis can be carried out efficiently. We focus our attention on two main multi-core architectural paradigms: cache-based and scratchpad-based platforms.

On multi-core cache-based architectures, we design, implement and analyze a set of OS-level techniques that enforce hardware resource partitioning. In this context, we set two main goals. Our first objective is to achieve strong inter-core performance isolation in spite of inherent hardware resource sharing. On the other hand, our techniques are designed to remain transparent from an application perspective. This requirement allows for minimum re-engineering being required to port legacy single-core systems on multi-core platforms partitioned with the proposed techniques.

On scratchpad-based platforms, we follow a different approach. In fact, we propose a redesign of the OS-level scheduling strategies. The goal is to include scratchpad space scheduling, as well as shared memory bus access, together with traditional processor time scheduling. The resulting resource co-scheduling strategy introduces a set of new challenges compared to processor-only scheduling. Nonetheless, it allows to significantly mitigate the problem of inter-core performance interference, as we describe in our evaluations.
ACKNOWLEDGMENTS

It is my absolute pleasure to thank all the people that made this dissertation possible, and who also made my Ph.D. a spectacular experience. First and foremost, my gratitude goes to my advisor, Prof. Marco Caccamo. It is hard to describe the uncountable ways in which Marco impacted my life and career. My encounter with him absolutely changed the course of my history in ways that I could never imagine about 5 years ago. He shared with me his expertise and always provided constructive guidance. He involved me in exciting projects and valued my suggestions. He often believed in my capabilities more than I would have believed myself. All in all he has been an irreplaceable advisor and continues to be a precious friend.

I am extremely grateful to the members of my thesis committee: Prof. Lui Sha, Dr. Björn Brandenburg, and Prof. Tarek Abdelzaher. Each of them in a peculiar way helped me in my endeavor, supported my work, and significantly contributed to make it valuable. Prof. Lui Sha has covered a key role in the development of my career as a young researcher since the very first day I joined UIUC. His natural ability to identify intellectually stimulating yet practical challenges and to envision ambitious solutions fundamentally steered the path of my research. I am very thankful to Björn. Not only his continuous and detailed feedback has dramatically improved this work, but he has also offered me invaluable support for opening the next chapter of my academic career. I learned unforeseen aspects about my own research from Tarek in a number of different contexts: as a student in his class; as a collaborator in research projects; and during many lively conversations around the department. Apart from his role in focusing my thesis, I am grateful to Tarek for sharing with me his unique vision on cyber-physical systems.

What I address as “my work” has always been an elaboration, refinement and crystallization of countless conversations, interactions and exchanges with a number of collaborators. In rough chronological order, Dr. Emiliano Betti was certainly my first collaborator. I owe to him my first steps as a Ph.D. student and my first contacts with UIUC. Alongside, I am thankful to my Master’s thesis advisor and collaborator, Prof. Marco Cesati, for his contagious passion for operating systems. I am also glad that I had the opportunity to work with Prof. Rodolfo Pellizzoni, whom I truly admire for his technical depth and scientific rigorousness. I am grateful to Prof. Heechul Yun: his technical insights always represented a positive contribution the quality of this work.

I am very thankful to the people who shared with me days and nights, weekdays and weekends in UIUC. These are my friends, my collaborators and my colleagues. A special thank goes to Fardin Abdi, not only for
our endless conversations but also for being my “Ph.D. brother” since my very first day in UIUC. Thanks also to my friend and business partner Or Dantsker. Similarly, a thank goes to Rohan Tabish, Prakalp Srivastava, and Alessandra Melani who worked with me on a number of projects. At the same time, I would like to thank those friends who, despite being geographically far, always rooted for me: Emiliano Masi and Emanuele Ghedin.

It is hard to properly thank my life partner Neriman Tokcan. Not only she has my undivided admiration for her beautiful mathematical mind, but she also shared with me a significant portion of my Ph.D. adventure. Without her, I would have never considered Urbana-Champaign my *home in USA*.

Last but not least, I wish to thank my family for their never ending support. First, I am immensely thankful to my mother, Teresa, who always encouraged me to find my own path. I owe to her the serenity of mind to focus on my studies in spite of ongoing family issues. I am grateful to my sister, Iolanda, who has always been on my side and who had to cover for my absence way too many times. A thank goes also to my brother, Marco, who has always inspired me to see beyond formulas and algorithms to admire the beauty in human history, literature and poetry.
# TABLE OF CONTENTS

List of Tables .................................................................................................................. vii

List of Figures ................................................................................................................... viii

CHAPTER 1 Introduction ................................................................................................. 1
  1.1 Challenges with Multi-Core Systems ........................................................................ 1
  1.2 Scope and Contributions of This Work ...................................................................... 6
  1.3 Intellectual Merit and Impact .................................................................................... 10
  1.4 Structure .................................................................................................................. 12

CHAPTER 2 Background and Related Work .................................................................... 13
  2.1 Multi-Core Architectures .......................................................................................... 13
  2.2 Cache Organization and Management ...................................................................... 18
  2.3 DRAM Organization and Management .................................................................... 24
  2.4 Scratchpad Organization and Management .............................................................. 27
  2.5 Regulations for the Adoption of Multi-Core Platforms ............................................ 29
  2.6 Related Work on Real-time Multi-Core Resource Management ............................ 31
  2.7 Related Work on Workload Profiling ....................................................................... 33

CHAPTER 3 Predictability on Cache-based Architectures .............................................. 35
  3.1 System Model and Assumptions .............................................................................. 35
  3.2 Colored Lockdown - Cache Management .............................................................. 40
  3.3 PALLOC - DRAM Bank Partitioning ....................................................................... 48
  3.4 MemGuard - Memory Bandwidth Partitioning ......................................................... 64
  3.5 Single-Core Equivalence ......................................................................................... 65
  3.6 Verification of Colored Lockdown ............................................................................ 80
  3.7 Summary ................................................................................................................ 92
  3.8 Future Work .......................................................................................................... 94

CHAPTER 4 Predictability on Scratchpad-based Architectures ..................................... 96
  4.1 System Model and Assumptions .............................................................................. 96
  4.2 Scratchpad-Centric OS .......................................................................................... 100
  4.3 Fault-Tolerant Scratchpad-Centric Scheduling ....................................................... 115
  4.4 Multi-stage Scheduling and Optimization ............................................................... 127
  4.5 Summary .............................................................................................................. 128
  4.6 Future Work .......................................................................................................... 129

CHAPTER 5 Workload Profiling .................................................................................... 131
  5.1 Extracting Application Behavior ............................................................................. 131
  5.2 Profiling to Drive Cache Allocation ........................................................................ 148
  5.3 Profiling to Reorganize Application Behavior ........................................................ 153
  5.4 Summary .............................................................................................................. 171
  5.5 Future Work .......................................................................................................... 172
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Lockdown features on multi-core embedded systems</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Summary of parameters for resource partitioning techniques in cache-based platforms</td>
<td>40</td>
</tr>
<tr>
<td>3.3</td>
<td>Page allocation overhead</td>
<td>54</td>
</tr>
<tr>
<td>3.4</td>
<td>HRT task runtime statistics on Intel Xeon</td>
<td>58</td>
</tr>
<tr>
<td>3.5</td>
<td>HRT task runtime statistics on Freescale P4080</td>
<td>58</td>
</tr>
<tr>
<td>3.6</td>
<td>SPEC2006 characteristics on Intel Xeon</td>
<td>59</td>
</tr>
<tr>
<td>3.7</td>
<td>Characterization of SD-VBS benchmarks</td>
<td>73</td>
</tr>
<tr>
<td>3.8</td>
<td>Runtime and $C_{sec}$ estimation with fixed cache allocation</td>
<td>76</td>
</tr>
<tr>
<td>4.1</td>
<td>Suitable Commercial Multicore COTS platforms</td>
<td>99</td>
</tr>
<tr>
<td>4.2</td>
<td>Summary of parameters for resource co-scheduling techniques in scratchpad-based platforms</td>
<td>99</td>
</tr>
<tr>
<td>4.3</td>
<td>Characteristics of Freescale MPC5777M SoC</td>
<td>109</td>
</tr>
<tr>
<td>4.4</td>
<td>Details of OS Parameters</td>
<td>111</td>
</tr>
<tr>
<td>4.5</td>
<td>Details of EEMBC Benchmarks</td>
<td>114</td>
</tr>
<tr>
<td>4.6</td>
<td>Details of OS Parameters</td>
<td>125</td>
</tr>
<tr>
<td>4.7</td>
<td>Details of EEMBC Benchmarks</td>
<td>126</td>
</tr>
<tr>
<td>5.1</td>
<td>Types of memory events in the output file</td>
<td>140</td>
</tr>
<tr>
<td>5.2</td>
<td>Symbol name format for dynamic memory events</td>
<td>141</td>
</tr>
<tr>
<td>5.3</td>
<td>Summary of memory events for disparity benchmark</td>
<td>141</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of benchmarks normalized runtimes</td>
<td>147</td>
</tr>
<tr>
<td>5.5</td>
<td>Memory profiling results</td>
<td>153</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

1.1 Slowdown induced on applications due to inter-core performance interference. Synthetic benchmark results by Thales Ground Transportation Systems. Results presented during keynote speech “Mixed-Criticality Systems - a Journey Embedded in Time and Space” (M. Paulitsch) at 27th Euromicro Conference on Real-Time Systems (ECRTS’15), 2015 .......................... 2

1.2 Synthetic benchmark results obtained by Lockheed Martin in collaboration with us. The results were also presented in [1]. .................................................................................. 3

2.1 Main components of a modern multi-core cache-based architecture. ........................................... 15

2.2 Main components of targeted multi-core scratchpad-based architectures. ................................. 17

2.3 An example of a 2-way set-associative cache (a). A main memory block can be placed in one of the two cache locations; and the structure of an address in a W-way set-associative cache (b). ........................................................................................................... 18

2.4 Classification of cache partitioning approaches: (a) overview of the index-based partitioning. (b) overview of the way-based partitioning. ........................................................................................................... 21

2.5 Examples of cache locking variations: (a) overview of the way locking. (b) overview of the cache line locking. ........................................................................................................... 21

2.6 Internal organization of: (a) DRAM controller; and (b) DRAM memory [2]. ................................. 24

2.7 Best-case (a) and worst-case (b) scenario in the pattern of access of multiple cores to DRAM banks. ........................................................................................................... 26

3.1 Comparison between the legacy (a) and the proposed (b) real-time task model on a multi-core architecture. ........................................................................................................... 42

3.2 Physical address structure as seen by the cache controller (a) and the OS (b). ................................. 43

3.3 Graph of observed worst case execution time for each benchmark, with 1) no interference and no protection, 2) interference and no protection, and 3) interference and protection. .................................................................................. 47

3.4 Graph of execution time depending on the number of pages locked, for the benchmark a2time. .... 48

3.5 Graph of execution time depending on the number of pages locked, for the benchmark canrdr. .... 49

3.6 Resulting observed average time for four benchmarks scheduled at the same time with different priorities. The highest priority level (a2time) is the top priority. ........................................... 50

3.7 PALLOC implementation .................................................................................................................. 51

3.8 Memory address mappings ................................................................................................................. 52

3.9 Samebank vs. Diffbank on Intel Xeon (a); and Samebank vs. Diffbank(s) on Freescale P4080 (b). ........................................................................................................... 55

3.10 HRT runtime distribution on Intel Xeon. Core0-HRT, Core1-Xserver(none for (a)). ..................... 57

3.11 Performance impact of private DRAM banks on uncore .................................................................. 60

3.12 Performance impact of private DRAM banks (PB) and private cache (PC) on uncore. ............... 61

3.13 Slowdown ratios of SPEC2006. Corei=1-axis, Core1-3=470.lbm×3. ........................................... 62

3.14 Normalized IPC of SPEC2006. Core0: X-axis, Core1-3: 470.lbm×3. ........................................... 63

3.15 Progressive lockdown curve for tracking benchmark. ....................................................................... 74

3.16 Csce calculation and experimental MemGuard runtime for tracking. .......................................... 75

3.17 Sensitivity of Csce to variations in Lmin (i.e BWmax) parameter. ..................................................... 77
3.18 Progressive Lockdown Curve for Task $\tau_1$ and resulting WCET(1) for different amount of memory pages allocated in last-level cache. ......................................................... 77
3.19 Impact of DRAM controller bandwidth regulation via MemGuard on WCET($m$). ........ 78
3.20 Scheduling of 2 IMA partitions with 2 tasks per partition on a single processor (CPU 1). 79
3.21 Scheduling of 4 IMA partitions on 2 processors. Arrival and departure of data from I/O devices is depicted on the bottom row ("Physical I/O"). Serialization of I/O transactions on dedicated I/O Core is depicted on the third row ("I/O Core$^3$"). .......................................................... 80
3.22 Verification runtime for scenario: $P_s = 8, B_w = 32, O = 6, I = 3$ and associativity $W \in [1,7]$. 91
3.23 Verification runtime for scenario with $P_s = 12, B_w = 32, O = 6, I = 10$ and $W = 1$. . . 92

4.1 Multicore architecture satisfying our hardware assumptions. ........................................ 98
4.2 Scheduling CPU, DMA and local memory. ................................................................. 102
4.3 Interaction between I/O Core and Core 1 for task scheduling. ................................. 103
4.4 SPM-centric OS task scheduling. Scheduling intervals are highlighted. ................. 105
4.5 Experimental execution time for synthetic benchmarks. .......................................... 112
4.6 Experimental execution time for EEMBC benchmarks. ........................................ 113
4.7 Schedulability with SPM-based and traditional scheduling models. ......................... 115
4.8 The scheduling example with the proposed error recovery mechanism. ...................... 117
4.9 SPM-centric OS task scheduling. Scheduling intervals are highlighted. .................... 119
4.10 Block diagram of error handling circuitry. ............................................................... 124
4.11 Schedulability degradation with fault recovery mechanism. ................................... 127

5.1 Graphic view of disparity benchmark memory profile . ........................................... 142
5.2 Comparison on memory trace extraction time. ......................................................... 146
5.3 Profile creation for the benchmark “a2time01” and relations between the files involved, starting from the ranked list of accessed memory pages (a) to the final profile file (f). ........ 150
5.4 Predictable Interval with constant execution time ................................................. 155
5.5 Memory layout in a sample scenario . ...................................................................... 159
5.6 Memory layout in a sample C application .............................................................. 163
5.7 Heap Memory Graph with a self-loop ..................................................................... 165
5.8 Sample organization of virtual memory based on graph from Figure 5.7 ................. 165
5.9 Coverage measurements: ratio between misses captured by Light-PREM and manual PREM with respect to the misses observed in the non-PREM case (first two bars); Ratio between prefetches issued by Light-PREM and prefetches issued by manual PREM (third bar). Both EEMBC (a) and MiBench (b) are reported. ........................................................... 167
5.10 Exec. phase runtime and variance for Light-PREM and non-PREM ......................... 170
5.11 Trend of time required to perform the Light-PREM analysis (in seconds) compared to the execution time of the benchmark (in milliseconds). Benchmarks ordered by runtime. .......................................................... 171
CHAPTER 1
INTRODUCTION

Since its very inception, real-time computing theory and the embedded systems community in general have attained important results on the design of hardware and software components for safety-critical applications. Major milestones in fields that range from scheduling theory to hardware/software co-design have significantly impacted the way the academic community and industries approach the analysis, development, verification and validation of safety-critical systems. In the last decade, however, there has been a steady uptrend in the popularity of embedded multi-core platforms. This represents a turning point in the theory and implementation of real-time systems.

Safety-critical systems are often referred as hard real-time systems. In this context, the additional hard attribute conveys the idea that being able to deterministically meet strict timing constraints on applications' execution time is fundamental for the correctness of the system. Consider an airbag system in a modern car, which is an explanatory example of a highly time-sensitive safety-critical system. When a crash occurs, the driver is projected forward – within the abruptly decelerated car body – due to conservation of momentum. An electronic Airbag Control Unit (ACU) monitors a wide range of sensors (e.g. accelerometers, impact sensors, pressure sensors) to determine the occurrence of a crash. From the moment of the impact, the ACU has between 20 and 30 milliseconds to initiate an airbag deployment, that typically completes within 80 milliseconds. The goal is to gradually decelerate the forward-projected driver. If the airbag deployment is initiated too slowly, the forward-projected driver is impacted by a deploying airbag instead. Safety-critical systems, like ACUs, need to produce their outputs before the expiration of strict time deadlines to prevent catastrophic failures. In fact, the majority of sense-and-control systems can be formulated in terms of deadline-constrained periodic applications, also known as tasks.

1.1 Challenges with Multi-Core Systems

One of the biggest challenges the real-time community faces today is trying to analyze the temporal behavior of real-time applications running in parallel on multi-core computing systems due to their extensive sharing of hardware resources. To understand why this problem arises, consider once again an ACU. In federated systems like traditional airbag implementations, the ACU is implemented using an ad-hoc microcontroller. Hence, the ACU can be studied in isolation from the rest of the system to verify that it can meet its
timing requirements. Unfortunately, federated systems do not scale well with system complexity. Recently, the majority of industries tend to integrate and implement logically unrelated subsystems within the same microcontroller. When many subsystems are integrated onto the same single-core chip, partitioning the processor time to accommodate a set of different timing constraints represents the main concern. Once an application, say task A, runs on the CPU in a single-core system, all the hardware resources are exclusively used by A. Conversely, on multi-core chips, two applications, say A and B, may be running simultaneously on two different cores. As they run, they use a wide range of shared hardware resources, such as caches, main memory, and I/O buses. If task A and B compete to access shared resources, the behavior of A impacts the execution time of B, and vice versa.

The current trend in the industry of embedded systems is to extensively use Commercial-Off-The-Shelf (COTS) multi-core platforms to build hard real-time systems (e.g., infrastructures, avionics, medical aerospace, and automotive systems to name a few). Unfortunately, such platforms are not designed to support real-time applications, but rather to optimize performance in average. Due to extensive sharing of hardware resources, multi-core architectures are shaking the very foundation of modern real-time computing theory, i.e. the assumption that worst-case execution time (WCET) can be calculated on individual tasks to compute the schedulability of the complete system when tasks are running together. This fundamental assumption has been broadly accepted by classic scheduling theory for the past three decades; unfortunately, it is not even true in an approximate sense in a modern multi-core chip.

Figure 1.1: Slowdown induced on applications due to inter-core performance interference. Synthetic benchmark results by Thales Ground Transportation Systems. Results presented during keynote speech “Mixed-Criticality Systems - a Journey Embedded in Time and Space” (M. Paulitsch) at 27th Euromicro Conference on Real-Time Systems (ECRTS’15), 2015

(a) WCET Trend in Multi-Core

(b) Interference overhead

Figure 1.1 reports the results for two experiments performed by Thales Austria GmbH. In both the Fig-

1 See presented slides at [http://control.lth.se/ecrts2015/files/keynote.pdf](http://control.lth.se/ecrts2015/files/keynote.pdf)

2 Part of the Thales Ground Transportation Systems.
Figure 1.2: Synthetic benchmark results obtained by Lockheed Martin in collaboration with us. The results were also presented in [1].

A set of synthetic benchmarks has been used to observe the performance degradation arising from inter-core performance interference. The setup can be summarized as follows: the observed task is activated on a given core, say Core 0, while all the other cores in the system are idle or disabled. Next, the execution time for the task under analysis is observed as an increasing number of co-running applications are activated on the other cores. The focus of Figure 1.1(a) is on the time to perform single main memory transactions, under four different scenarios: when the observed task is performing read operations and the interfering tasks are also performing read operations (“read-read”); when the observed task is performing read operations while the interfering tasks are performing write operations (“read-write”); and so on. It can be noted that the time scale on the y-axis is logarithmic, meaning that the worst-case memory access time increases over-proportionally with the number of active cores. The effect on the global execution of the entire task is depicted in Figure 1.1(b). As can be seen, the concurrent activity of 4 cores determines a 3x slowdown compared to the single-core case. The loss in performance can be considered as an interference-induced overhead. Note also that, compared to Figure 1.1(a), the impact of inter-core interference on the overall task execution is lower: 3x slowdown vs. the 10x slowdown in Figure 1.1(a) for 4 active cores. This is because, during typical task execution, a portion of the time is spent for computation-only operations that can progress in parallel and that often hide the extra time required for non-blocking memory operations.

The results of an experiment performed in collaboration with Lockheed Martin are presented in Figure 1.2. The experiment was performed on a Freescale/NXP P4080 [4] 8-core platform using synthetic tasks. Just like the setup used in Figure 1.1, we observe the execution time of an application running on Core 0. As can be seen, the interference between cores can cause delay spikes as high as 600% as shown in Figure 1.2. The first bar in each cluster shows the WCET of a single task running on Core 0 when increasing number of tasks are run simultaneously without any resource management scheme to address inter-core interference. As Figure 1.2 shows, the Core 0 task’s WCET as a function of an increasing number of competing tasks running on Cores 1 through 7, reaches a peak of a 6x higher WCET, when 7, not 8 cores are used. This
means that the worst case configuration of a multicore chip is nondeterministic, creating great challenges to system integration and certification of hard real time systems such as avionics. The second bar in each cluster depicts the same measurement on the observed task’s execution time when some of our techniques (see Chapters 3–5) to mitigate inter-core interference are deployed.

**From an academic perspective**, two main pieces of information are required to compute the worst-case execution time of real-time tasks on COTS multi-core platforms: (i) architecture-specific micro-architectural models; and (ii) for each task A under analysis, a detailed characterization of all the tasks that could run in parallel with A in the rest of the system. When COTS systems are used, micro-architectural models are often unavailable to designers and certification authorities as they constitute chip manufacturers’ intellectual property. Hence, theoretical frameworks that rely on this assumption currently find limited applicability in industry. On the other hand, assuming accurate knowledge of the entire system workload also faces practical limitations. In fact, well-studied safety-critical components (e.g. the ACU) may operate in parallel with non-critical software modules whose interaction with the hardware is largely unknown.

**From an industrial perspective**, an additional dimension needs to be considered. On single-core platforms, the design of safety-critical systems has historically relied on extensive platform evaluation and performance assessment to compute safe WCET bounds for real-time tasks. The set of engineering practices and processes have been optimized, field-tested and standardized by vendors and certification authorities. Alongside, a substantial base of certified single-core software has been produced and deployed for commercial applications. As mentioned earlier, the analysis carried out for single-core systems was largely simplified by the presence of a unique workload execution flow. This property meant that if overheads arising from workload switching were properly accounted, applications could essentially be analyzed in isolation. More importantly, the composition of individually analyzed applications could be used to reason about the integrated system. The latter property goes under the name of *composability*. In multi-core systems, however, the presence of many parallel execution flows operating with a loose time synchronization essentially undermines the ability to achieve composability. Not only a task in isolation has a substantially different timing behavior when executed in parallel with other applications. In addition, as the number of parallel execution units (cores) increases, the problem of exhaustively testing all the possible platform/workload configurations combinatorially explodes.

Nowadays, certification authorities, such as FAA, officially express their concerns about the adoption of multi-core platforms in safety-critical systems due to a fundamental lack of analysis methodology [5]. At the same time, hardware manufacturers progressively discontinue the production of single-core chips to keep up with the increasing demand for performance in the general purpose computing market. Hence, platform selection for the design of modern safety-critical systems often involves outdated technological solutions or highly inefficient exploitation of current hardware.

Yet, avionics, automotive, medical and aerospace systems, to name a few, become increasingly complex from generation to generation and often incorporate real-time image processing, machine learning and ad-
vanced human-machine interfaces. Multi-core platforms would represent the natural answer for the increase of computation demand in safety-critical applications. In particular, COTS systems are attractive to industries since they deliver the desired performance, are cost effective and can reduce time-to-market compared to custom solutions. What currently keeps players in the real-time systems market away from multi-core solution, thus, is not the raw cost of the chips, but rather the much higher collateral costs of software re-engineering and certification. Unfortunately, a trade-off needs to be established between mitigating inter-core performance interference and offering perfect backward compatibility of software components designed for single-core platforms.

In order to adopt a *divide and conquer* approach in certifying safety-critical systems, achieving isolation is key. Isolation in time and space can be described as follows:

1. **Spatial Isolation:** modern platforms are comprised of a number of different resource types. Application cores, memory subsystems (e.g., flash, DRAM, on-chip memories and caches), special-purpose co-processors - such as GPUs and DSPs. It is crucial for modern platforms to confine the access of software components to a restricted set of hardware resources. This requirement is currently well understood as it is mostly inherited from general-purpose computing and has represented a key requirement to perform multi-tasking since the early nineties. Modern architectures in fact have extended memory protection capabilities in two main important directions. First, horizontally: modern Memory Management/Protection Units (MMUs/MPUs) are not only able to restrict access to portions of the physical memory, but also to different components of the I/O subsystem. This has determined the transition from traditional MMUs to so called IOMMUs implemented in Intel³ AMD⁴ and ARM⁵ processors. Second, vertically, with the introduction of an additional, high-privilege level often referred as hypervisor-level. Spatial isolation ensures that erroneous results produced by some application A do not affect the computation of a second, logically unrelated application B.

2. **Temporal Isolation:** by temporal isolation, the timing required by application A to complete its execution is unaffected by the activity of a logically unrelated application B on a different core. In other words, the performance of application A is independent from the behavior of application B. For this reason, we will also refer to temporal isolation with the term *performance isolation*. Hardware resource sharing in multi-core platforms is achieved following a best-effort scheme to maximize throughput. It follows that modern architectures do not provide extensive support to achieve strong temporal isolation among applications. At the same time, typical OS design addresses only a subset of the formidable challenges that need to be tackled to achieve strong temporal isolation. In this sense, there exists a gap between (i) what is required to efficiently design, develop, verify and validate safety-critical systems

---


1.2 Scope and Contributions of This Work

This work represents an attempt to enable the deployment of complex, next-generation safety-critical systems using modern multi-core platforms. In agreement with the industrial tendency to use COTS platforms, we explore OS-level and application-level solutions that require no hardware modifications and aim at improving temporal isolation in hard-real time systems. At the same time, we discuss aspects of fault-tolerance and verification to streamline certification of safety-critical systems. In this work, we consider techniques that apply to two categories of COTS platforms: cache-based and scratchpad-based architectures. In both cases, we focus on the enforcement of OS-level techniques that achieve a good level of temporal isolation on commercially available hardware components. The underlying philosophy is also alike: in both cases, the hardware behavior is restricted so to export deterministic properties at a scheduling and analysis level. Nonetheless, the different approaches taken on cache- and scratchpad-based platforms correspond to two different trade-offs between backward compatibility and efficiency of performance isolation methodologies.

1.2.1 Approach for Cache-based Platforms

Cache-based platforms represent an adaptation of general-purpose computing architectures to the embedded domain. Legacy single-core applications can be easily ported with few or no modifications to operate on this class of multi-core platforms. Hence, industries that intend to migrate to multi-core solutions and yet have a large body of certified single-core software are more incline to adopt cache-based SoCs. For this reason, backward compatibility and application-level transparency represent driving factors in the design of OS-level mechanisms to achieve temporal isolation on cache-based systems. The key idea is to prevent shared resources to be utilized beyond their saturation point by actively managing their usage. The employed partitioning strategies need to remain completely transparent to user-space applications. The goal is to execute tasks in parallel as if each application was executed on dedicated hardware resources so that no re-engineering is required to port existing single-core software components. As discussed in Chapter 3, this approach is feasible using COTS components and leads to strong performance isolation. Nonetheless, without proper optimization, the resource partitioning mechanisms can introduce large performance overhead.

In order to attain this goal, we focused on the memory hierarchy as it represents the primary source of inter-core performance interference. The memory hierarchy can be decomposed at a high-level in three layers: (i) cache; (ii) DRAM controller; (iii) DRAM memory. For each of these layers we have developed a regulation solution to mitigate interference. We have finally integrated these solutions to create a unified framework, namely Single-Core Equivalence (SCE), to enforce real-time performance isolation among CPUs.
in a multi-core system. As a result, SCE is able to provide both performance benefits, since multi-core platforms can be fully exploited, and a reduction of certification costs.

In the context of cache-based architectures, we make the following contributions:

**Colored Lockdown:** An OS-level technique to deterministically manage shared last-level cache in COTS multi-core platforms. The proposed cache management scheme was implemented and tested on ARM Cortex-A9 and NXP/Freecale QorIQ SoCs. Our approach uses a combination of cache coloring and lockdown and remains transparent to user-space applications. The scheme allows to utilize the last-level cache space entirely at the granularity of single memory pages. Experimental results show that, in the considered benchmarks, eliminating the interference in the last-level cache can lead up to a 250% improvement of the execution time, with a resulting cache occupation as small as 4 memory pages. This work was published in [6].

**Palloc:** A DRAM bank-aware memory allocator that can allocate memory to specific DRAM banks by leveraging the page-based virtual memory system of modern operating systems. By allocating sets of disjoint DRAM banks to different cores, we prevent inter-core performance interference at the level of DRAM banks. This scheme goes under the name of *private banking*. We perform an experimental study that investigates the performance impact of private banking in two COTS multicore platforms. Most notably, we find that the private banking strategy reduces performance variations (up to 4.4x), and offers better real-time performance on COTS multicore platforms. Palloc was originally presented in [7].

**Single Core Equivalence:** A complete integration of last-level cache, DRAM bank and DRAM controller bandwidth management for hard real-time applications on a COTS multi-core platform. We have developed and integrated the discussed regulation techniques in a unified OS-level framework that has the following main properties: (i) allows real-time tasks to run according to a traditional execution model; (ii) actively and transparently regulates at an OS-level the usage of shared hardware resources; and (iii) relies on existing hardware support available in COTS multi-core platforms. We derive analytic results to derive the WCET of tasks running with the proposed performance-isolation techniques. Our framework has two major benefits: first, if less than the total number of cores are used in production, the remaining ones can be activated at a later upgrade without system-wide re-certification; and second, any of the certified cores can be treated as an independent processor in a conventional single-core chip, meaning that local workload changes require local re-validation only. SCE was published in [8, 9].

**Verification of Colored Lockdown:** A proof-of-concept verification of one of our resource management modules. Often, resource management techniques proposed in the literature have been shown to achieve substantial real-time benefits. Nonetheless, these solutions are not widely adopted in production, partly
because there is a fundamental lack of confidence about their implementation. Indeed, industries prefer to rely on consolidated technologies, despite paying relatively high performance penalties, rather than including experimental logic in safety-critical systems. The fear is justified considering that hardware management mechanisms often operate at high-privilege level, and thus their misbehavior can lead to substantial failures. We demonstrate that it is possible to verify the logic of an OS-level component at the source code level in a modular way; i.e. without verifying the entire OS that can be assumed verified or trusted [10]. Specifically, we present a verification approach for our a real-time last-level cache management scheme implemented in the Linux kernel.

1.2.2 Approach for Scratchpad-based Platforms

The approach followed on scratchpad-based architectures does not aim at achieving backward compatibility with legacy single-core software components. Instead, it focuses on re-structuring the execution model of tasks and OS components so to organize the access to shared hardware resources. The presence of well-structured (e.g. clustered) requests to shared resources can be exploited at an OS level to perform co-scheduling of CPU, memory and I/O resources. Often, specialization of COTS hardware resources is exploited to better handle different stages of task execution. The main advantage of this approach lies in its efficiency, as comparatively less overhead is paid to control access to shared resources. However, the lack of backward compatibility makes the latter approach more suitable for the design of new systems performed from scratch.

Scratchpad memories, in fact, have been proven to provide better temporal isolation when compared to traditional caches [11][12]. Alongside, we exploit additional hardware features that vendors are now including in some modern families of multi-core platforms designed for the embedded market, such as: separate I/O and memory buses, the presence of dual-port memories with DMA support, and core specialization. Thereby, this work makes the following contributions:

Scratchpad-centric OS: An OS design to achieve temporal predictability exploiting task execution from scratchpad memories. Our OS design targets multi-core embedded COTS platforms, exploits core specialization, and enforces predictable resource management policies on shared hardware resources. Furthermore, we integrate a scratchpad-based CPU scheduling mechanism with an application-schedule-aware I/O subsystem. We perform a full implementation of the proposed OS using a commercially available multi-core micro-controller. Its design has been validated using a combination of synthetic and EEMBC benchmarks. Our evaluations reveal that strictly deterministic timing can be achieved for hard real-time applications with performance improvements up to 3.5x compared to legacy approaches. This work was published in [13].

Scratchpad-centric Analysis: An analysis is derived to calculate the response time of real-time tasks under the proposed scheduling strategy [13]. The analysis considers a task model composed of three
phases: load (onto scratchpad); execution on local CPU; and unload (from scratchpad). The analysis takes into account not only task execution, but also the time required to setup the task context onto the scratchpad memory, as well as the system overheads.

Fault-tolerant Scratchpad Scheduling: A set of OS-initiated strategies that extend our scratchpad management layer to recover from detectable transient memory errors. Specifically, we describe how it is possible to recover from bit errors, in both main memory and scratchpad, that are detected but not corrected by the hardware logic via error-correcting code (ECC). The strategy that we follow largely leverages the existing redundancy in the employed multi-phase task model. A minimum amount of additional redundancy is introduced to protect data that do not exist as multiple copies in the original scheme. Once again, we validate and evaluate our error recovery strategies using a combination of synthetic and realistic (EEMBC) benchmarks. Schedulability analysis was extended to take into account the overhead introduced by the proposed recovery mechanisms. The proposed analysis consider different application recovery scenarios and isolates the critical path of the error-handling procedures, which often correspond to entire task re-execution. The result is a fault-tolerant scheduling framework for multi-phase real-time tasks, and it was presented in [14].

1.2.3 Approach for Application Profiling
Modern multi-core architectures are characterized by heavily parallelized computational resources and a mostly centralized memory hierarchy. It follows that being able to efficiently manage the components of the memory hierarchy is a crucial aspect to address toward practical resource regulation and control. In order to attain this goal, two are the main requirements: (i) a good understanding about the functional and timing behavior of the underlying hardware components; and (ii) some degree of knowledge about the behavior of the tasks under analysis. The former can be acquired via the hardware manufacturer in the form of documentation or through platform benchmarking and represents a required per-platform knowledge base. Conversely, the latter is strictly dependent on the characteristics of the application workload and thus needs to be acquired on a per-task basis.

We briefly review aspects and methodologies for application workload profiling, i.e. the process of extracting behavioral knowledge out of existing applications. The problem of determining the memory access pattern of a task under analysis for workload characterization purposes has been approached from several perspectives, resulting in solutions with different advantages and trade-offs, as we briefly discuss is Section 2.7. In our work, we focus on profiling performed through the observation of live execution of the target program. In this context, we make the following contributions:

Memory Access Detection: A technique to perform memory access detection that relies on memory protection mechanisms. The methodology and proposed implementation is aimed at demonstrating
how it is possible to acquire accurate memory traces by introducing controlled memory faults and trap exceptions. In this way, we (i) do not rely on advanced debugging hardware capabilities, (ii) do not need to instrument any of the instructions in the observed binary, (iii) do not necessarily need to instrument the source code. Compared to other memory tracing approaches, the proposed technique shows similar or better performance. The main advantage of this approach are the following: (i) it is minimally intrusive on the memory layout of the program under analysis; (ii) its implementation is lightweight with a small portion of the code being platform-specific. The memory access detection technique as well as some details about our proof-of-concept implementation were presented in [15].

**Cache Profiling:** A technique to use information from memory traces to build a task profile to drive cache allocation using Colored Lockdown. Specifically, as the task runs in the profiling environment and memory accesses are traced, per-page access statistics are maintained. Next, (i) pages of the task’s addressing space are ranked by access frequency; and (ii) a profile is produced identifying frequently accessed (hot) pages by their relative position in the addressing space. Thanks to its relative positioning mechanism, derived memory profiles are valid from run to run despite the presence of memory address layout randomization. This work was presented in [6].

**Memory Access Pattern Re-Structuring:** A technique to use information extracted via memory tracing to re-structure the memory access pattern of tasks. As previously mentioned, when access to memory resources is clustered, it is possible to co-schedule CPU and memory to achieve efficient performance isolation. This work bridges the gap between resource management techniques in the context of cache- and scratchpad-based platforms. In fact, we demonstrate that it is possible to automatically re-structure legacy application-level code to exhibit a regular structure of clustered memory and execution on cache-based platforms. The resulting execution model goes under the name of PRedictable Execution Model [16]. The proposed technique, namely Light-PREM [17], uses memory tracing to automate code re-factoring.

### 1.3 Intellectual Merit and Impact

Overall, the coordinated partitioning and protection of shared resources in a multi-core chip is challenging. However the benefits of the proposed approaches are significant since they allow engineers to port, develop, verify and certify applications core by core, and eventually partition by partition, as if running on a single-core chip. Without appropriate protection of the timing behavior of applications on a per-core basis, the combinatorial inter-core interference problem can easily jeopardize the system integration process and lead to undesired complexities at the level of liability management. Even if we propose an extensive set of initial solutions to these problems, it is worth to notice that large improvement margins can be further achieved by optimizing the coordinated shared resource allocation mechanisms.
We have chosen a subset of the aforementioned challenges as the main topic for this research work. Our research has focused on the fundamental problem of ensuring timing predictability for safety-critical applications on modern, multi-core platforms. This work has led to the development of novel solutions that are technologically practical and that have received good recognition by the real-time and embedded community.

More in details, we proposed several engineering solutions for real-time multi-core performance isolation. In the context of resource regulation, “Palloc” and “Cache Colored Lockdown” were developed to control cores’ memory bandwidth usage and manage cache space in a predictable manner. These techniques are transparent to the application and they are implemented in the OS. We were able to produce up to a 250% improvement in terms of worst-case execution time reduction on benchmark tasks. Furthermore, the proposed technology is compatible with legacy applications. A team of Lockheed Martin (LMCO) engineers has expressed interest in the transfer of these technologies to LMCO engineering practice. Quoting Russell Kegley, Lockheed Martin Fellow at LM Aeronautics, from January 2013: “The UIUC technology directly addresses the problem that industry practitioners are facing in migrating existing sets of applications to multicore processors”. He also added: “We expected to encounter issues with multicore processors in our laboratories, and were pleased to find that UIUC was already developing solutions to those problems”.

Apart from devising solutions to enhance predictability targeting individual architectural components, we have produced an integrated framework. In this case, our impact goes beyond its proof-of-concept implementation and proposed analysis. In fact, we in this context we have defined a philosophy for the development of hard real-time systems on architectures with parallel computation elements. At the base of this philosophy is the concept of strong inter-core performance isolation that we aim at pursuing with Single Core Equivalence (see Section 3.5). The definition of this new philosophy and the ability to provide an insight for its feasibility had significant broader impact highlighted. In fact, the SCE philosophy (more than its technology) has an important role in the new draft of “Minimal Multicore Avionics Certification Guidance” joined by Academia, Government Labs, and Industry internationally. So far, the Minimal Multicore Avionics Certification Guidance has been endorsed by Rockwell Collins Inc. USA, Lockheed Martin Corporation USA, WindRiver Systems USA, Carnegie Mellon University USA, University of York UK, BAE Electronic Systems UK, Air Force Research Lab USA, ETRI, South Korea.

Additionally, in the context of a multi-year collaboration with Hitachi America Ltd. we explored a from-scratch design of a real-time OS (RTOS) revolving around the idea of explicit shared memory resource management, ultimately defining the nuts and bolts of system-wise resource control techniques. Once again, our aim was to demonstrate that it is possible to simplify system deployment and validation on COTS multi-core systems with careful OS-level hardware resource management. Our evaluations also revealed that performance improvements are possible while achieving strong performance evaluation, with boosts of up to 200% with respect to legacy execution models.
1.4 Structure

The reminder of this dissertation is organized as follows. First, Chapter 2 reviews important background concepts. A brief discussion of the work related with this dissertation is provided in Chapter 2. Next, Chapter 3 describes our work on techniques to achieve strong performance-isolation on cache-based multi-core systems. In Chapter 4 instead, we focus on scratchpad-based multi-core platforms and we provide a description for the proposed OS-level design to achieve predictability via memory, CPU and I/O co-scheduling. A set of methodologies and tools that can be used on both scratchpad-based and cache-based systems to perform application behavior inspection is discussed in Chapter 5. Chapter 6 concludes the work. In this chapter we also discuss the limitations of the presented approaches, review the ongoing trends in computing platforms, and provide an overview of possible extensions for this research.
This chapter introduces a number of background concepts required to better understand the work proposed in the following chapters. Hereby, we first introduce a set of general features considered on cache- and scratchpad-based multi-core architectures. Next, we revise some basic concepts related to CPU cache and DRAM operations. Finally, we briefly review the related work in the context of hardware resource management for real-time systems, with specific emphasis on works that target COTS multi-core systems.

2.1 Multi-Core Architectures

Modern multi-core architectures, their structure and their features represent a central topic in this work. As multi-core platforms have become mainstream in the embedded systems domain, different vendors and manufacturers have incorporated a number of stable and experimental features. These features often attempt to capture additional requirement of the market segment they target. For instance, they offer extensive performance monitoring capabilities, and/or support for error correction codes (ECC) in hardware. Nonetheless we can distinguish between two main trends in the multi-core computing industry:

1. Cache-based Architectures;

2. Scratchpad-based Architectures.

Despite the profoundly different organization of hardware resources in the two classes of multi-core architectures, a number of similarities can be highlighted. Both architectures feature a number of processing elements, also referred as cores, processors, or CPUs. Throughout this work, we use any of these terms interchangeably. Each processor in a multi-core system represents a hardware component with parallel circuitry for instruction decoding, memory addressing, register files, interrupt handling, arithmetic operations and the like. In order to execute instructions and process data, the CPUs access the main memory via a shared interconnect. In general, we consider multi-core architectures that adhere to the Von Neumann architecture, where data and instructions are stored in the same physical memory space. Each core fetches its

\footnote{This may not be true as many micro-controllers use flash memory to store non-volatile data. In these cases, we first copy relevant data into main memory at boot time. Thus, real-time tasks execute from either cacheable main memory or scratchpad memory.}
own instructions and operates on its own data characterizing a Multiple Instruction Multiple Data (MIMD) system. Since the cores can initiate memory transactions on the shared interconnect, they are said to be masters on the interconnect. We will often use the term bus to refer to the main interconnect. Once a main memory request is initiated, a memory controller responds as a slave to the request and is responsible for (i) fetching the data from the memory storage; and (ii) formulate a response to the original request with the requested data as the payload. In general, the CPUs may not be the only masters on the interconnect. Digital Signal Processors (DSP) and other specialized co-processors, Direct Memory Access engines (DMA), and DMA-capable devices, to name a few, can also initiate memory transactions and represent memory masters. We will consider the activity/presence of additional bus masters when they play an active role in the system. The technology used to implement buses, main memory and controllers highly varies across different architectures, vendors and even revisions of the same chip. For this reason, we consider and abstract the behavior of two main classes of memory technology: Dynamic Random Access Memory (DRAM) and Static Random Access Memory (SRAM) [18].

The majority of the proposed techniques have been implemented and validated on COTS multi-core platforms. Often, we rely of hardware-specific features to produce working proof-of-concepts. We typically focus our attention on: ARM Cortex-A9 SoCs implemented by Samsung and Freescale/NXP; Freescale/NXP QorIQ P40xx platforms; and Freescale/NXP MPC57xx micro-controllers. Nonetheless, we explicitly list in the following sections other commercially available platforms that provide compatible hardware support to implement the proposed techniques. In general, few key requirements exist on the considered platforms. First, in order to control whether memory requests are forwarded to the main memory or resolved locally (i.e. without traversing the interconnect), we either require capabilities for cache locking (see Section 2.2.2) or the presence of core-local scratchpad memories (see Section 2.4). Second, if DRAM memories are used, we will require a basic knowledge of the way requests for physical memory are translated by the memory controller. This information can be retrieved from the SoC technical reference manuals or experimentally derived using synthetic benchmarks, as discussed in Section 3.3.

2.1.1 Cache-based Architectures

An high-level representation of the main components in a cache-based multi-core architecture is provided in Figure 2.1. Apart from the interconnect and the processing elements, labeled as “Core 1” to “Core m”, we now make specific considerations on the components highlighted in red.

We assume that all the cores in the system share at least one level of cache, labeled as “shared cache” in the figure. Cache-based platforms typically feature a unique shared Uniform Memory Access (UMA) or a cache-coherent Non-Uniform Memory Access (ccNUMA), easing data sharing. Apart from a shared cache level, cores may have a private level-1 (L1) cache. As such, the shared cache is often referred as level-2 (L2) cache, or level-3 cache (L3) when an intermediate cache level exists. In general, we will often refer to the
shared cache as last-level cache (LLC), i.e. the last level of cache before the main memory. We provide a basic overview on the organization and operation of cache memories in Section 2.2. The connecting bus between the processors and the last level cache can be a dedicated one (i), if the LLC is placed before the main interconnect. Nonetheless, on some platforms (e.g. Freescale/NXP P4xxx and T4xxx series), the LLC is placed on the main interconnect (ii), i.e. between them main memory bus and the DRAM controller. From our evaluation, it emerges that the bandwidth of the bus that connects CPUs and LLC (either dedicated or the main memory bus) is typically over-provisioned and thus it is not a crucial source of inter-core interference. For this reason, we do not manage directly the memory bus. For the same reason, our techniques apply to both case (i) and case (ii) mentioned above. A request for cacheable memory originated at any of the $m$ cores performs a lookup in LLC if it misses in all the previous cache levels. If a the lookup results in a hit, no memory request is forwarded to the DRAM subsystem\footnote{In some platforms, speculative memory requests can be sent to the DRAM subsystem before the result of the LLC lookup is determined. This feature allows to partially mask the waiting time for DRAM requests after a LLC miss. For our purposes, however, we consider this feature as disabled or not implemented.} Conversely, in case of a miss, the request is forwarded to the DRAM subsystem. For this reason, there is a difference of several orders of magnitude between memory accesses that hit or miss in LLC. It follows that being able to predictably manage the LLC space is a key requirement to achieve deterministic yet acceptable performance in multi-core systems, as we discuss in Section 3.2.

The memory controller, or DRAM controller, represents a hardware component that acts as a slave on the main memory bus accepting incoming transactions, and as a master toward the DRAM storage. We provide the basic information about its inner structure and operation in Section 2.3. Due to limitations in the speed at which the DRAM storage is able to satisfy memory requests, the memory bandwidth that can be extracted from the DRAM subsystem via the DRAM controller is limited. Moreover, this limit is typically lower than the bandwidth that can be requested by the aggregated cores’ activity. For this reason, the concurrent activity of several applications can easily saturate the capacity of the DRAM controller, leading to unpredictable queuing delays. Thus, it is important to enforce a strict bandwidth partitioning.
among different cores, as we discuss in Section 3.4. Since the DRAM controller represents a key component for the performance of the system, it is typically implemented on-chip in modern architectures and exposes a series of performance monitoring capabilities.

The DRAM memory represents the actual storage of the DRAM subsystem. More details about its internal organization are provided in Section 2.3. DRAM is widely used in modern platforms because it represents a low-cost and high-capacity memory technology. In contrast, SRAM, which is faster but more expensive than DRAM, is typically used for on-chip memory resources (e.g. scratchpads, caches), where performance is a crucial requirement. DRAM memory is characterized by a simple structure: only one transistor and a capacitor are required per each memory bit, compared to the four to six transistors required for a SRAM bit. For this reason, DRAM manufacturing can reach high levels of bit-per-area density. Due to the dynamic nature of its memory cells, that need to be periodically re-charged (refresh) to hold their status, DRAM memories consume a relatively large amounts of power. As such, power consumption management for DRAM cells in embedded real-time systems is an important requirement.

Finally, I/O channels play a key role for the interaction of any SoC with the external world. Modern architectures feature a number of different types of I/O controllers: PCI, PCI-e, I²C, SPI, UART, CAN, and so on. In a typical cache-based design, all the I/O controllers interact with the processors by initiating or responding to transactions on the main interconnect. I/O controllers and peripherals that are able to initiate transactions are also called DMA-capable devices. DMA-capable devices can asynchronously initiate bus transactions and thus can represent a non-negligible source of interference. Unfortunately, in case of cache-based architectures, the hardware provides little control over the behavior of the DMA-capable peripherals. As such, two approaches are possible. First, one can insert an intermediate arbiter that is transparent from the device point of view and is responsible for buffering device-originated data. The intermediate arbiter can then be explicitly instructed by the system scheduler to forward the data on the main memory bus. This approach has been followed to design the Real-Time Bridge [19]. Nonetheless, a Real-Time Bridge is device-specific and it requires a substantial engineering effort. Moreover, the approach is suited only for devices mounted on extension slots, as it must operate between the device and the bus. The second approach consists in disabling the devices’ DMA capabilities and use explicit polling modes instead. Despite performance can be heavily impacted, this method allows direct control over peripheral traffic [20].

2.1.2 Scratchpad-based Architectures

Figure 2.2 provides a block diagram of the main components in modern scratchpad-based architectures. Once again, these architectures feature a set of application cores, labeled as Core 1 to Core \( m \) in the figure. Since scratchpad-based platforms are typically more engineered for safety and predictability, a set of important differences with traditional cache-based platforms can be noted. First, one or more of the application cores has redundant computation logic that executes the same instruction and memory operation as the main
core, with a set of hardware-implemented redundancy checks. This feature is often referred as “lockstep core”, or “delayed lockstep core”, depending on whether or not the redundancy check happens during the same execution cycle or a clock cycle after.

Clearly, a key component is the presence of scratchpad memory in the hierarchy. Nonetheless, cache memories can also be present, as it is often the case. The implemented scratchpad modules exhibit performance that are comparable to cache memories and, if efficiently exploited, can provide large boosts in application performance. On the other end, scratchpad memory is either distributed on a core-local based, or connected to a high-speed memory interface. It follows that multiple cores can access the scratchpad memory without incurring in large timing penalties. In addition, to achieve better predictability and power efficiency, DRAM memories are replaced with SRAM memories. These properties make scratchpad-based SoCs more expensive from a manufacturing point of view on a per-bit basis. In fact, the overall amount of on-chip SRAM memory available in this category of SoCs is currently around 1 MB. The majority of vendors also implement Harvard architectures, where instructions and constant data are stored in flash memory, as opposed to read/write data stored in SRAM. Thereby, the main memory storage in these architectures is composed of both SRAM and flash memory modules.

Specialization is also a key aspect of modern scratchpad-based platforms. As such, these platforms often feature a dedicated I/O subsystem that can be left isolated from the application cores. The two domains are often referred as “application shell” and “I/O shell”. The I/O shell features one or more general-purpose programmable processors, namely I/O core(s), and a dedicated bus to communicate with I/O peripherals without interfering with workload in the application shell. There is no restriction on the ISA of I/O cores. These can have a very different ISA compared to application cores, even though in the majority of cases, application and I/O cores have similar ISAs for sake of programmability. It is often the case, however, that the two types of CPU differ in a set of extensions (e.g. presence of DSP-oriented instructions) or in clock speed. It is fundamental that scratchpad memory is present on the I/O shell to buffer device data before it is transmitted on the main interconnect, and without stalling I/O devices’ operation.
2.1.3 Virtual Memory

The key idea behind virtual memory is to divide the address space of a program in blocks, called pages. Each page is a series of contiguous memory addresses. Pages are mapped into physical memory locations, but do not need to be in the physical memory to execute a program. The Memory Management Unit (MMU) translates logical addresses in a page to physical memory addresses dynamically as programs access their own pages. This mechanism is called paging and it is available on the majority of systems that support virtual memory. Paging is the key to perform some cache partitioning mechanisms, as we discuss in Section 2.2.

2.2 Cache Organization and Management

Currently, the most used cache organization is a set associative cache. In a set-associative cache, a memory block has a fixed number of positions in which it can be placed in the cache. A W-way set-associative cache has W locations for a memory block. Also, the cache consists of a number of sets, each of which consists of W blocks. A main memory block maps to a unique set in the cache and the block can be placed in any element of that set. Figure 2.3(a) shows an example of a 2-way set-associative cache.

![Figure 2.3](http://example.com/fig2.3.png)

Figure 2.3: An example of a 2-way set-associative cache (a). A main memory block can be placed in one of the two cache locations; and the structure of an address in a W-way set-associative cache (b).

The mapping from a memory block address to a set is performed by the following equation: set = (memory block) modulo (number of sets). Since the block can be placed in any element of the set, there is the need to search in all elements within a set. Figure 2.3(b) shows the form of a memory address in a W-way set associative cache. The block offset field is the address of the requested data within the block, the index field selects the set containing the address, and the tag field represents the block address.

Current memory organizations use W-way set associative caches. For example, the Samsung Exynos 4412 which implements a ARM Cortex-A9 multi-core SoC has a (configurable) 4-way set-associative L1 cache and

---

a (configurable) 16-way set-associative L2 (LLC) cache. The same design principles apply to a number of embedded platforms. When the set of cache lines is full, the cache line replacement policy must decide which line will be replaced for the new data that is being brought from the main memory. This algorithm is implemented in hardware to execute as fast as possible. Furthermore, the objective is to evict the line as optimally as possible. In this context, cache line replacement algorithms are compared to the optimal algorithm. The optimal algorithm replaces the lines that will not be used for the longest period of time. This algorithm must know all future memory accesses and thus it is not implementable. Below, we discuss some of the main cache line replacement policies:

- **Random replacement**: this policy evicts a cache line randomly. A random replacement policy is easy to implement, but its performance is bad, because the choice of an eviction is not based on the cache lines usage. For data caches, random performs on average about 22% worse than the Least Recently Used (LRU) policy (see below) [21]. Moreover, real randomness is hard to obtain in hardware, hence real components normally implement a Pseudo-Random replacement policy.

- **Least Recently Used (LRU)**: the LRU policy uses the principle of locality and is an approximation of the optimal algorithm. The objective is to evict the least recently used line. Consequently, the policy must keep track of all accesses to cache lines and replace the block that has been the least recently accessed. For large caches with a great number of ways, the use of LRU is expensive and its implementation consumes time and power [21].

- **First In First Out (FIFO)**: the FIFO heuristic evicts the cache lines in a sequential order, replacing the oldest cache line. This policy also takes advantage of the principle of locality, but is simpler than the LRU. However, for data caches this policy performs on average about 20% worse than LRU. This replacement policy is also often referred as Round-Robin (RR).

- **Pseudo-LRU (PLRU)**: the LRU policy has good performance, but requires a complex hardware to keep track of block accesses. The PLRU policy is an approximation of LRU policy which uses less computational resources. Due to this approximation, the least recently accessed cache line is not always the evicted cache line [21]. There are several different ways to implement PLRU. Two examples are the Most Recently Used-based Pseudo-LRU (PLRUm) and the Tree-based Pseudo-LRU (PLRUt). See [21] for an overview of these two policies. PLRUm ranges from 1% worse than to 3% better than LRU and PLRUt is 1-5% worse than LRU [21].

2.2.1 Problems with Cache Sharing

In commercial multi-core systems, contention for allocation of memory blocks in last level cache is a major source of unpredictability. We distinguish four main types of cache interference:
1. intra-task interference, also known as self-eviction, occurring when a task evicts its own cached blocks;

2. inter-task interference, occurring whenever a task evicts cached blocks of another task scheduled on the same CPU;

3. cache pollution caused by asynchronous Kernel Mode activities, such as Interrupt Service Routines (ISR) and deferrable functions; and

4. inter-core interference, occurring when two tasks running on different cores evict each other on a shared cache.

While the first three types of interference occur in both single-core and multi-core systems, the last one creates inter-core dependency which makes traditional schedulability analysis not applicable. Due to all the different sources of interference that exist in a multi-core system, enforcing a deterministic behavior on traditional caches means making them operate in a restrictive manner. Various solutions have been proposed in the literature to address these different scenarios. A brief literature review on proposed cache management techniques is provided below.

### 2.2.2 Categorization of Cache Management Mechanisms

We categorize the cache management mechanisms for real-time embedded systems in cache partitioning and cache locking. In a cache partitioning mechanism, the central idea is to assign a portion (i.e., a partition) of cache to a given task or core in the system to reduce inter-core interference, to increase the predictability, and to ease WCET estimation. There are two forms of cache partitioning: index-based or way-based partitioning. In the former, partitions are formed as an aggregation of associative sets in the cache. In the latter, partitions are formed as an aggregation of individual cache ways. Figure 2.4 shows an example of the two cache partitioning approaches in an $W$-way set-associative cache. In Figure 2.4(a), each set is considered a different and isolated partition (horizontal slicing). One or more sets are individually assigned to a task or core and all memory allocations performed by this task or core is mapped to the assigned set(s). In Figure 2.4(b), each way is considered an individual partition and one or more ways can be assigned to a task or core (vertical slicing). Cache partitioning can be further divided in hardware- or software-based approaches. Moreover, software-based approaches may need a compiler or an OS support.

The second cache management mechanism for real-time embedded systems is cache locking. The central idea behind cache locking is to lock a portion of the cache in order to exclude the contained lines in this portion from being evicted by the cache replacement policy and due to intra-task, intra-core, or inter-core interference. Cache locking is a hardware-specific feature, which typically is done at a granularity of a single way or line. Figure 2.5 presents two cache locking variations. Figure 2.5(a) shows the locking of an entire way. The lock of a whole way means that the contents within that way across all sets cannot be evicted. Locking a whole way has not been explored deeply, because the number of ways is usually limited (in the range from
Figure 2.4: Classification of cache partitioning approaches: (a) overview of the index-based partitioning. (b) overview of the way-based partitioning.

4 to 32) [6]. Figure 2.5(b) shows the locking of an individual way. The cache line locking strategies provided by most of the current commercial embedded platforms use a set of registers to: (i) mark portions of cache (typically entire ways) as locked; or (ii) to enable the locking of all the lines fetched after the change in the status a configuration register. These mechanisms are considered non-atomic as opposed to the support in the CPU Instruction Set Architecture (ISA) for atomic fetch-and-lock instructions. Non-atomic cache locking mechanisms make it difficult to predict what is cached and what is not. Moreover, multi-core shared caches are usually physically indexed and tagged. This means that in the worst case, physical pages allocated to tasks can map to the same cache sets, and thus that no more than $W$ locked lines can be kept at the same time, where $W$ is the cache associativity. This problem can be overcome if explicit control is enforced on the physical addresses of the locked entries (usually by the OS), as proposed in [6].

Figure 2.5: Examples of cache locking variations: (a) overview of the way locking. (b) overview of the cache line locking.

Both cache locking and partitioning can be available to applications through the OS memory allocator. An OS memory allocator is responsible for managing free blocks of memory in a large pool of memory (heap), serving the application requests for memory spaces. Two main concerns are common to OS memory allocators: performance and fragmentation. Performance depends on the ability of the allocator to organize free blocks of memory such that the search for a block is performed efficiently. Fragmentation depends on the ability to avoid small gaps between allocated memory blocks. Besides performance and fragmentation, a memory allocator for real-time embedded systems must be predictable, i.e., the time to allocate a memory
block, despite its size, must be known. Furthermore, a memory allocator can also provide means for applications to allocate physical memory pages following hardware-aware policies. Hence, allocators can internally employ cache partitioning or locking mechanism.

2.2.3 Related Work

Several well-developed cache analysis techniques have been proposed for single-core processors. These techniques analyze the interference due to intra-task and intra-core cache conflicts. The latter is known as cache related preemption delay (CRPD). The CPRD focuses on cache reload overhead due to preemptions while the intra-task analysis focuses on the cache conflicts within the same task assuming non-preemptive execution. Since we are mostly interested in techniques that manage the cache (to achieve spatio-temporal isolation rather than joint analysis), a detailed description of such analysis techniques is not included in this dissertation.

Unfortunately, the single-core timing analysis techniques are not applicable for multi-core processors with shared caches. In this case, inter-core interference is caused by tasks that can run in parallel and this requires analyzing all system’s tasks. The analysis of non-shared caches has been already considered as a complex process and extending it to shared caches is even harder. In fact, the researchers in the community of WCET analysis seems to agree that "it will be extremely difficult, if not impossible, to develop analysis methods that can accurately capture the contention between multiple cores in a shared cache" [22]. Despite this challenge, few works have been proposed to address the problem of shared caches. However, these techniques are only applicable for simple architectures and statically scheduled tasks.

The first work that studies the analysis of shared caches in multi-core processors is proposed in [23]. This work assumes a system with two tasks simultaneously running on two cores with direct-mapped shared instruction cache. Later, a cache conflict graphs were used to capture the potential inter-core conflicts [24]. The work in [25] improves upon [23] by exploiting the lifetime information of tasks and bypassing the disjoint tasks (tasks that cannot overlap at run-time) from the analysis. This work assumes a task model where all tasks are synchronized. Clearly, for systems with dynamic scheduling, it will be extremely difficult to identify the disjoint tasks. Other research [26] proposes to bypass the shared cache for single-usage cache lines to avoid inter-core conflicts and therefore improve the timing analysis. For systems where tasks are allowed to migrate between cores, cache related migration delay (CRMD) has been studied in [27]. This work estimates the number of cache lines that can be reused from the L2 shared cache when a task migrates from one core to another. Due to the lack of analysis techniques for multi-core platforms with a complex hierarchy of shared caches, an empirical study has been proposed in [28] to evaluate the impact of cache-related preemption and migration delays (CPMD).

In contrast to timing analysis techniques where caches are used without restrictions, the approach of managed caches have the advantage to avoid complex analysis methods for estimating the cache behavior.
Indeed, the time predictable architecture in [29] proposes a statically-partitioned L2 cache to avoid the inter-core cache conflicts. In addition, managed caches can be used in situations where the static analysis cannot be used, for example, the case where the cache replacement policy is not documented. On the other hand, while managing the cache space provides a timing isolation between tasks, the reduced cache space may impact the task execution time. We are not aware of any work that compares the managed shared caches with statically analyzed caches.

A number of works have proposed hardware modifications to implement different schemes for hardware-enforced cache partitioning [30, 31, 32, 33, 22, 34]. Cache memories are non-deterministic system components by design. Nonetheless, many of the proposed techniques not only provide soft real-time benefits, but also provable hard real-time guarantees [30, 35, 22]. Instead of cache partitioning, many real-time oriented methodologies propose to deterministically maintain in cache a subset of cache lines. The latter approach is known as cache locking. Hardware adaptations for cache locking have been discussed in [36, 37].

In [37], hardware-assisted locking is used to bound task migration delay, while in [36] the cache controller was augmented to retain in cache those lines that cause more cache misses during the execution of a task.

In order to perform software-based cache partitioning, page coloring is often used [38, 39, 40, 41, 42, 43]. Page coloring leverages the virtual-to-physical page address translations layer to manipulate the position of user-space memory pages in physical memory, and hence their mapping in physically-indexed caches. In MC$^2$ [43] memory colors are considered shared resources accessed in a serialized scheme. In [44], different page colors are used for applications’ and OS’s dynamic data. Changes to the compiler were first proposed in [39] as a way to enforce software-based cache partitioning for real-time applications, while a combined compiler/hardware approach was presented in [45]. Cache locking has been largely investigated using a combination of (existing) hardware support and software management [46, 47, 48, 49, 50, 51, 52, 53, 54].

Intra-task and intra-core interferences using static cache locking of instruction caches was investigated in [46, 47, 48], while a dynamic approach based on control-flow graph analysis was devised in [49]. Next, [50] combines compile-time analysis with cache locking of data to improve the WCET bounds. Similarly, three algorithms to perform locking of instructions in cache based on compile-time data were proposed in [51]. An OS-augmentation to perform profiling-driven cache allocation at context switches was proposed in [52]. A flexible approach to cache allocation via locking that considers the iterative nature of real-time algorithms is studied in [53]. In this context, we proposed a memory management framework that performs deterministic allocation of instructions and data in cache via application sandboxing, profiling and frequency analysis [6]. The importance of profiling to drive cache allocation was also discussed in [54]. The task profiling strategy proposed in [54] relies on cache miss statistics obtained via hardware performance counters to estimate tasks’ working-set size (WSS). The WSS is used as an heuristic to infer the amount of cache required to optimize tasks’ execution time. Scheduling is then adjusted based on WSS estimation. The goal is to run jobs that belong to the same application, and are likely to have the same WSS, in sequence, preventing cache-content trashing.
2.3 DRAM Organization and Management

As previously mentioned, modern DRAM memory systems are composed of a memory controller and memory devices. The controller handles requests from CPUs or DMAs (masters) and memory devices store the actual data.

![Figure 2.6: Internal organization of: (a) DRAM controller; and (b) DRAM memory](image)

The typical structure of a DRAM memory controller and the inner organization of DRAM memory storage are depicted in Figure 2.6(a) and Figure 2.6(b), respectively. The device and controller are connected by a command bus and a data bus. The controller typically has a front-end and a back-end. The front-end receives requests, keeps track of the status of the device, and generates a set of memory commands required to handle each request. The back end handles command arbitration, ensure that all timing constraints are satisfied, and issues commands to the device. Modern memory devices are organized into ranks and each rank is divided into multiple banks, which can be accessed in parallel provided that no collisions occur on either buses. Each bank comprises a row-buffer and an array of storage cells organized as rows and columns. Furthermore, some systems have multiple memory channels, each of which can be operated independently. Typically, multiple channels are configured to interleave at the cache-line granularity to improve average throughput.

In order to access the data stored in a DRAM row, an Activate (ACT) command must be issued to load the data into the row buffer first before it can be read or written. Once the data is in the row buffer, a CAS command (read or write) can be issued to retrieve or store the actual data. If a second request wishes to access a different row from the same bank, the row buffer must be written back to the array with a Pre-charge (PRE) command first before the second row can be activated. Finally, since DRAM storage

---

*Footnote: DRAM rows are also referred to as ‘pages’ in the literature; we use the term rows to avoid confusion with a virtual memory page.*
contains capacitors, the device must be periodically restored to full voltage level, a periodic Refresh (REF) command must be issued to all ranks and banks. The result of REF is that all row buffers are written back to the data array (i.e., all row buffers are empty after refresh).

Due to hardware limitations, the memory device takes time to perform different operations and therefore timing constraints between various commands must be satisfied by the controller. The operation and timing constraints of memory devices are defined by the JEDEC standard [55]. The key results of these timing constraints are: 1) the latency for accessing a closed row is much longer than accessing a row that is already open. 2) Different banks can be operated in parallel since there are no long timing constraints between banks. In particular, the nominal device throughput can only be achieved by simultaneously reading or writing in multiple banks at once.

Due to these reasons, the vast majority of COTS DRAM controllers employ an interleaved bank strategy, together with per-bank request queuing. Under this scheme, consecutive memory blocks in physical address space, typically of the size of a memory page, are allocated to different banks. Once a request of a core reaches the front-end of the memory controller, its physical address is used to identify the targeted bank and then the corresponding commands are entered in a queue for that bank. Since modern cores can issue a large number of outstanding memory requests, with this mechanism even a single application can simultaneously open and access multiple rows in different banks, thus increasing its average memory throughput.

2.3.1 Problems with DRAM Bank Sharing

Unfortunately, the discussed strategy has two major shortcomings when applied to real-time systems. First of all, there is no guarantee that an application will indeed be able to access multiple banks: if the OS is unaware of bank interleaving, in the worst case it could allocate all memory pages for that application to the same bank, resulting in much increased memory latency compared to the average case. This dependency on run-time decisions by the memory allocator can be a significant potential source of unpredictability.

Second, even if the application memory is spread over multiple banks, throughput can be significantly degraded in a multi-core system due to the effects of bank sharing. Since banks are interleaved, any core in the system can access any bank. If two applications running in parallel on different cores access two different rows in the same bank, they can force the memory controller to continuously pre-charge the row buffer and open a new row every time an access is performed. This loss of row locality can result in a much degraded row hit ratio and thus a corresponding latency increases for both applications. Furthermore, it creates a dependency between applications running on different cores, which can greatly complicate timing analysis. Finally, notice that in the worst-case both issues can be simultaneously manifested, resulting in the worst-case scenario of Figure 2.7 where all cores access the same bank at the same time.

To avoid the bank sharing problem while still allowing for bank-level parallelism in a multi-core system, existing work in the area of predictable real-time controllers has proposed to employ a private bank scheme,
Figure 2.7: Best-case (a) and worst-case (b) scenario in the pattern of access of multiple cores to DRAM banks.

where each core is assigned either one or a set of exclusive banks [56, 57]. Since under these schemes banks are not shared, cores cannot interfere with each other by closing rows opened by another core. As a matter of fact, interference is mostly limited to data bus contention, resulting in better worst-case latency bounds [57]. Unfortunately, the discussed solutions suffer from two main limitations: first, COTS memory controllers do not typically support private bank allocation. Hence, partitioning banks in hardware would require modifications to existing memory controllers, which is undesirable in COTS-based systems. Second, hardware bank partitioning can be highly inflexible, since banks must be statically allocated to cores.

2.3.2 Related Work

The basic idea of using DRAM organizational information in allocating memory at the OS level is explored in several recent work [58, 59, 60]. In [58], the goal is to maximize throughput by enforcing randomness in the page frame allocation decision. This approach works when the number of banks is much larger than the number of cores as the probability of bank-conflicts would be low. It does not, however, eliminate bank-conflicts hence the worst-case behavior is still the same. Our work is different in that we focus on performance isolation and partition banks to completely eliminate inter-core bank-conflicts. In [59], the main goal is to reduce power consumption by arranging page allocations in part of DRAM to allow the unused part of DRAM to make transitions to low-power states. While their application level coloring scheme can also be used for performance isolation, their kernel level memory allocator does not naturally support bank (and rank) interleaving modes, which are necessary for bank-level partitioning. The work in [60] is most similar to our work as it also partitions DRAM at the bank level amongst cores. The differences are it focuses on throughput and the evaluation is based on a simulator while our work focuses on performance isolation and is based on actual implementation on two real hardware platforms.
There are several predictable memory controller proposals that are closely related to our work [56, 57, 61, 62, 63]. The work in [56] and [57] both use the private banking scheme similar to our work. Because each core accesses its own banks, interferences due to bank sharing are eliminated. They differ in that the controller in [56] uses close page policy with TDMA scheduling while the work in [57] uses open page policy with FCFS arbitration. Both approaches, however, require hardware support to partition the banks from the memory controller, while our work is implemented entirely in software on COTS multi-core platforms. AMC [62] and Predator [61] utilize interleaved bank and close page policy. Both approaches treat multiple memory banks as a single unit of access which effectively transforms multiple resources into a single resource. They differ in that AMC uses a round-robin arbiter while Predator uses the credit-controlled static-priority (CCSP) arbitration [64], which assigns priority to requestors in order to guarantee minimum bandwidth and provide a bounded latency. Since both work use bank interleaving, this increases the bank conflict between the cores which cause more interference that leads to more pessimistic latency bound. Goossens et al. proposed a mix-page policy memory controller [63]. Their approach is based on leaving a row open for a fixed time window to take advantage of row hits. In the worst case, however, their approach is the same as close page policy unless the exact time at which requests arrive at the memory controller is known. Again, these proposals require hardware modifications to be implemented. On the other hand, our focus is on existing COTS systems.

2.4 Scratchpad Organization and Management

Scratchpad (SPM) memories have been proposed as a more predictable alternative to caches, especially but not limited to real-time systems. The scratchpad memory is a special SRAM memory placed close to the processor (on-chip, similar to L1-cache). The address space of the scratchpad is mapped onto predefined memory address of the processor. Unlike caches, the scratchpads have to be explicitly managed. In other words, the memory blocks have to be moved in software from main memory and copied into the scratchpad before being used. Thus, scratchpads are highly predictable in the sense that they have one access latency compared to caches with two different latencies for cache hit and miss. However, the need for explicit program management means that legacy programs cannot be easily executed on scratchpad-based systems; for this reason, support for scratchpad memory in available commercial systems is limited. We provide a brief summary of work concerning scratchpad allocation below.

2.4.1 Related Work

Most of the work related to scratchpad has focused on analyzing a program and determining with memory blocks to load into the scratchpad memory. Dynamic approaches focused on optimizing performance for a single task are discussed, for example, in [65, 66, 67]. These techniques analyze the program and divide it
into multiple regions, separated by reloading points; compiler-inserted code then dynamically copies code and data to scratchpad upon reaching a reloading point. Dynamic allocation schemes for multi-tasking real-time systems have been proposed in [68, 69]. However, they require significant hardware support to load/unload the scratchpad using DMA, and furthermore, the allocation scheme is strictly dependent on the specific scheduling policy employed in the system.

In general, cache locking and scratchpad allocation have similar objectives: to control the set of memory blocks present in local memory at any time. However, as noted in [67, 70], there are significant differences between caches and scratchpad. First of all, scratchpads do not suffer from intra-task interference, since the allocation of memory blocks is entirely under the control of the programmer/compiler; cache locking cannot prevent conflicts due to too many cache lines being allocated to the same associative set. However, since scratchpads are not transparent with respect to address translation, management schemes have to impose significant constraints on analyzable code; in particular, memory aliases must be statically resolved, since otherwise the management scheme risks loading the same data into two different positions in the scratchpad.

A second important difference is related to the type of fragmentation. Cache-based systems suffer from internal fragmentation, since they cannot load data blocks smaller than the size of a cache line. Scratchpad-based systems suffer from external fragmentation, since they are forced to load memory blocks of varying size into contiguous memory locations in the scratchpad. As shown in [67], the relative performance of locked caches vs scratchpads is thus dependent on the size of the considered memory blocks. Due to the discussed differences, we argue that most techniques and results for cache-based systems cannot be directly applied to scratchpad-based systems and vice versa.

In fact, a number of works have explored the benefits of scratchpad memories over traditional caches for multi-core platforms [11, 12]. Other works on embedded systems exist that propose scratchpad memory allocation strategies targeting real-time applications [71, 72, 73, 74, 75]. In [76], the authors propose a scratchpad memory management technique for preemptive multi-tasking systems where they introduce three methods for SPM partitioning that are: (i) spatial partitioning, where each task has its exclusive space in SPM; (ii) temporal partitioning, where a running task can use the entire SPM while its content is swapped at context switch time; and (iii) hybrid approaches where higher priority tasks can temporarily use the space of lower priority tasks.

The work in [77] assumes a similar scratchpad-based architecture to what has been used for our evaluation, namely Software Managed Multicore (SMM), with a key important difference: application cores cannot directly access main memory, but only their local scratchpad memory. The work in [77] proposes two WCET-aware mapping techniques of task memory regions to memory resources. A first technique finds optimal mapping to minimize WCET. The second approach is a near-optimal polynomial-time heuristic. This work belongs to a set of techniques for scratchpad management in SMM architectures [78, 79, 80]. These works have many points of contact with our work as they consider the use of DMA engines to manage scratchpad memory. However they are mostly concerned with minimizing task WCET via scratchpad allocation rather
then focusing on inter-core performance isolation.

Our use of scratchpad memories, described in Chapter 4, shares some similarities with scratchpad scheduling approaches that have been proposed in [81, 82, 83, 69]. Compared to these works, our approach mainly differs in three aspects: (i) it is not focused exclusively on scratchpad management, but we rather show how a scratchpad can be integrated within an overall OS design; (ii) a full OS design is implemented on a commercially available (COTS) micro-controller; and (iii) it is also discussed how I/O traffic issued by different cores is deconflicted.

OS-level strategies to perform co-scheduling of shared resources represent instances of Deterministic Platform Software (DPS) as defined in [84]. In fact, under these strategies, applications are executed following a deterministic execution model. Actively controlling and scheduling access to shared resources are crucial responsibilities of a DPS. Following the nomenclature proposed in [84], since tasks need to be specifically engineered and compiled to comply with our task model, the approach that we adopt is application aware. The scratchpad management scheme that we use is also consistent with the Acquisition Execution Restitution (AER) task model proposed in [85, 86].

The AER model [85] achieves predictability by executing tasks from local core memories (scratchpads), while shared memory resources are only used for inter-core communication and device I/O during acquisition and/or restitution phases. Inter-core interference arising from unregulated access to shared memory is mitigated by ensuring that: (i) the execution phase of different tasks can progress in parallel on multiple cores; and (ii) at most one acquisition or restitution phase is in execution at any instant of time. In [85], the fundamental assumption is that the total footprint of all the tasks assigned to a core fits inside the core’s local memory. In our work, we relax this constraint and only require that a task fits in half of the local memory space. This relaxation leads to important differences in the RTOS design. (i) Dynamic loading and unloading of tasks from/to local memories (together with I/O data) need to be handled. Additionally, (ii) task loading/unloading is pipelined with execution by using DMA engines. Finally, (iii) asynchronous I/O device activity is deconflicted from applications by exploiting hardware specialization at the bus level and by handling system-to-device interaction inside an isolated I/O subsystem.

2.5 Regulations for the Adoption of Multi-Core Platforms

Temporal coupling among concurrently running applications on different cores makes certification of safety-critical systems particularly challenging. Such problem particularly impacts the avionics and automotive industries, and has been largely recognized by the academic community. In the avionics domain, the Certification Authorities Software Team (CAST), an international group of avionics certification and regulatory representatives from North and South America, Europe, and Asia has formally expressed its concern toward the adoption of multi-core solutions for avionics system. In May 2014, CAST released Position Paper 32 on Multi-Core Processors (MCP, see CAST-32) to discuss topics related to safety of avionics software on multi-
core systems. Among other issues, the position paper identifies MCP Interference Channels, such as shared memory, cache and interconnect features, as possible sources of safety violations. While the position paper does not constitute binding policy on certification, it nevertheless strongly suggests that all such sources of interference must be: (i) identified; (ii) analyzed; and (iii) certifiably mitigated.

In the automotive domain, the International Organization for Standardization has published the ISO-26262 standard for functional safety \[87\] in 2011. This standard represents an adaptation from the more generic 2010 International Electrotechnical Commission (IEC) 61508 standard for “Functional safety of electrical/electronic/programmable electronic safety-related systems” \[88\]. The ISO-26262, in fact, specifically focuses on automotive systems’ life-cycle as well as vehicle controllability and guidelines for software certification. The standard introduces 4 main Automotive Safety Integrity Levels (ASIL), with ASIL-D corresponding to requirements for safety-critical components and ASIL-A characterizing non-critical/best-effort components. The ASIL level for a software component can be determined by performing risk assessment and hazard analysis. Specifically, a hazard is defined in agreement with the impact of failures resulting from faulty system states, and is adjusted for the likelihoods that the considered faults cause those failures. The ISO-26262 allows to perform ASIL decomposition by partitioning OS applications to different areas of the same multi-core ECU. The requirement, however, is that logically independent partitions run without causing any mutual interference. This property is specifically addressed as freedom from interference. Although the requirements are well understood, there is a lack of consensus on the technology for next-generation automotive systems using multicore platforms.

The requirements for avionics system provide probably the most stringent guidelines for high assurance safety-critical systems. The DO-178C \[89\] is document published by the Radio Technical Commission for Aeronautics (RTCA) in 2012 and that was accepted by the Federal Aviation Administration (FAA) as a set of official guidelines for civil aviation. The document has replaced the previous revision, namely the DO-178B \[90, 91\]. The DO-178C document impose precise guidelines for the development, verification and validation of avionics software. These guidelines are designed such the resulting software/hardware systems provide high assurance for safety-critical applications. All the RTOS’s currently used for commercial applications for safety-critical avionics functionality are certified following the DO-178C guidelines. The DO-178C, however, specifically targets single-core systems. To date, there is no document officially endorsed by the FAA that standardizes the certification process of multi-core platforms and RTOS’s for safety-critical applications in civil avionics.

It is expected however, that a possible trend for future avionic standards will represent an extension to multi-core of current avionics standards. In a number of avionics standards, such as the ARINC (Avionics Application Standard Software Interface) series – ARINC-653 \[92\] and ARINC-651 \[93\], the concept of resource partitioning is central for the design of safety-critical systems. Specifically, the ARINC-653 \[92\] standard defines a programming interface called APEX (Application Executive) where the concept of spatio-temporal partitioning has a central role. The standards explicitly impose that, in case of applications sharing
hardware resources, a misbehaving task cannot impact the timing or logic behavior of a different task. The mentioned APEX are specified in the context of Integrated Modular Avionics (IMA), formally specified in the DO-297 standard [94]. Nonetheless, the good flexibility of the guidelines, as well as the wide applicability of the spatio-temporal isolation philosophy has contributed to the application of the ARINC principles outside the avionics industry.

In a ARINC-653-compliant systems, each multi-tasking application is executed within a partition, with dedicated and protected memory resources. Many partitions can share the (single) CPU in a static time-sharing scheme (TDMA). Nonetheless, even if different partitions execute on the same physical processor, the behavior/misbehavior of a software component should not affect the execution of another component running on a separate partition [95]. In single-core systems, the requirement for inter-partition isolation can be achieved by employing time division and fault containment strategies. On multi-core systems, however, how to enforce and certify strong partitioning across different cores is still an active research topic. On one hand, this approach implies inter-partition temporal/spatial isolation, hence allowing ARINC-635-compliant systems to be certified under DO-178C. On the other hand, entire partitions can be ported across different systems that adhere to the same standard and are defined using the same APEX. It follows that the existence of a strong yet flexible standard for the certification of safety-critical systems not only streamlines the production of reliable safety-critical systems, but also has a large impact on certification and re-engineering costs. It has been calculated in 2000 [96] that the yearly saving of the avionics industry thanks to the adoption of the ARINC standards totals to more than $ 219 million.

In [97], Jean et al. provide a high-level discussion of the main issues for the extension of existing avionic standards to multi-core systems. The work considers multi-core IMA systems were partitions may run in parallel on different cores. The authors raise the concern that in the presence of faults, the use of shared hardware resources may lead to a violation of strict inter-partition isolation requirements. In multi-core systems, interference channels (if not carefully mitigated) are also present under normal operating conditions. As previously mentioned, certification authorities [5] have acknowledged this issue and it currently represents a source of concern for the use of multi-core processors in avionics systems.

2.6 Related Work on Real-time Multi-Core Resource Management

The problem of inter-core interference in multi-core architectures is well known and has been largely studied in literature. As a result, several different ways of approaching the challenges introduced by multi-core platforms have been proposed.

The problem of scheduling real-time tasks on multi-processors have received significant attention in the literature [98]. New scheduling strategies that are optimal on such platforms have been developed, such as RUN [99], Pfair [100], LLREF [102], LRE-TL [103]. However, the commonly adopted model in multi-core scheduling theory assumes that WCET of tasks does not vary according to what is being scheduled on other
cores at the time of their execution. Hence, the work on scheduling well integrates with the proposed resource regulation techniques as they create an abstraction where the constant-WCET assumption is guaranteed.

Static analysis tools that leverage abstract interpretation \[104, 105, 106\] and symbolic execution \[107, 108, 109\] have been developed. Using such techniques it is possible to analyze a program from binary, reconstructing execution paths and extracting information about memory access patterns. In order to scale to multi-cores, however, this approach requires extensive knowledge about those software components that can run in parallel. Alternatively, resource usage properties exported by resource regulation techniques could be embedded into static analysis techniques to limit the analysis to a single application and achieve composable results.

On multi-core, static analysis for caches proposed by \[110\] has been extended to shared caches \[111, 112, 113\]. Similarly, shared memory buses have been analyzed in \[114, 115\]. A complete attempt of system-wide static analysis which includes shared cache and bus in a multi-core platform has been proposed in \[116\], on top of which a unified WCET framework has been formulated \[117\]. Overall, an interesting plethora of work has been developed in this direction. However, these works need to make strong assumptions about: (i) the behavior of some architectural components (e.g. TDMA-based buses or pure LRU-based caches); and (ii) the presence of a fixed workload in the system, meaning that system-wide analysis must be reiterated if changes to individual software components are made.

A radically different approach is to design multi-core and many-core architectures that provide better guarantees on the worst-case execution time of critical software components. The precision timed (PRET) architecture \[118, 119\]; the MERASA project \[120\] and the P-SOCRATES project \[121\] are ongoing projects to achieve real-time guarantees on top of specifically engineered multi-core and many-core platforms. Although careful hardware design can provide stronger guarantees, our goal is to develop regulation techniques that are implementable on existing COTS platforms.

In line with this philosophy, Agrawal et al. \[122\] presented a preliminary work for enabling slot-level time-triggered scheduling on a real COTS multicore platform like the P4080. This work addresses the problem of inter-core interference and provides a mitigation strategy for static off-line schedulers like cyclic executive. Similarly, the MC\(^2\) framework \[123, 124\] has been developed to execute mixed-criticality workloads on multicore, allowing different scheduling techniques to be adopted at different criticality levels. Notably, the MC\(^2\) framework has been designed and implemented on ARM Cortex-A9 COTS SoCs.

Temporal predictability is a crucial design-time constraint for real-time operating systems (RTOS). Several RTOS designs have been proposed, and a number of implementations are available, such as: QNX Neutrino\[^5\], FreeRTOS\[^6\], Wind River VxWorks\[^7\]. These RTOS were designed for single-core platforms, where the use of real-time scheduling policies, efficient inter-process communication and prioritized interrupt handling were enough to ensure temporal predictability. The shift in computer manufacturing from single-core systems to

\[^6\]http://www.freertos.org/
\[^7\]http://www.windriver.com/products/vxworks/
multi-core systems has induced a profound transformation in the way OS’s and real-time OS’s (RTOS) are
designed and implemented. Among the most formidable challenges that the OS and real-time community
has faced in this transition the following have received a remarkable amount of attention: multi-core task
scheduling, multi-CPU load balancing, efficient and predictable resource serialization, predictable interrupt
and exception handling. Nonetheless, comparatively less work has been invested in adapting modern OS-
RTOS design to address a new, yet fundamental set of challenges to achieve predictability on multi-core
systems. The new challenges arise from the extensive sharing of hardware resources that cause inter-core
timing interference. In this sense, the development of an additional layer for shared resource management
at an OS level [119, 120, 8] is key to achieve predictability in multi-core systems.

2.7 Related Work on Workload Profiling

Trace-based profiling is only one of the possible approaches to workload profiling, and a plethora of estab-
lished techniques that reason offline about the application behavior exist. A body of work in this direction
uses abstract interpretation [104, 125] to determine the possible sequences of memory references across all
the possible execution paths of a given task. Symbolic execution [107] represents another technique that has
been largely used to determine possible execution paths in the control flow of a task, and thus the set of
possible memory references [126, 127]. These methodologies, however, are typically able to derive only a safe
super-set on the memory accesses performed by an application. Unfortunately however, complex application
behavior and unknown hardware functionality in COTS systems can lead to large resource over-provisioning.

A second approach for accurate memory profiling consists in performing instrumentation of the task code.
In this case, tracing routines are invoked when instructions that perform memory accesses are encountered.
Tools that allow code instrumentation are usually classified in two main categories: source-level and binary-
level. When instrumentation is performed on the binary code, additional instructions are dynamically
added to a compiled program. This can be done before the code starts its execution, as in QPT [128]
and Pin [129], or while the program is running, as done by Valgrind [130]. Conversely, in the source-level
approach, instrumentation is performed at compile time. Existing automatic source-level instrumentation
is done either by performing source-to-source translation (e.g., ROSE [131]), or by introducing additional
specific compilation logic [132, 133].

A number of works in the literature has investigated the capability to understand the behavior in memory
of an application at compilation time. The work in [134] targets general purpose workload and investigates
how prefetch instructions inserted at compile time can reduce cache misses. In [135] three algorithms are
proposed to perform efficient prefetching of array-based structures in scientific applications. The extension of
such algorithm that exploits multiprocessing to mask memory latency is detailed in [136]. In [137, 138] more
general algorithms are proposed to generate prefetch sequences by analyzing stride memory access patterns
to handle both array-based and pointer-based structures. Efficient compile-time strategies to automatically
insert prefetch statements, when pointer-based and recursive structures are in use, are studied in [139, 140]. Similarly to our Light-PREM [17] some prefetching techniques rely on memory traces [141, 142] to detect stride addressing patterns.

In our work, we use memory access detection based on memory protection mechanisms (see Section 5.1) and dynamic binary instrumentation (see Section 5.2) and demonstrate that it is possible to acquire accurate memory traces to drive cache allocations or to restructure workload execution (see Section 5.3).
CHAPTER 3

PREDICTABILITY ON CACHE-BASED ARCHITECTURES

As previously stated, hardware resource sharing on multi-core platforms represents the primary source of non-determinism. As a result of the concurrent requests generated by multiple CPUs, shared resources can become saturated and represent a bottleneck. From a real-time perspective, if a given shared resource becomes a bottleneck, the measured worst-case execution time (WCET) of a task in a core can vary significantly when that resource is heavily used without deterministic partitioning. This is because severe congestion can be experienced leading to unpredictable execution delays. It follows that one strategy to mitigate the occurrence of inter-core interference is to restrict the hardware behavior such that shared resources are never utilized beyond their saturation point.

Avionic and automotive industries have a large base of certified software developed for single-core chips. However, due to inter-core interference, schedulability analysis results derived for single-core systems cannot be reused when migrating to multi-core platforms. In this context, there is the need for a solution that allows reusing consolidated software components, development, schedulability analysis and certification process as is.

In order to attain this goal, we focused on the memory hierarchy as it represents the primary source of inter-core performance interference. The memory hierarchy can be decomposed at a high-level in three layers: (i) cache; (ii) DRAM controller; (iii) DRAM memory (see Section 2). For each of these layers we have developed a regulation solution to mitigate interference. We have finally integrated these solutions to create a unified framework, namely Single-Core Equivalence (SCE), to enforce real-time performance isolation among CPUs in a multi-core system. As a result, SCE is able to provide both performance benefits, since multi-core platforms can be fully exploited, and a reduction of certification costs. In this chapter, we briefly discuss the software solutions that compose SCE and the main theoretical results about SCE.

3.1 System Model and Assumptions

This section summarized the assumptions about our task model, scheduling and hardware features. A summary of the parameters and symbols used in this chapter is provided at the end of this section.
Processors In our system, $m$ represents the number of active cores. In this work, we assume that all the processors have the same speed and ISA, i.e., we consider homogeneous multi-core platforms. We assume that processors are out-of-order and that each core is allowed to have more than one outstanding memory request. We assume that dynamic power and frequency scaling capabilities are disabled or not implemented.

Task model We consider a system with mixed criticalities, where both critical and non-critical tasks can be active at the same time. For the sake of simplicity, we assume a partitioned, priority-based scheduler, even if this work can be easily extended to systems having a global scheduler. In our model, non-critical tasks are considered as black boxes, and we do not enforce any constraint on them. On the other hand, we assume that critical tasks are periodic and have an initial start-up phase executed once for initialization, and a series of periodically released jobs.

Moreover, for the purpose of cache profiling, an additional constraint is required: that the task under analysis does not perform any dynamic memory allocation after the start-up phase. This is because dynamic allocations can trigger the creation of a new memory region. As we will show, we need the set of memory regions detected at the end of the start-up phase to remain stable throughout the task execution, and we need that the number and the order of such memory regions is unaltered from execution to execution. Note however that when dealing with real-time systems, this assumption is generally satisfied, since dynamic memory allocations are a source of unpredictability and are strongly discouraged for certification purposes [143].

We use $N$ to indicate the number of critical real-time tasks in the system with decreasing priority. As such, we consider a set of implicit-deadline tasks. $\Gamma = \{\tau_1, \ldots, \tau_N\}$ is the critical task set. Each task $\tau_i$ ($1 \leq i \leq N$) is composed of a start-up phase $s_i$ and a sequence of jobs $\tau_{i,1}, \tau_{i,2}, \ldots$. The start-up phase is used to perform initialization and resource allocation, and it does not play a role in the schedulability analysis. Conversely, periodically activated task instances (jobs) have timing constraints. Thus, each job in a task $\tau_i$ is characterized by a WCET in isolation $C_i$ and is released periodically every $T_i$ time units. Each job $\tau_{i,j}$ is preemptive. Furthermore, we define as $P_i$ the set of the $r_i = |P_i|$ most frequently accessed memory pages $\{P_1, \ldots, P_{r_i}\}$ addressed by the jobs in $\tau_i$.

One of the most important goals of this work is that if $m$ is the number of active cores in a system, then $1/m$ of shared memory resources are assigned to each core. Consequently the WCET of a task directly depends on the number of active cores and can explicitly denote this dependency with the notation $\text{WCET}(m)_i$. As such, $\text{WCET}(m)_i$ is the WCET of the task under analysis when $m$ cores are active. It follows that $C_i = \text{WCET}(1)_i$. We will use the notation $C_i$ or $\text{WCET}(1)_i$ as well as $C_{\text{sce}_i}$ or $\text{WCET}(m)_i$ interchangeably. While $C_i$ is known and can be measured on the platform or via single-core static analysis, the value of $C_{\text{sce}_i} = \text{WCET}(m)_i$ needs to be derived, as we show in Section 3.3.4.

A second important aspect is that through cache management, the maximum number of residual last-level cache misses is known at analysis time. The upper-bound of last-level cache misses, i.e. maximum number
of task-initiated DRAM transactions (since we assume write-back cache policy) after cache lockdown is denoted as $\mu_i$. We assume that the scheduling policy is based on tasks’ fixed priority assignment (e.g. Rate Monotonic [3]) upon a partitioned multi-core system; this is a common practice used in industrial applications.

Note that the value of $C$ and $\mu$ can be derived by using either static analysis or an experimental approach. Static analysis can be used whenever a micro-architectural model for the considered platform is available [104, 105]. However, for complex architectures, it is often common industrial practice to experimentally derive the value of WCET. The measurement-based approach may determine a WCET that does not correspond to the absolute WCET. However, for COTS platforms where many micro-architectural details are undisclosed this approach is the only viable option for practically dimensioning a complex system at design time. Existing tools that adopt this approach are part of the industry practice toward WCET determination on single-core platforms [144, 145].

**Cache model and features** Our work can be applied to systems featuring a last level cache organized as a write-back physically indexed, physically tagged $W$-way set associative cache. Furthermore, we assume that the way size in bytes is a multiple of the memory page size. This is required to enforce memory coloring at a granularity of one memory page. This assumption is generally valid because modern embedded systems typically have pages of 4 KB and caches in the range from 128 KB to 2048 KB, with a number of ways $W$ in the range of 4 to 32.

We also assume that the system provides a set of registers or dedicated instructions to manage the behavior of the cache controller. In particular, to enforce a deterministic behavior in a SMP system (without stalling any CPU), we have to avoid evictions caused by other cores while fetching on a given CPU. This can be done if: (A) there exists an atomic instruction to fetch and lock a given cache line into the cache, or (B) it is possible to define, for each single CPU in the system, the lockdown status of every cache way. The last mechanism is called *lockdown by master* in multi-core systems, and can be thought as a generalization of the single-core equivalent lockdown by way [146].

On a dual-core platform featuring a 2-way set associative cache and a controller with a lockdown by master mechanism, we can set up the hardware so that way 1 is unlocked for CPU 1 and locked for CPU 2, while way 2 is locked for CPU 1 and unlocked for CPU 2. This means that: first, a task running on CPU 1 will deterministically allocate blocks on way 1; second, blocks allocated on way 1 could never be evicted by a task running on CPU 2. The mirrored situation happens on way 2 referring to CPU 2. This assignment can be easily changed at runtime by manipulating a set of registers provided by the cache controller interface.

If the platform provides an atomic instruction to fetch and lock a cache line, a software procedure that realizes a mechanism functionally equivalent to the lockdown by master can be easily built. Thus, in the rest of this work we assume that the system provides either a lockdown by master or a lockdown by line mechanism. Table 3.1 lists the lockdown features of a few current embedded multi-core systems.
<table>
<thead>
<tr>
<th>Name</th>
<th>Cores</th>
<th>Cache size</th>
<th>Master</th>
<th>Line, atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>TI OMAP4430</td>
<td>2</td>
<td>1024 KB</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>TI OMAP4460</td>
<td>2</td>
<td>1024 KB</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Freescale/NXP P4080</td>
<td>8</td>
<td>2048 KB</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Freescale/NXP P4040</td>
<td>4</td>
<td>2048 KB</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Freescale/NXP T4240</td>
<td>12 (24 threads)</td>
<td>6 MB L2, 1.5 MB L3</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Freescale/NXP LS2045A</td>
<td>4</td>
<td>1024 KB</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Freescale/NXP LS2085A</td>
<td>8</td>
<td>1024 KB</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Freescale/NXP i.MX6Q</td>
<td>4</td>
<td>1024 KB</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Nvidia Tegra 2</td>
<td>2</td>
<td>1024 KB</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Nvidia Tegra 3</td>
<td>4</td>
<td>1024 KB</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Xilinx Zynq-7000</td>
<td>2</td>
<td>512 KB</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Samsung Exynos 4412</td>
<td>4</td>
<td>1024 KB</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.1: Lockdown features on multi-core embedded systems

Recently, it has been shown that Miss Status Holding Registers (MSHR) represent a shared resource among multiple cores and can become a significant source of interference [147]. MSHR are cache-local registers that hold the status of outstanding cache misses in non-blocking cache architectures. Unfortunately, the number of MSHR available in modern architectures is limited. When all the MSHR are in use, the cores may be stalled, causing a timing dependency among cores in spite of cache space partitioning. We assume that MSHR contention mitigation techniques as the ones described in [147] are in place.

**DRAM subsystem** The following parameters model the behavior of the key components of the underlying DRAM memory subsystem. First, $L_{size}$ represents the size in bytes of a single cache line. As such, $L_{size}$ bytes will be requested and transferred to last-level cache for every suffered cache miss. Prefetchers and speculative execution units are assumed to be disabled. We also assume that the DRAM controller, as well as the bus arbiter, implement a round-robin scheduling policy.

Many modern COTS multi-core chips (e.g. Freescale MPC56xx and MPC57xx chip families) designed for safety-critical operations can be configured to implement a round-robin policy on the main memory controller arbiter. Nonetheless our analysis can also be derived under the assumption of a FIFO policy, leading to more pessimistic results. Hereby, we describe our analysis with a round-robin policy, leaving the discussion about the implications of other scheduling policies as a part of our future work.

Once a memory request is issued, we assume that the DRAM access time for a single transaction is bounded between the best-case access time $L_{min}$ and the worst-case access-time $L_{max}$. The latter parameter captures the maximum amount of delay suffered on a single memory request due to the activity of other cores. In an experimental approach, $L_{min}$ can be derived by ensuring that only the core under analysis is active, that all its requests are hitting in the same DRAM row, and that there are no dependencies between subsequent requests. Conversely, $L_{max}$ can be found by analyzing a benchmark in isolation where: a) each memory transaction has a data dependency with the previous one; b) subsequent memory requests access...
different DRAM rows. These parameters can also be rewritten in terms of bandwidth: \( L_{\text{min}} = \frac{L_{\text{size}}}{BW_{\text{max}}} \) and \( L_{\text{max}} = \frac{L_{\text{size}}}{BW_{\text{min}}} \), where \( BW_{\text{min}} \) is the DRAM guaranteed bandwidth and \( BW_{\text{max}} \) is the maximum achievable DRAM bandwidth (as measured in isolation, without regulation and while all the other cores are off).

As we discuss in Section 3.4, we employ memory bandwidth regulation. In this context, the parameter \( R \), instead, represents the budget replenishment period of MemGuard such that the memory access budget for each core will be restored every \( R \) time units. Finally, \( Q_t \) represents the number of memory accesses that the core to which \( \tau_i \) has been assigned is allowed to perform within each MemGuard period \( R \). Under even reservation, \( Q_t \) can be calculated as \( Q_t = \frac{R}{m L_{\text{max}}} \).

**Architectural considerations** In this work we do not address the problem of contention at the level of shared cache bus and controller. The cache bus is normally on-chip and features a much higher bandwidth compared to the main memory bus. Contention at this level has been found to affect some modern platforms [148]. However, due to its high bandwidth capabilities, cache buses are normally able to sustain, without hitting the saturation point, the simultaneous activity of several CPUs. According to our benchmarks, on the Freescale P4080 platform used for our evaluations, contention at this level produces no visible effects up to 4 active cores and yields negligible slowdowns with 8 active cores.

Additionally, COTS architectures are often not time-composable between CPU pipeline and cache hierarchy [149]. This means that under certain circumstances, a local reduction of execution time (e.g. a cache hit) can produce a global increment in the execution time and vice versa, determining what is known as timing anomaly. In SCE, however, cache blocks are allocated to tasks using lockdown. As a result, the memory locations that will produce a cache hit are known at profile time and do not change across different runs, while the rest of the cache is unavailable for allocation. Consequently, if timing anomalies exist, their effects are already embedded in the experimentally derived WCET. No additional timing anomalies will be experienced after production as there is no variation in the sets of accesses that hit/miss in cache.

The derived theoretical model also assumes *delay additivity*. If delays are not additive, a pending memory request (or other instruction) not only introduces a delay in the execution due to the current operation, but may also determine additional delays in subsequent instructions. As previously discussed, an exact microarchitectural model is typically not available for COTS systems. Thus, formally determining and modeling their delay additivity is often unfeasible. Conversely, one of the main goals of this work is to derive WCET estimates that can be used at design time for complex systems through a measurement-based approach. In practice, however, the following consideration can be made: out-of-order processors like the one considered in our evaluation are likely to satisfy delay additivity. In fact, instructions that cause long delays, like those accessing memory, can execute in parallel with different operations or with instructions of the same type. This effect counterbalances memory-induced delays, so that the resulting increase in execution time can be less or equal than the single instruction delay.
3.1.1 Summary of Parameters

Table 3.2 summarizes the list of parameters that will be used throughout this chapter to perform cache allocation, DRAM management and to estimate the value of WCET($m$) for a task in a system that is compliant with the described assumptions.

3.2 Colored Lockdown - Cache Management

Multi-core architectures often feature several levels of CPU cache. We primarily consider the problem of efficiently managing shared caches, even though some of the concepts also apply to lower (private) cache levels [150]. Specifically, we have developed a mechanism, Colored Lockdown [6], that is able to solve inter-core interference on allocated memory areas and provide a good trade-off between efficiency and flexibility. In order to perform Colored Lockdown, it is necessary to know the cache profile of the considered task. We
discuss in Section 5.2 how such knowledge can be acquired.

Multiple DRAM pages can be mapped to a given set of shared cache pages. The pages in the same set are said to have the same “color”. Pages with the same color can be allocated across cache ways, so that as many pages as the number of ways can be allocated simultaneously in last level cache. The color only depends on the physical address of the considered page, hence it is possible to rely on the virtual-to-physical address translation layer to perform a remapping of pages in cache that is transparent to user-space applications. In this way, our technique is able to reposition task memory pages within the available colors, in order to maximize allocation flexibility.

Real-time applications are dominated by periodic execution flows. This characteristic allows for an optimized use of last level cache by locking hot pages first. Relying on profile data, we first color frequently accessed memory pages to remap them on available cache ways; next, we exploit hardware cache lockdown support to guarantee that such pages (once prefetched) will persist in the assigned location (locked), effectively overriding the default cache replacement policy.

The Colored Lockdown approach consists of two distinct phases. In the first phase, called start-up color/way assignment, the system assigns to every hot memory page of every considered task a color and a way number, depending on the cache parameters and the available blocks. During the second phase, called dynamic lockdown, the system prefetches and locks in the last level of cache all (or a portion of) the hot memory areas of a given task. In our model this procedure can be deferred until the activation of the first job in the task that will address a certain set of hot memory areas, as shown in Figure 3.1b.

3.2.1 Visual example

The set of hot memory pages is known by the OS by the end of the start-up phase, when the Colored Lockdown procedure is invoked, relying on the corresponding profile file. A description about how hot memory pages are detected and about the methodology followed to construct the cache profile is provided in Section 5.2. In the rest of this section, we will assume that this information is available and focus on cache space management.

To show an example, we now refer to Figure 3.1 to show how this mechanism can work on a dual-core system featuring a 2-way set associative shared cache whose total size is $S_c = 16$ KB (4 pages). The system is using a partitioned scheduler to bind $\tau_1$ to CPU 1 and a non-critical task to CPU 2. No assumptions are made on the non-critical task. Pages $P_{1,1}$, $P_{1,2}$, $P_{1,3}$ are hot memory pages for jobs in $\tau_1$, however $P_{1,2}$ is not accessed during the considered time window.

Recall that we use $W_l$ to identify the minimum number of cache ways that are needed to hold the considered hot pages. Once this parameter has been calculated (see Equation 3.3), the first $W_l$ cache ways are locked at system initialization. In our example $W_l = 2$, as shown in Figure 3.1l.

Using the legacy model (i.e. no property is enforced on the cache behavior) shown in Figure 3.1h, the
status of the shared cache is unknown because: (1) $P_{1,1}$ and $P_{1,3}$ can evict each other, since they have the same color if no reassignment is made; (2) the non-critical task can evict any of the cache lines at any moment. Thereby, in the WCET computation we have to consider the case in which each hot page access results in a cache miss. The status of the hot pages before any color/way assignment operation is shown in Figure 3.1c.

In the proposed model (Figure 3.1b), we execute a color/way assignment phase right after the start-up phase $s_1$. The resulting cache coordinates in terms of color and way are shown in Figure 3.1d. The assignment phase has a variable length that depends on the number of modified colors, but since it is executed before the first job is released, it has no impact on system schedulability. In our example, only page $P_{1,3}$ needs to be recolored.

### 3.2.2 Cache color/way assignment

Let $K$ be the number of available colors. In this phase to each memory page $P_{i,k} \in P_i$ ($1 \leq k \leq r_i$) of every task $\tau_i$ is associated a couple $\langle W_{i,k}, K_{i,k} \rangle$. Here, the value $1 \leq W_{i,k} \leq W_i$ encodes the way in which the page $P_{i,k}$ will be prefetched and locked, while $1 \leq K_{i,k} \leq K$ encodes the color assigned to the page $P_{i,k}$. The assignment is done with just one constraint, namely that different way numbers shall be assigned to pages having the same color. The following property holds:

\[
\forall P_{i,k}, \forall P_{j,l} \quad 1 \leq i, j \leq N, 1 \leq k \leq r_i, 1 \leq l \leq r_j \\
W_{i,k} = W_{j,l} \Rightarrow K_{i,k} \neq K_{j,l} \quad (3.1)
\]
As we have previously mentioned, page coloring is a software technique which can be operated at the OS level to control the mapping between physical memory pages and cache blocks. To understand how this can be done, refer to Figure 3.2. Figure 3.2a shows the structure of a physical address from the point of view of the cache controller, as divided in tag, index, and offset. Figure 3.2b shows the same physical address from the point of view of the OS in a system which uses virtual-to-physical address translation.

Figure 3.2: Physical address structure as seen by the cache controller (a) and the OS (b).

We define the color of a cache page as the value of the (physical) address bits \([I: 12]\), where \(I\) is the most significant bit of the cache index (while the 12 less significant bits encode the offset in a 4 KB page). Color bits (highlighted in the gray area) have the following properties: (1) given the virtual address of a memory page, by manipulating color bits stored in the corresponding Page Table Entry the OS can influence the mapping of memory to cache page; (2) the operation is completely transparent to the task running in user space.

In an \(W\)-way set associative cache having size \(S_c\) bytes, with a page size of \(P_s\) bytes, the number of available colors \(K\) is given by:

\[
K = \frac{S_c}{W P_s} \quad (3.2)
\]

Given \(K\), it is possible to determine \(W_i\) for a task set:

\[
W_i = \left\lceil \frac{1}{K} \sum_{i=1}^{N} r_i \right\rceil \quad (3.3)
\]

If the way size in bytes is greater than the size of one memory page, the value of \(I\) can be calculated as:

\[
I = \log_2 \left( \frac{S_c}{W} \right) - 1 \quad (3.4)
\]

In the example shown in Figure 3.1, the number of available colors is \(K = 2\). Then, since the number of ways is \(W = 2\), no more that two pages can have the same color. In the example, \(P_{1,3}\) has been recolored from color 1 to color 2 (Figure 3.1; and 3.11). In our example, \(P_{1,3}\) has been assigned to way number 2,
and this information will be used later to determine where to prefetch $P_{1,3}$.

### 3.2.3 Dynamic lockdown

During this phase we execute the prefetch and lock procedure on the given set of hot pages. As previously stated, this step can be executed at different times, depending on when the colored hot pages are requested. In our example in Figure 3.1, page $P_{1,1}$ is prefetched right at the end of the previous phase, while $P_{1,3}$ is prefetched at the beginning of job $\tau_{1,2}$. Conversely, in the context of our Single-Core Equivalence (SCE) framework (see Section 3.5), this operation is performed at the beginning of each IMA partition instance.

As previously stated, an invariant of the procedure is that before and after the routine is executed, the first $W_l$ cache ways are locked for every CPU in the system, so that none of them can perform an allocation in said ways.

Since lockdown by master lets us define a different cache way lockdown status for each CPU in the system, in order to prefetch and lock a given page $P_{i,k}$ on a given CPU, say CPU $h$, we operate as follows: (1) we make sure that none of the lines in $P_{i,k}$ are cached in any level of cache; (2) we lock all the ways but $W_{i,k}$ for the CPU $h$; (3) we sequentially read the page $P_{i,k}$, in a non-preemptive manner, in order to trigger the desired cache allocations; (4) we restore the lock status of all the $W$ cache ways for the CPU $h$ as it was before step (2). Once this step has been executed for a given page, we flag it as *prefetched*, so that we never run this procedure twice on the same page.

If the previous assignment constraints are met, every time a prefetch and lock procedure is executed, the following assertions hold: (1) After the invalidation (or trashing) step, the data contained in $P_{i,k}$ cannot be cached in a higher level of cache; in addition, it can not be cached in any of the last $W - W_l$ ways, nor in any of the first $W_l$ ways, because they have been invalidated and locked at the very start-up of the system. (2) Every line addressed in $P_{i,k}$ will cause a cache allocation in exactly the way $W_{i,k}$, since any other way is locked for the CPU which is running the prefetch. (3) Inside the given way $W_{i,k}$ the prefetch procedure will not evict any previously prefetched page because page $P_{i,k}$ has a color $K_{i,k}$ which no other page locked on the same way can have (See Equation 3.1).

Therefore, if the prefetch and lockdown procedure has been executed for a given page $P_{i,k}$, then every access of a job in task $\tau_i$ to any cell contained in $P_{i,k}$ will certainly result in a cache hit. Thanks to this property, the maximum number of cache misses $\mu_i$ for task $\tau_i$ can be computed. This property is exploited at analysis time in order to compute tight WCET bounds for the critical tasks, as we discuss in Section 3.5.4.

Note that a part of the cache will be unusable by non-critical tasks, and this can have a negative impact on their average performances. However, this is usually an acceptable trade-off in real-time systems. Note also

---

1 This step is hardware specific. It can be done by trashing higher cache levels and, on the last level, through by-line invalidation mechanisms (if supported), or invalidating all the last $W - W_l$ cache ways. Doing so, no line in the page that is going to be prefetched could trigger a cache hit in any other cache level or way, thereby causing an allocation in exactly way $W_{i,k}$. Other platforms (e.g. PowerPC-based systems) provide specific instructions to initiate cache prefetches targeting a specific level of cache.
that a subset of hot pages could be shared among a group of tasks. This is often the case for shared libraries. In case a library page requires to be locked in cache, it needs to be re-colored (if necessary) and locked only once. It is important that the allocation in cache of library pages is performed before any task (critical and non-critical) that used the library is started. In fact, a late re-coloring of shared library pages may introduce logic misbehavior of those applications that have already mapped the re-colored library. Another option is to prevent dynamic linking of shared libraries by using static linking in critical tasks.

Finally, based on experimental results, it is worth to mention that task execution time as a function of the number of locked pages is well approximated by a convex function (see Figure 3.4 and 3.5). This has an important consequence; in fact, whenever the total number of hot pages (in the task set) exceeds the available cache size, an optimal cache allocation to multiple tasks can be approximately computed by using a convex optimization algorithm [151] like Q-RAM [152]. The resulting curve of observed WCET as a function of hot memory pages allocated in cache goes under the name of “Progressive Lockdown Curve”.

3.2.4 Evaluation

In this section we present the results obtained with an experimental implementation of the proposed cache management scheme, as well as the relevant hardware details of the testbed platform.

**Methodology** We adopted for this work a Pandaboard development board. It is a low-cost embedded ARM platform featuring a OMAP4430 processor [153]. This processor incorporates a dual-core ARM Cortex-A9 MPCore with symmetric multiprocessing (SMP). The OMAP4430 operates at a clock frequency of 1.0 GHz. The total DRAM available is 1 GB and there are two cache levels [154]. In particular, there are two L1 caches which are private to each core. Since our interest is to study the interference on the shared cache, the L1 caches have been disabled to perform our experiments. In general, performing the evaluations on the last level (shared) cache disabling the higher levels allows us to obtain an upper bound of the response time of the cache hierarchy. Such upper bound can be more or less pessimistic according to the cache features and the memory footprint of the tasks under analysis.

The second and last level of cache is a unified, physically indexed, physically tagged L2 cache. It is shared among the two Cortex-A9 cores, has a total size of 1024 KB and it is internally organized as a 16-way set associative cache. The cache is controlled by a hardware circuit called PL310 which exposes a set of memory mapped registers to control the cache behavior [146]. The PL310 supports the following lockdown policies: by line, by way, by master. The lockdown by master policy works as explained in Section 3.1. Lockdown by way is the lockdown by master policy when a single CPU is connected to the cache controller. Finally, the provided lockdown by line mechanism is not atomic, thus implementing Colored Lockdown using such lockdown policy is non trivial.

Each Cortex-A9 core features a set of performance counters, including a 32-bit clock cycle counter. We
have used this counter to collect accurate execution time measurements during our experiments because it can be configured to be accessed from user space with a negligible overhead.

For our experiments, we have used a set of benchmarks from the EEMBC AutoBench suite [155] which are reported in Table 5.5. The benchmarks in this suite feature algorithms to test the performance of embedded platforms in automotive and industrial applications. For this reason, they share some key characteristics with real-time applications (limited memory footprint, limited use of dynamic memory, etc.) and can therefore be considered representative of a typical real-time workload.

As reported in [155], each benchmark features an iterative structure and allows us to specify a customized number of iterations per each run. However, since the execution time of each iteration is too small to be considered a job of a typical real-time task, we performed some adaptations to make each adopted benchmark compliant with the periodic task model. In particular, we have determined the number \( I \) of iterations, reported in the last column of Table 5.5, needed by the algorithm to loop over the input buffer once. In this way, we consider a job as the aggregation of \( I \) iterations. Using a combination of `setitimer`/`sigaction` system calls, a new job is released every 30 ms. Each sample presented in the following results summarizes the execution of 110 jobs. The first 10 jobs are used to put the cache in a hot status, therefore their execution time is not accounted.

Moreover, to simulate a partitioned scheduler, we bind the benchmark tasks to the first CPU and schedule them according to a real-time, fixed-priority scheduling policy. Interference on the shared cache is created running a synthetic, memory intensive task on the second CPU.

**Single-task results** Once we generated the memory profile for each benchmark, we compared the execution time of these benchmarks with and without Colored Lockdown. The results of these experiments can be seen in Figure 3.3. The plotted values are those of the observed worst case. The first bar displays the execution time of the benchmark without any protection and in isolation. The second bar displays the execution time of the same task still without protection, but, this time, with the interference task running on the second CPU. In all the benchmarks, in the latter case, the execution time increases significantly, sometimes more than 2.5 times. Finally, the third bar of each cluster displays the observed worst case execution time when interference is still present and the Colored Lockdown protection is enforced. All bars are normalized to the first bar. We observe that, in most cases, Colored Lockdown eliminates the effects of the interference task, leading us to the conclusion that Colored Lockdown effectively isolates the task by eliminating the interference on the shared last level cache. In some cases, enforcing protection through Colored Lockdown slightly improves execution time with respect to the isolation case. This is due to the fact that self-evictions (cache evictions caused by the same task) are also reduced.

The number of pages locked by Colored Lockdown in the above mentioned results were also minimized: between three to six pages were locked in each benchmark. For instance, in the case of `a2time`, we see from Table 5.5 that as many as 15 pages were found to be accessed by the benchmark, but only the four hottest
Figure 3.3: Graph of observed worst case execution time for each benchmark, with 1) no interference and no protection, 2) interference and no protection, and 3) interference and protection.

Co-scheduling results Another experiment has also been carried out to understand the behavior of multiple tasks with different priorities scheduled at the same time on the same CPU. Again, we are interested in comparing what happens if protection is enforced using Colored Lockdown versus the case in which no cache allocation is performed. The results are shown in Figure 3.6. In the graph, the first four bars represent the normalization base for the following cases and correspond to the case in which all the tasks are running without protection and no interference is generated on the last level cache by the other core. Note that, in
Figure 3.4: Graph of execution time depending on the number of pages locked, for the benchmark a2time.

this case, some cache interference coming from the other three tasks on the same CPU is suffered by each task. However, since the footprint of the considered tasks is small, applying the protection in this case does not influence performances much, as shown by the second group of bars.

According to the depicted values, in case interference in the last level cache is introduced running memory intensive processes on the second CPU (third group of bars), even the highest priority task can suffer a 2.5x slowdown. Conversely, in the last cluster of bars, we show the behavior of the system when, per each task, the number of hottest memory pages reported in Table 5.5 is colored and locked in the last level cache. In this case, the enforced protection enhances the isolation between all the tasks of the system, so that just a negligible slowdown is suffered. To understand the source of this slowdown, we can compare Figure 3.3 and Figure 3.6. In particular, considering the cases where interference is generated on CPU 2 and no protection is enforced (Figure 3.3 second bar of each cluster), we note that almost all the tasks run slower in the co-scheduling experiment (Figure 3.6 third cluster). This is because, in the latter case, DRAM accesses generated by inter-task interference on non-locked pages compete with those generated by CPU 2. In fact, this effect is not visible when CPU 2 is idle (in the first and second cluster of Figure 3.6) but generates the slight slowdown observed in the last cluster of the same result set.

3.3 PALLOC - DRAM Bank Partitioning

As described in Section 2.3, the structure of DRAM memories is organized into ranks, banks, rows and columns [156]. Whenever a given row is accessed in a bank, subsequent accesses on the same row (row-hits) can be serviced with a small latency. Conversely, if a subsequent access requires data in a different row
(row-miss), a significant increase in the latency is introduced. Different banks of the same DRAM chip can satisfy requests in parallel. The mapping between physical addresses and banks is hardware-specific and can be determined as described in [7, 2]. In a multi-core scenario, several cores can potentially access the same DRAM bank. In this case, the row-miss ratio of a task can increase as multiple cores access the same bank. In addition, reordering of requests at the bank level operated by the memory controller can significantly increase the worst-case per-bank queuing delay for the core under analysis [2].

Typically OS’s do not control how physical memory pages are mapped to DRAM banks. In order to overcome the discussed shortcomings, we have developed a DRAM bank-aware OS-level memory allocator, named PALLOC [7], which allows system designers to assign specific DRAM banks to cores (or applications). PALLOC can be used to assign disjoint sets of DRAM banks (private banks) to applications running on different cores. This way, tasks running in parallel do not collide on DRAM banks and do not suffer inter-core conflicts at this level, as long as there is a sufficient number of banks to accommodate them.

PALLOC is a kernel-level memory allocator that exploits page-based virtual-to-physical memory translation to selectively allocate memory pages of each application to the desired DRAM banks. The goal of PALLOC is to control applications’ memory locations in a way to minimize memory performance unpredictability in multicore systems. As discussed in the previous section, such unpredictability can be minimized by eliminating bank sharing among parallel executing applications. Unlike hardware based approaches [57, 56], however, PALLOC is a software based solution, which is fully compatible with existing COTS hardware platforms and transparent to applications (i.e., no need to modify application code.)

Figure 3.7 shows simplified pseudocode of the proposed allocator. For simplicity, here we assume that the kernel maintains a single free page list, freelist, although the actual implementation deals with multiple
freelists with different sizes (more details later in this section). Here, \texttt{rmqueue\_smallest()} is the main allocator function called by the kernel. For user-level applications, this is called when a page fault occurs. Instead of simply returning the head of the freelist, PALLOC maintains a set of lists—\texttt{bank\_bins}, one per DRAM bank—to quickly find a page from the selected banks, \texttt{bankmap}. If a page is found in one of the bins for the banks, it returns the page and removes the entry from the bin. If such page is not found, however, PALLOC checks every page in the freelist iteratively, until it finds a matching page. In the process, unmatched pages are removed from the freelist and inserted into the corresponding bins, to decrease the overhead of future allocations. While this greatly reduces the average overhead, PALLOC may still need to traverse the entire freelist in the worst-case. As such, we do not intend to target “real-time” memory allocation. Rather, we are interested in application’s real-time performance after memory allocation is completed. Nevertheless, we provide detailed overhead analysis in Section 3.3.1 that shows reasonable allocation time performance of our implementation. An important requirement for PALLOC is the ability to determine the bank address for a given physical address, abstracted by \texttt{addr\_to\_bank()} in the pseudocode. We describe ways to identify this mapping information in Section 3.3.1.

We implemented PALLOC in Linux. The standard kernel memory allocator used in Linux is based on the buddy algorithm [157]. The buddy algorithm aggregates blocks (buddies) of contiguous pages of exponential size (order) \(^2\) and arranges them in a set of free-lists (a list for each order). On a memory request, the allocator returns the head of a free-list with a matching size. We extend the buddy allocator to implement the algorithm in Figure 3.7. PALLOC only handles the order 0 (4 KB) page allocation, while the original buddy allocator handles the rest (order 1 or above). Because user-level memory allocations are eventually

\(^2\) An order \(N\) block refers to a block of \(2^N\) contiguous memory pages.
Figure 3.7: PALLOC implementation

performed at the page fault handler with the page granularity (4 KB). PALLOC can properly control bank assignments for user-level applications. The most common kernel internal allocation requests (getting a page frame) are also handled by PALLOC. Note that freeing pages is handled by the original buddy allocator so that the kernel maintains the buddy structure. This also allows the kernel to aggregate the freed pages to make a bigger page. In this way, the kernel can keep both bank-aware order 0 pages and bigger pages at the same time.

Finally, banks are configured through the CGROUP interface in Linux. Once a CGROUP partition is created, it provides a file interface to describe desired DRAM banks for the partition. The setting can be modified at runtime and the modified setting is immediately applied to all subsequent memory allocations in the partition. If the assigned banks are used up, in our current implementation, the standard kernel action—trying to reclaim from the page cache and swapping to the disk—will be taken even if other banks are free. For better safety, we can use the standard CGROUP memsize limit controller so that the OOM killer can kill off tasks in the CGROUP partition.

One limitation in using PALLOC is shared memory regions, such as shared program text (disk cache

---

3 We currently do not consider huge page allocation via mmap interface.
When a process executes in a CGROUP partition, the pages that are already allocated elsewhere, for example, the `bash` program text pages, will be shared instead of reallocating the pages. Therefore, these pages may not conform to the DRAM bank setting of the CGROUP. In such case, a page-migration scheme to migrate existing pages to different banks can be considered, similar to NUMA page migration \cite{158}.

### 3.3.1 Evaluation Setup

In this section, we present details on the hardware and software platform used in our evaluation. In particular, we discuss how to derive the bank mapping employed by the memory controller, and we provide overhead analysis for our Linux-based PALLOC implementation.

**Hardware platform** The primary platform used to evaluate PALLOC is a quad-core Intel Xeon W3530 based desktop computer. The processor has private 32K-I/32K-D (4/8 way) L1 cache, a private 256 KB (8 way) L2 cache for each core and a shared 8MiB (16 way) L3 cache. The memory controller (MC) is integrated in the processor and supports 1333 MHz DDR3 memory with the maximum theoretical transfer rate of 10.6 GB/s. The computer has a single-channel dual-rank 4 GiB PC10666 DDR3 DIMM module, which has 16 DRAM banks (8 banks per rank). We disabled hardware prefetchers and the turbo-boost feature to improve predictability.

We also conducted some experiments on a Freescale P4080 platform, which features an eight-core PowerPC processor. Each core has a private 32K-I/32K-D (8/8 way) L1 cache, a private 256 KB (8 way) L2 cache and shares two 1 MiB (32 way) L3 caches (cache-line interleaved). The MC supports 1333 MHz DDR3 and the platform has two 2 GiB DDR3 DIMM modules in a dual-channel configuration, each of which has 16 DRAM banks (for a total of 32 banks).

**DRAM address mapping** A prerequisite of PALLOC is the exact knowledge of the address mapping of the DRAM controller. The physical address of a memory request is translated by the memory controller and mapped onto the physical structures of the DRAM module: columns, rows, banks, and ranks. There are many possible mapping schemes for a given hardware platform. For example, one configuration could assign two most-significant bits of a physical address to the “rank” address of the DRAM module, while one could assign them to “bank” address instead.
Unfortunately, the exact address mapping information is typically not publicly available, which is indeed the case of our Intel Xeon platform. Therefore, we experimentally determined the mapping information of the Xeon platform. We use a specifically written micro-benchmark that traverses a linked list over the physical address space (by mapping the /dev/mem in Linux). The list has 32 entries and it is engineered in such a way that the distance between two successive entries is 8 MiB. The distance is large enough to span through all the memory banks in the system, so that each entry in the list will be placed in exactly the same bank, under the assumption that banks are interleaved \[^{[159]}\]. Note also that each entry is located at the exact same cache-set. Therefore, iterating the 32 entries exhausts the 16 cache-ways, forcing to access DRAM all the time. While a first instance of the benchmark is running on a core, we execute a second instance on a different core. Note that, in the latter instance, the addresses of all the entries in the list are shifted by \(b\) bits (i.e., offset = 1 \(\ll b\)). If the shifted addresses are mapped to the same memory banks as the first instance, then the measured memory performance will drop due to bank conflicts. Conversely, if the addresses are mapped to a different bank, the measured memory performance will be higher since no bank conflict will be experienced. By varying \(b\), we identify the address bits that are associated with memory banks.

On the other hand, the P4080 platform provides detailed documentation about the exact physical address-to-DRAM mapping information.

Figure 3.8 shows the identified memory mappings of both platforms. The banks, channel, and cache-sets show address bits for DRAM banks, MC channels, and L3 cache index, respectively.

**Implementation overhead analysis** We implemented PALLOC on the standard Linux 3.6.0 kernel for the Intel Xeon platform. Kernel modifications to implement PALLOC are small and easily portable: mostly by replacing \_rmqueue\_smallest() in mm/page alloc.c. We also ported it to the Freescale's custom Linux 3.0.6 kernel for the P4080 platform.

Due to the design of PALLOC, the page allocation time can vary widely depending on the availability of free pages for the selected DRAM banks. To measure the allocation overhead, we instrumented the allocator to collect time spent on the allocator, while running a synthetic program that allocates and accesses a specified amount of memory.

Table 3.3 shows the average and the worst-case per-page allocation time of PALLOC and the unmodified buddy allocator. For PALLOC, we run the benchmark on a CGROUP partition with four DRAM banks assigned to it. Therefore, the maximum allocatable memory space is 1024 MiB (256 MiB per DRAM bank). On average, PALLOC adds less than 200ns overhead for each page allocation, compared to the buddy allocator. The worst-case allocation time is, however, noticeably higher than the buddy allocator, especially as the total allocation size grows. This is because, as the number of allocated pages increases, the number of remaining pages that fall in the selected banks becomes increasingly low, even though there are free pages in other DRAM banks. Note that the experiment is unfavorable for PALLOC because all its per-bank free
lists (bins) are initially empty. In real multi-programmed environment, however, the lists can be cached by memory requests generated from other processes in different partitions, thereby increasing PALLOC’s overall efficiency. Furthermore, it is important to note that the buddy allocator can also suffer significantly longer allocation delays when the system-wide free pages are low, forcing the kernel to reclaim pages from the kernel page cache and, eventually, to swap pages to disks. For this reason, it is a standard practice to allocate memory pages at the process initialization phase and outside the critical real-time regions in order to make sure that the allocation procedures are never invoked while processing time-sensitive critical sections.

We also investigated the allocation overhead using the SPEC2006 benchmarks and found that the total allocation overhead is lower than 0.4% of the total execution time in the most memory allocation intensive benchmark. Therefore, we consider that the overhead of PALLOC is acceptable.

### 3.3.2 Results with Synthetic Benchmarks

In this section, we investigate the performance impact of using private DRAM banks, through PALLOC, with a set of synthetic benchmarks. We also investigate the architectural differences in DRAM controllers of the two hardware platforms that we considered.

In this experiment, we use a synthetic benchmark *Latency* [160] to illustrate the performance impact of using private DRAM banks. The benchmark is a pointer-chasing application using a randomly shuffled linked-list. The two successive memory instructions that access the elements of the linked list contain read after write (RAW) dependency. Hence, the second instruction cannot proceed and must stall until the first one is completed. In other words, the benchmark can generate only one outstanding memory request at a time. The size of the linked-list is configured to be two times bigger than the size of the LLC (last level cache), in order to make sure all memory requests result in cache-misses. Since the list is randomly shuffled over a big memory area, successive memory requests are likely to target a different DRAM row. Therefore,
Latency is used to measure worst-case memory performance in terms of latency and bandwidth. By running multiple instances of the Latency benchmark, we now show the performance impact of using private DRAM banks.

The experiment setup is to run a Latency instance on Core0 while varying the number of co-running instances from 0 to 3 (one instance per core). We repeat the experiment in two memory settings. In Samebank, all the memory pages of all Latency instances are allocated in Bank0. In Diffbank, Core0 uses Bank0 and the other cores use Bank1.

Figure 3.9: Samebank vs. Diffbank on Intel Xeon (a); and Samebank vs. Diffbank(s) on Freescale P4080 (b).

Figure 3.9(a) shows the average memory access latency of the Latency benchmark running on Core0. The X-axis shows the number of co-running cores, each of which executes a separate instance of the Latency benchmark. Note that the average access latency is increased significantly in Samebank. This is because all memory requests are serialized in Bank0, i.e. they all result in bank conflicts. In Diffbank, however, the memory requests from Core0 are serviced in parallel with memory requests generated from other cores, because they are accessing a different memory bank. As a result, the instance of the Latency benchmark running at Core0 does not suffer any noticeable performance reduction. This demonstrates the potential of private banking in providing performance isolation.

Having the capability to control memory bank placement using PALLOC allows us to investigate DRAM controller’s characteristics. In particular, we are interested in the queueing structure of the DRAM controller, as it has profound impact on DRAM performance isolation. According to [101], the most common queueing structures are 1) a shared global queue and 2) per-bank queues, but this information is rarely publicly available on COTS components.

In this experiment, we reverse engineer such information using PALLOC. The experiment setup is as follows. Each core executes one Latency instance. Core0 always accesses Bank0. For co-running cores, there are three cases: (1) In Samebank, they access Bank0 (the same as Core0); (2) In Diffbank(B1-7), they access Bank1-7; (3) In Diffbank(B1), they access Bank1.
Figure 3.9(b) shows Core0’s average latency as a function of the number of other cores on the P4080 platform. Similar to Figure 3.9(a), Samebank shows the worst performance. Interestingly, however, Diffbank(B1-7) shows better performance than Diffbank(B1), even though Core0 always accesses its own Bank0 in both cases. From the result, we can infer that P4080’s MC is unlikely to feature per-bank queues, because if each bank had its own queue, then Diffbank(B1) would show the same performance as Diffbank(B1-7). Instead, it is likely to have a single global queue. Then, the better performance of Diffbank(B1-7) can be explained as it can process multiple requests in parallel (faster) using more banks.

We repeat the same experiment on the Intel Xeon platform. Unlike the P4080 platform, we found that the results of Diffbank(B1) and Diffbank(B1-7) are almost identical in the Xeon platform. Therefore, the figure for this experiment looks similar to Figure 3.9(a). This result suggests the Xeon’s MC has per-bank queue and/or support request re-ordering mechanism that favors open banks over closed banks.

In summary, the two platforms have different queueing structures based on our experiments. For P4080, even if we partition DRAM banks, the shared queue is still being contended among cores, which may result in poor performance isolation. On the other hand, Xeon can benefit better from DRAM bank partitioning as contention at the memory controller queue would be eliminated.

In the next experiment, we explore the real-time performance impact of using private DRAM banking in a realistic scenario, modeling a simple real-time data acquisition system. The system is composed of two tasks: a periodic real-time task and a non-real-time task. The real-time task periodically reads a memory region and performs basic processing. The computation must be completed within a given deadline to avoid data corruption, since the hardware can overwrite the buffer with incoming data. The non-real-time task is responsible for analyzing and post-processing the data acquired by the real-time task.

We simulate the scenario on the Intel Xeon platform as follows. First, we create a synthetic real-time task \textit{HRT} that periodically reads a chunk of main memory. Jobs are released with a 20 ms period (50Hz) and have to be finished within a deadline of 13 ms. We schedule the task with the SCHED\_FIFO real-time scheduling policy. The HRT task is engineered to be insensitive to the shared L3 cache, so that all its memory requests result in cache-misses. Second, for the non-real-time data processing task, we use the standard X-window server, which generates memory traffic whenever it updates the screen. To make the X-server update the screen, we keep printing text strings (the standard output of the HRT program) on a gnome-terminal. The X-server runs with the SCHED\_OTHER scheduling policy.

The two desired goals of the system are (1) to meet the deadline of the HRT task and (2) to provide a high frame-rate for the X-server. The former goal is more important than the latter. We use the deadline miss ratio and the execution time distribution to gauge the first goal. We use the CPU utilization of the X-server to measure the latter goal.

We repeat the experiment on four different settings. In both \textit{Buddy(solo)} and \textit{Buddy}, we use the standard buddy allocator, hence utilizing all 16 DRAM banks. In \textit{Samebank}, both HRT and the X-server are assigned to the same eight banks - Bank0-7. In \textit{Diffbank}, HRT uses Bank0-7 while the X-server uses Bank8-15. In
Figure 3.10: HRT runtime distribution on Intel Xeon. Core0-HRT, Core1-Xserver(none for (a))

In Buddy(solo), we only run HRT on Core0, while in the other settings, we also run the X-server on Core1.

Figure 3.10 shows the runtime distribution of HRT. Each figure is plotted using 1000 samples collected over 20 seconds. The X-axis shows the observed execution time (ms) while the Y-axis shows the number of samples observed in the time range. Table 3.4 also shows relevant statistics for the same experiment.

In Buddy(solo), the runtime distribution of HRT is very stable at around 10 ms and the 99 percentile runtime is 10.2 ms. When the X-server is co-scheduled in Buddy, however, the average and the 99 percentile runtimes are increased to 12.2 ms (up by 22%) and 14.3 ms (up by 40%) respectively, resulting in 28% deadline violations. In Samebank, the average and the 99 percentile runtimes are further increased to 13.1 ms and 15.8 ms, respectively, resulting in 45% deadline misses. In Diffbank, however, the average and the 99 percentile runtimes are decreased to 11.5 ms and 12.4 ms, respectively, and only one deadline violation is observed. This result shows that isolating DRAM banks for critical tasks can significantly improve the real-time performance of the system.

We also performed a similar experiment on the P4080 platform. In this experiment, however, instead
Table 3.4: HRT task runtime statistics on Intel Xeon.

<table>
<thead>
<tr>
<th>Conf.</th>
<th>average (ms)</th>
<th>99 pct. (ms)</th>
<th>deadline miss(%)</th>
<th>stdev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buddy (solo)</td>
<td>10.0</td>
<td>10.2</td>
<td>0</td>
<td>0.08</td>
</tr>
<tr>
<td>Buddy</td>
<td>12.2</td>
<td>14.3</td>
<td>27.9</td>
<td>1.05</td>
</tr>
<tr>
<td>Samebank</td>
<td>13.1</td>
<td>15.8</td>
<td>44.5</td>
<td>1.47</td>
</tr>
<tr>
<td>Diffbank</td>
<td>11.5</td>
<td>12.5</td>
<td>0.1</td>
<td>0.56</td>
</tr>
</tbody>
</table>

Table 3.5: HRT task runtime statistics on Freescale P4080.

<table>
<thead>
<tr>
<th>Conf.</th>
<th>average (ms)</th>
<th>99 pct. (ms)</th>
<th>stdev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buddy (solo)</td>
<td>20.1</td>
<td>20.2</td>
<td>0.03</td>
</tr>
<tr>
<td>Buddy</td>
<td>48.7</td>
<td>49.6</td>
<td>0.28</td>
</tr>
<tr>
<td>Samebank</td>
<td>75.1</td>
<td>75.6</td>
<td>0.21</td>
</tr>
<tr>
<td>Diffbank</td>
<td>28.9</td>
<td>29.8</td>
<td>0.09</td>
</tr>
</tbody>
</table>

of running the X-server (as we do not have a working X-server on the P4080 platform), we use Bandwidth benchmark \[160\], which simply accesses a big array without RAW dependency (hence generating large memory traffic). Similar to the previous experiment on Intel Xeon, in Samebank, all benchmarks are assigned to use the same Bank0. In Diffbank, the seven instances of Bandwidth are assigned to all DRAM banks except Bank0. Table 3.5 shows that the Diffbank configuration offers superior real-time performance.

3.3.3 Results with SPEC2006

In this section, we investigate the performance impact of partitioning DRAM banks and caches with the SPEC2006 benchmark suite on the Intel Xeon platform. We investigate both solo performance (single core) and co-run performance (four cores) in the presence of heavy memory contention. We use the Instruction Per Cycle (IPC) as a performance metric. Higher IPC denotes faster execution, as less time is spent waiting for the completion of memory transactions.

Table 3.6 shows the characteristics of SPEC2006 benchmarks used for our evaluation. RSS and IPC denote Resident Set Size and Instruction-Per-Cycles respectively. The benchmarks are sorted in descending order of the average memory bandwidth usage (MB/s). We excluded benchmarks that are either CPU intensive (i.e., bandwidth < 100 MB/s) or allocate too much memory space (i.e., RSS > 750MiB) for the purpose of our evaluation.

**DRAM Bank Partitioning** In this experiment, we investigate the performance impact of DRAM bank partitioning to the single thread performance.

We first introduce a bitmap notation to denote DRAM banks mapped to the partition. As shown in Figure 3.8(a), the Intel Xeon platform has 16 DRAM banks that are addressed by four bits in the physical address: bits 20, 19, 13, and 12. We denote the selected banks by concatenating these four bits in order. For example,
Table 3.6: SPEC2006 characteristics on Intel Xeon

<table>
<thead>
<tr>
<th>benchmark</th>
<th>bandwidth (MB/s)</th>
<th>RSS (MiB)</th>
<th>average IPC</th>
<th>memory intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>470.lbm</td>
<td>3158</td>
<td>409</td>
<td>0.88</td>
<td></td>
</tr>
<tr>
<td>462.libquantum</td>
<td>3124</td>
<td>64</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>2346</td>
<td>123</td>
<td>0.76</td>
<td></td>
</tr>
<tr>
<td>433.milc</td>
<td>2313</td>
<td>523</td>
<td>0.80</td>
<td>High</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>1649</td>
<td>40</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td>450.soplex</td>
<td>1211</td>
<td>108</td>
<td>0.70</td>
<td></td>
</tr>
<tr>
<td>434.zeusmp</td>
<td>1122</td>
<td>502</td>
<td>1.13</td>
<td></td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>798</td>
<td>110</td>
<td>0.63</td>
<td></td>
</tr>
<tr>
<td>436.cactusADM</td>
<td>702</td>
<td>623</td>
<td>0.93</td>
<td></td>
</tr>
<tr>
<td>403.gcc</td>
<td>618</td>
<td>196</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>473.astar</td>
<td>378</td>
<td>325</td>
<td>0.66</td>
<td>Medium</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>203</td>
<td>173</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td>447.dealII</td>
<td>136</td>
<td>6</td>
<td>1.61</td>
<td></td>
</tr>
<tr>
<td>481.wrf</td>
<td>131</td>
<td>570</td>
<td>1.89</td>
<td></td>
</tr>
<tr>
<td>400.perlbench</td>
<td>124</td>
<td>147</td>
<td>1.50</td>
<td></td>
</tr>
</tbody>
</table>

[000X] specifies two DRAM banks in which the address bit 20, 19, and 13 are all zero, and the bit 12 is either 0 or 1. Likewise, [0XXX] indicates eight DRAM banks in which the address bit 20 is zero and all remaining three bits (bit 19, 13, 12) are either 0 or 1.

The experiment setup is as follows. We create a CGROUP partition and assign a subset of DRAM banks to it. We then run each SPEC2006 benchmark in the partition for 10 seconds and measure the average IPC. We repeat the experiment in four different bank assignments. In Buddy, we do not use PALLOD. Hence, all 16 memory banks are utilized by the standard buddy allocator; In 4banks/8banks/16banks, banks are selected by [00XX]/[0XXX]/[XXXX], respectively.

Figure 3.11 shows the normalized IPC of SPEC2006 for each bank assignment scheme. Overall, the number of banks is generally positively correlated with the performance, although not always. This is because the probability of achieving more memory level parallelism (MLP) is higher when more banks are used to allocate the same amount of memory. Because modern out-of-order processors can generate multiple outstanding memory requests at a time, a single threaded program can benefit from the parallelism. For example, 470.lbm’s performance improves by 29% by increasing banks from four to eight. For most benchmarks, however, the performance impact of partitioning DRAM banks is modest: the difference between 4banks and Buddy is 9% on average.

**Cache Partitioning** In this experiment, we investigate the performance impact of cache partitioning, coupled with DRAM bank partitions, to the single thread performance.

In most processors, L2 and L3 are indexed by the physical address. For example, our Xeon processor has 16-way 8 MiB (512 KB/way) shared L3 cache in which the cache-set is determined by using address bits [18:6] of the physical address (L2 cache uses [14:6]). Note that two bits are overlapped with the lower
part of DRAM bank bits [13:12] (see Figure 3.8(a)). This means that selecting bits [13:12] to partition DRAM banks has a side effect of partitioning cache-sets. To investigate the performance impact of such cache-partitioning, we repeat the experiment described in the previous subsection, but in two different bank (and cache) assignments. In PB, we use four DRAM banks, selected by [00XX]. In this setting, each task can use the entire cache; In PB+PC, however, DRAM banks are selected by [XX00] that addresses only 1/4 of the L2&L3.

Figure 3.12 shows the normalized performance of the SPEC2006 benchmarks under PB and PB+PC (normalized to PB). As expected, partitioning cache PB+PC generally negatively affects to the performance, compared to partitioning DRAM banks only (PB). One exception is 462.libquantum, which shows better performance under PB+PC. One reason of this behavior is that the benchmark is completely cache insensitive unless the cache size is bigger than its working set size [102]. Since the 8 MiB L3 cache on our Intel Xeon is smaller than its working set size, cache partitioning does not negatively impact its performance. Overall, the single thread performance of SPEC2006 benchmarks is 18% better, on average, in PB (full cache) than in PB+PC (1/4 cache).

3.3.4 Performance Isolation and Throughput

In this experiment, we investigate the degree of performance isolation achieved by partitioning DRAM banks (and cache) in the presence of memory intensive co-runners.
The basic setup is as follows: We first run the subject SPEC benchmark on Core0 for 10 seconds and measure the average IPC—i.e., Solo IPC. Next, we launch three 470.lbm instances on the other cores (Core 1, 2 and 3) to generate memory traffic, then repeat the experiment to measure the average IPC of SPEC2006 on Core0—i.e., Corun IPC. We repeat the whole experiments in three different memory allocation schemes. In Buddy, we use the standard Linux Buddy allocator; in PB, DRAM banks are equally partitioned to each core so that it can exclusive access 4 banks (out of 16 total banks), but the caches are not partitioned—Core0=[00XX], Core1=[01XX], Core2=[10XX], and Core3=[11XX]; In PB+PC, both DRAM banks and the caches are partitioned to each core—Core0=[XX00], Core1=[XX01], Core2=[XX10], and Core3=[XX11].

To quantify the degree of performance isolation of each memory allocation method, we use the slowdown ratio metric, which is defined as follows.

\[
\text{Slowdown ratio} = \frac{\text{Solo IPC}}{\text{Corun IPC}}
\]  

If the slowdown ratio is high, it indicates poor performance isolation. On the other hand, if the ratio is close to one, it means stronger performance isolation.

Figure 3.13 shows the slowdown ratios of the SPEC2006 benchmarks under the three memory allocation schemes. On average, partitioning DRAM banks and caches considerably improves performance isolation: The average slowdown ratio of Buddy is 3.29, while the slowdown ratios of PB and PB+PC are 2.67 and 2.13, respectively. Note, however, that even in PB+PC where both DRAM banks and caches are partitioned,
the slowdown ratio is far from the ideal unitary value. One possible reason is that the memory data bus is still being shared by all the cores. Therefore, the memory bandwidth can become a bottleneck, especially considering the fact that the co-running benchmark, the 470.lbm, is a highly memory bandwidth intensive one.

Finally, the low slowdown ratios of PB+PC are partly caused by relatively low solo performance compared to PB (see Section 3.3.3 and Figure 3.12). To better understand performance tradeoffs, Figure 3.14 shows co-run performance of the benchmarks. In this figure, although PB+PC provides better performance over PB, the difference is only 4%. Therefore, depending on system requirements, one may favor PB over PB+PC as it has 18% single thread performance advantage on average, while providing a similar degree of co-run performance.

3.3.5 DRAM Bank Partitioning Recommendations for Real-Time Systems

Having shown the potential benefits of DRAM bank-aware allocation in the previous section, we now provide some partitioning recommendations to best utilize this capability and discuss limitations.

Private banking strategy: Assuming you have a set of partitions (or process groups) that are needed to be isolated with each other, a straightforward strategy is to assign different memory bank to each partition, hence we call private banking. This can ensure that no concurrently running partitions on multiple cores to access the same bank at any time and, therefore, provides best performance isolation property. This

Figure 3.13: Slowdown ratios of SPEC2006. Core0=X-axis, Core1-3=470.lbm×3.
strategy is, however, not always feasible because of a number of reasons. First, the number of DRAM banks are limited, typically 8 banks for each DRAM rank. In our P4080 platform, there are total 32 DRAM banks (2 dimms x 2 ranks/dimm x 8 banks/rank = 32). As such only 32 partitions can have a dedicate bank. Second, the size of DRAM bank is also limited. In our platform, each bank is 128MB. Therefore if the applications running on a partition requires more memory space, the partition must be assigned more than 1 bank, further limiting the number of partitions with private banks. Finally, accessing the smaller number of banks also mean less peak memory bandwidth. For example, accessing 8 banks concurrently can achieve 8 times more peak memory bandwidth compared to accessing only one bank. Therefore, the memory bandwidth requirement of each application should be considered in assigning banks.

*Mixed private and shared strategy:* Because of aforementioned reasons—#of banks, space, and bandwidth limitations, some scheduling partitions may share certain DRAM banks. In this case, we can achieve the same level of isolation performance by ensuring the partitions concurrently executing at any given time to use non-disjoint DRAM banks. One potential strategy is to schedule only the partitions that do not share the DRAM banks. Because each partition may have different real-time requirements, this requires careful coordination. Exploring optimal partition scheduling is left as future work.

Figure 3.14: Normalized IPC of SPEC2006. Core0: X-axis, Core1-3: 470.lbm×3.
Apart from de-conflicting DRAM banks, the amount of memory bandwidth (i.e. memory requests per unit of time) that can be extracted from a typical DRAM subsystem is limited. This is typically lower than the traffic that all the cores in the system can request. Unfortunately, the design principles of COTS DRAM controllers value throughput over predictability, as discussed in Section 2.3. As a result, when the processors saturate the bandwidth of the DRAM subsystem, memory requests are queued and can be re-ordered to maximize performance-optimizing heuristics. It follows that the order in which the memory controller satisfies queued requests does not consider task priorities. As such, when saturation occurs, unregulated contention at the level of the memory controller is experienced. The result is that memory requests from critical real-time applications can be significantly delayed due to the activity of other cores.

DRAM bandwidth management to prevent saturation is required. To achieve isolation, an OS level memory bandwidth regulation system, called MemGuard [163] was developed. The goal of MemGuard is to provide bandwidth reservation for each core, so that the stall time each application suffers due to memory bandwidth regulation is predictable and can be analyzed. We were not directly involved in the development of MemGuard and only performed a final integration of this component both in terms of system-level implementation and analysis. Nonetheless, MemGuard represents a crucial component of the final framework. Hence, we hereby introduce the fundamental principles of MemGuard and refer the reader to [163, 164] for a more detailed description.

MemGuard uses a series of per-core regulators that are responsible for monitoring and enforcing the memory bandwidth allocation. This can be done by relying on hardware-specific performance monitoring capabilities. Specifically, each regulator monitors the amount of DRAM transactions performed by each core (or alternatively, the number of last-level cache misses). By considering the worst-case latency \(L_{\text{max}}\) for a single memory request to be serviced, it is possible to derive a worst-case (guaranteed) bandwidth at which the memory subsystem can operate.

MemGuard operates as follows: it is configured to enforce the bandwidth assignment at a given period \(R\). The amount of transactions \(Q_i\) that a given core will be allowed to perform during \(R\), under even bandwidth distribution, will simply be: \(Q_i = \frac{R}{m \cdot L_{\text{max}}}\), where \(m\) is the number of active cores in the system. At the beginning of each period, MemGuard configures the hardware performance counters to trigger an event when any of the cores exceeds the \(Q_i\) threshold of completed DRAM memory requests. To enforce the strict bandwidth assignment, upon reception of a budget-exhausted event, MemGuard idles the associated core. Any idled core resumes its activity at the beginning of the next replenishment period. The length of the regulation period \(R\) is a system-wide parameter and should be much smaller than the minimal application task period. In our current implementation, \(R\) is one millisecond matching also the OS scheduler tick interval.

The key insight is that by restricting the maximal memory bandwidth usage for each core, it is ensured that each core can always complete up to \(Q_i\) memory requests within a regulation period \(R\). Since non
memory-intensive applications typically use less than their reservation, MemGuard also offers different ways to share reserved but unused memory bandwidth across cores to achieve significant average-case performance improvements. Additional details on bandwidth reclaiming and sharing mechanisms can be found in [163].

3.5 Single-Core Equivalence

We have developed and integrated the discussed regulation techniques in a unified framework, namely Single-Core Equivalence (SCE) [8, 1], that has the following main properties: (i) allows real-time tasks to run according to a traditional execution model; (ii) actively and transparently regulates at an OS-level the usage of shared hardware resources; and (iii) relies on existing hardware support available in COTS multi-core platforms.

3.5.1 SCE Methodology for System Deployment

In this section, we briefly describe the main steps that are needed to deploy our SCE framework on top of multi-core platforms to support $m$ concurrently active cores. The key property is that once SCE is in place, the value of WCET($m$) calculated for each task on a given core (see Section 4.2.4) will not be affected by workload changes in other cores. This allows reusing consolidated schedulability techniques for single-core chips on each of the $m$ active cores.

The first step to obtain a SCE-compliant system is to select a commercial multi-core chip that features hardware primitives needed by SCE components. Specifically, the requirements are:

1. An MMU unit, in order to allow page-level coloring;

2. Atomic instructions to prefetch and lock individual lines in last-level cache, or a per-way allocation mechanism;

3. Knowledge (either from technical reference manuals or from benchmarking) about the mapping of physical addresses to DRAM ranks, banks and columns, so that PALLOC can be deployed;

4. Extensive performance monitoring capabilities, allowing per-core DRAM transactions to be observed in software, ultimately allowing MemGuard to operate.

The second step comprises workload characterization. In this step, all but one core of the system are powered off and tasks are studied in isolation. For each task, we extract the complete memory profile and progressive lockdown curve (see Section 3.2.3), together with the maximum number of residual last-level cache misses for each data-point in the curve.

Next, we propose to evenly distribute shared resources to create the pool of exported equivalent single-core machines. The even distribution of resources is not a strict requirement as the techniques comprising SCE
can also be used to perform uneven allocation. Nonetheless, enforcing even partitioning ensures that identical single-core machines are exported, simplifying migration of software components across cores without the need of reiterating a costly task schedulability analysis. The even resource assignment is enforced on the DRAM subsystem by: a) using PALLOC to allocate the same number of private banks to each core; and b) by configuring MemGuard to allocate an equal fraction of the guaranteed bandwidth to each processor. At this point, tasks deployed on different cores will observe a \( m \)-times slower but dedicated DRAM subsystem.

In the next step, the assignment of tasks to cores is performed. The assignment is application-specific, so that the exact task-to-core assignment strategy can be left to the system designer. If IMA partitions are used on top of SCE, partition-to-core assignment is performed in this step.

After tasks/partitions have been allocated to cores, last-level cache assignment is performed. For this purpose, it is possible to rely on the previously derived progressive lockdown curve (see Section 3.2.3): either the desired task WCET is fixed and its cache requirements are derived accordingly; or a given amount of cache is provided and the corresponding WCET is determined. Either way, it must always hold that the cumulative amount of last-level cache (LLC) allocated to all the tasks on the same core (or partition) is less or equal than \( \frac{S}{m} \).

Finally, WCET\((m)\) for tasks assigned on a given core can be derived and schedulability can be checked as described in Section 4.2.4. If the set of tasks on each core is determined to be schedulable, then the whole system is schedulable. Conversely, if any of the tasksets is found not to be schedulable, the previous cache allocation step can be reiterated, rearranging cache resources inside the considered core. Alternatively, the initial task-to-core assignment can be reconsidered and schedulability issues can be solved by migrating groups of tasks across cores.

3.5.2 I/O Core - Peripheral Traffic Serialization

As previously mentioned, in IMA systems real-time applications run within partitions, which represent the basic workload execution environment. However, different IMA partitions may use the same I/O devices and channels. While in a single-core system, accesses to I/O devices on behalf of IMA partitions are inherently serialized, in multi-core platforms device sharing across partitions is a source of unpredictability. In fact, inter-core interference due unregulated I/O peripheral traffic needs to taken into account. In IMA, zero partitions (I/O partitions) are used to handle I/O transactions. Thus, the problem of synchronizing I/O traffic on a multi-core can be reduced to the problem of synchronizing only I/O partitions. As a part of the SCE framework, we propose to assign a dedicated core to aggregated and execute all the I/O partitions of the system. Such core takes the name of I/O core. We have shown in how finding a schedule for the I/O core that respects the timing constraints means deconflicting I/O traffic across cores.

Specifically, finding a valid schedule for the I/O core means observing the precedence relation that exist among: (A) physical data production at the device, transfer to DRAM and beginning of processing; and (B)
end of processing, transfer to DRAM and physical device output. While physical I/O transactions at different devices can be performed in parallel, different instances of I/O partitions on the I/O core cannot overlap. Under this formulation, this I/O scheduling problem can be formulated as a constrained programming (CP) problem. Unfortunately, since this is an NP-hard problem [20], CP does not scale well. As a part of our solution, we developed a heuristic algorithm called hierarchically ordered scheduling (HOS). HOS starts with a random but partial assignment of the offsets for all the physical-I/Os. Next, offsets for processing partitions (at the general purpose cores) are computed, finally determining the offsets of all IMA I/O partitions (at the I/O core). This represents a final solution. The heuristic exploits a first-fit decreasing strategy for bin packing by meeting the hardest constraints first. The hierarchical searching employed in HOS allows to quickly find a solution on average and scales well with the problem size. Full details and implementation considerations can be found in [20].

In our analysis, the effect of PALLOC does not directly appear. It is important to underline that the enforcement of private DRAM banking across cores is fundamental for the soundness of the analysis. In fact, modern DRAM controllers perform reordering of requests at the bank level to maximize open-row hit ratio [2]. Due to reordering, if multiple cores are allowed to access the same bank, several requests from a different core can be processed before a request from the core under analysis, potentially leading to large delays. In multi-core chips this leads to intra-bank interferences if different cores are allowed to access the same set of DRAM banks. Conversely, when PALLOC is used to enforce private banking, the time required for a task to access the DRAM bank in isolation does not change when multiple cores are simultaneously active.

3.5.3 Worst-Case Regulation Stall

In this section, we identify the worst case memory access pattern and derive a bound for the maximum amount of stall that a task can suffer during any regulation period $R$.

Memory regulation (MemGuard) never allows one core to generate more than a given number of DRAM memory accesses ($Q_i$) inside each regulation period ($R$). As long as the core performs $Q_i$ memory requests or less, it is not stalled inside a given regulation period $R$. Conversely, when a core tries to perform the $(Q_i + 1)^{th}$ DRAM access, it is stalled until the beginning of the next period. We define stall as the sum of all the intervals of time during which the core under analysis has a pending memory transaction but is not being served by the memory subsystem. This can happen either because the memory subsystem is serving other cores (contention) or because it has been stalled by MemGuard (regulation).

The first lemma shows that under even bandwidth assignment across cores, the amount of stall induced on the task is upper-bounded by the regulation stall, given that the scheduling policy of the DRAM controller implements a round-robin scheme.

**Lemma 1** Consider a system with even bandwidth regulation, where $Q_i$ is the number of DRAM accesses
of each core during a regulation period of length $R$. If the policy for scheduling DRAM transactions at the bus arbiter and at the DRAM controller is round-robin, then each core is guaranteed to always perform up to $Q_i$ memory transactions within a regulation period $R$.

Proof: This lemma can be easily proven by contradiction. Recall that $Q_i = \frac{R}{mL_{\text{max}}}$. Suppose that the core under analysis takes $X > R$ to perform a burst of $Q_i$ memory accesses that was ready at the beginning of the period. Since the bus and the memory-controller have a round-robin arbitration policy, $X$ can be rewritten in terms of performed transactions:

$$X = Q_i L_{\text{max}} + K_{\text{oth}} L_{\text{max}}$$

Where $K_{\text{oth}}$ denotes the aggregate number of DRAM transactions performed by all the other cores during $R$. It follows that:

$$Q_i L_{\text{max}} + K_{\text{oth}} L_{\text{max}} > R \Rightarrow (Q_i + K_{\text{oth}}) L_{\text{max}} > R$$

$$\Rightarrow Q_i + K_{\text{oth}} > \frac{R}{L_{\text{max}}}$$

$$\Rightarrow Q_i + K_{\text{oth}} > mL_i$$

which contradicts the assumption that each core can perform a burst of $Q_i$ transactions per period $R$ without being regulated.

Next, we find an upper bound on the amount of stall that can be observed in a single regulation period $R$ for the core under analysis. In order to do this, we rely on the fact that at most $Q_i$ memory transactions can be performed by each core during a regulation period $R$ (Lemma 1).

**Lemma 2** For any single regulation period $R$, $R - Q_i L_{\text{min}}$ is an upper bound on the amount of stall suffered by the core under analysis.

Proof: In order to prove this lemma, we consider two cases:

**Case 1 (Regulation)** The task under analysis needs to perform $Q_i$ DRAM memory accesses and no transactions are requested by other cores. In this case, the worst-case stall is suffered when the task under analysis accesses the DRAM at its peak bandwidth. Thus the $Q_i$ accesses will be satisfied in $Q_i L_{\text{min}}$ time units, and the resulting regulation-induced stall will be: $R - Q_i L_{\text{min}}$. This also captures the case in which the memory budget is already exhausted when the task under analysis is scheduled.

**Case 2 (Contention)** The task under analysis needs to perform $Q_i$ DRAM memory accesses together with all the other cores. Since bus and memory controller use a round-robin scheduling policy, in the worst-case
the task will stall for \( Q_i(m - 1)L_{\text{max}} \). Thus, it holds that:

\[
(m - 1)Q_iL_{\text{max}} = mQ_iL_{\text{max}} - Q_iL_{\text{max}} = R - Q_iL_{\text{max}} \leq R - Q_iL_{\text{min}}
\]

The following lemma identifies the worst case DRAM memory access pattern for a given task in the system, within an arbitrary time window. The timing analysis will be then derived according to this scenario. The key insight is that if all memory accesses concentrate on one region, it represents burst arrival case and could possibly take several regulation periods to complete. While the core is stalled due to regulation, no progress on the task can be made. Thus, a clustered memory access pattern represents the worst case in terms of how much delay is introduced on task’s completion.

**Lemma 3** The memory access pattern that causes the worst-case regulation-induced stall on a given task is when all the accesses within the considered task are clustered, starting when the budget is exhausted.

**Proof:** From Lemma 2 it follows that a task cannot suffer more stall than \( R - K_pL_{\text{min}} \) (regulation-induced stall) per each regulation period \( R \). Thus, the worst-case memory access pattern corresponds to the pattern that maximizes this stall. It follows that the worst-case memory access pattern corresponds to the case where all the \( \mu \) memory accesses are clustered at the beginning of the task. If the number of memory accesses \( \mu \) is not a multiple of \( Q_i \), we assume the worst-case contention from all the other cores in the last period in which the leftover transactions occur.

By using Lemma 3 and Lemma 2 we can derive the worst-case regulation-induced stall for a given task that performs \( \mu \) DRAM memory accesses.

**Theorem 1** The worst-case regulation induced stall for a task that performs \( \mu \) DRAM memory-accesses in a system with evenly distributed bandwidth regulation can be obtained as:

\[
stall(\mu) = \left\lceil \frac{\mu}{Q_i} \right\rceil (R - Q_iL_{\text{min}}) + (m - 1)(\mu - (\left\lceil \frac{\mu}{Q_i} \right\rceil - 1)Q_i)L_{\text{max}}
\]  

**Proof:** The theorem follows from Lemma 3 and Lemma 2. The formula can be divided into two terms. The first term captures the regulation-induced delay in all the intervals where regulation occurs. The considered task is regulated every \( Q_i \) transactions, and it can be released right after the MemGuard budget has been exhausted. Thus, it will be regulated throughout its execution exactly \( \left\lceil \frac{\mu}{Q_i} \right\rceil \) times, every time
suffering a worst-case stall of $R - Q_i L_{\text{min}}$ (Lemma 3). The remaining $\mu - \left(\left\lceil \frac{\mu}{Q_i} \right\rceil - 1\right)Q_i$ transactions will not trigger the regulation mechanism and thus will be only subject to contention. Thus, with a round-robin scheduling policy at the bus and memory controller, they will be stalled in the worst-case by $(m - 1)L_{\text{max}}$ each.

For sake of simplicity, we show the derivation of the response time analysis assuming the case $\mu\%Q_i = 0$. In case the assumption does not hold, an upper-bound on the number of requests can be considered: $\hat{\mu} = \left\lceil \frac{\mu}{Q_i} \right\rceil Q_i$. It is easy to see that $\hat{\mu}\%Q_i = 0$ and that $\hat{\mu} \geq \mu$. Under this assumption, we can rewrite the stall term as follows.

\[
stall(\hat{\mu}) = \frac{\hat{\mu}}{Q_i} (R - Q_i L_{\text{min}}) + (m - 1)Q_i L_{\text{max}}
\]

\[
= \frac{\hat{\mu}}{Q_i} (R - RL_{\text{min}} \frac{m L_{\text{max}}}{m L_{\text{max}}}) + (m - 1)Q_i L_{\text{max}}
\]

\[
= \lim_{\hat{\mu} L_{\text{max}}}(R - RL_{\text{min}} \frac{m L_{\text{max}}}{m L_{\text{max}}}) + (m - 1)Q_i L_{\text{max}}
\]

\[
= \hat{\mu} L_{\text{max}} (m - L_{\text{min}} L_{\text{max}}) + (m - 1)Q_i L_{\text{max}}
\]

Since $\hat{\mu} L_{\text{max}}$ is the maximum time spent accessing memory, it is easy to see that under the assumption $L_{\text{max}} = L_{\text{min}}$, the stall term represents the case in which each task sees a memory channel that is $m$ times slower than the isolation case. Moreover, the constant value $(m - 1)Q_i L_{\text{max}}$ represents a blocking term deriving from the fact that the bandwidth assignment is enforced by MemGuard at a finite granularity $R$.

Ideally if the mechanism could be implemented such that $R \to 0 \Rightarrow Q_i \to 0$, this effect would disappear.

3.5.4 WCET($m$) and Response Time Analysis

We consider a level-$i$ busy interval\footnote{Defined as a time interval during which only jobs of task $i$ or belonging to higher priority tasks execute continuously on the processor.}. From Lemma 3, the worst-case memory access pattern can be obtained by clustering together all the memory accesses of task instances in the considered busy interval. Finally, we show that the value of WCET($m$) follows directly from this analysis. For ease of notation, we use the more compact term $C_{\text{see}}$ to indicate WCET($m$).

Equation 3.7 considers a level-$i$ busy interval and computes the response time analysis for tasks with regulation-induced stall as calculated in Equation 3.6.

\[
R_i^{(k+1)} = C_i + \sum_{\tau_j \in hpi(i)} \left[ \frac{R_j^{(k)}}{T_j} \right] C_j + stall(\mu_i^{(k)})
\]

Where $\mu^{(k)}$ captures the clustered memory accesses of all the task instances in the interval and is defined...
as follows:

$$\mu_i^{(k)} = \sum_{\tau_j \in \text{hep}(i)} \left\lfloor \frac{R_i^{(k)}}{T_j} \right\rfloor \hat{\mu}_j \quad (3.8)$$

Note that the stall term in this formulation only depends on the term $\mu_i^{(k)}$ since the amount of regulation-induced stall exclusively depends on the number of required DRAM memory accesses (last-level cache misses).

**Theorem 2** The response time of a periodic task in a system under bandwidth regulation and even bandwidth assignment can be calculated as:

$$R_i^{(k+1)} = C_{sce_i} + \sum_{\tau_j \in \text{hp}(i)} \left\lfloor \frac{R_i^{(k)}}{T_j} \right\rfloor C_{sce_j} + Q_i L_{max}(m - 1) \quad (3.9)$$

Where $C_{sce}$ represents the WCET(m) with m active cores under SCE and is defined as:

$$C_{sce} = C + \hat{\mu} L_{max}(m - \frac{L_{min}}{L_{max}}) \quad (3.10)$$

**Proof:** Equation 3.9 follows from Equation 3.7 after expanding the stall term:

$$R_i^{(k+1)} = C_i + \sum_{\tau_j \in \text{hp}(i)} \left\lfloor \frac{R_i^{(k)}}{T_j} \right\rfloor C_j + \text{stall}(\mu_i^{(k)})$$

$$= C_i + \sum_{\tau_j \in \text{hp}(i)} \left\lfloor \frac{R_i^{(k)}}{T_j} \right\rfloor C_j + Q_i L_{max}(m - 1) + \mu_i^{(k)} L_{max}(m - \frac{L_{min}}{L_{max}})$$

By expanding $\mu_i^{(k)}$ using Equation 3.8

$$= C_i + \sum_{\tau_j \in \text{hp}(i)} \left\lfloor \frac{R_i^{(k)}}{T_j} \right\rfloor C_j + Q_i L_{max}(m - 1) + \left( \sum_{\tau_j \in \text{hep}(i)} C_{sce_j} + Q_i L_{max}(m - \frac{L_{min}}{L_{max}}) \right)$$

Since $\left\lfloor \frac{R_i^{(k)}}{T_i} \right\rfloor = 1$ when checking schedulability for task $\tau_i$:

$$= C_i + \sum_{\tau_j \in \text{hp}(i)} \left\lfloor \frac{R_i^{(k)}}{T_j} \right\rfloor C_j + Q_i L_{max}(m - 1)$$
\[ + (\hat{\mu}_i + \sum_{\tau_j \in h_p(i)} \left[ \frac{R_{i j}}{T_j} \right] \hat{\mu}_j) L_{\text{max}} (m - \frac{L_{\text{min}}}{L_{\text{max}}}) \]

\[ = C_i + \hat{\mu}_i L_{\text{max}} (m - \frac{L_{\text{min}}}{L_{\text{max}}}) \]

\[ + \sum_{\tau_j \in h_p(i)} \left[ \frac{R_{i j}}{T_j} \right] \left( C_j + \hat{\mu}_j L_{\text{max}} (m - \frac{L_{\text{min}}}{L_{\text{max}}}) \right) \]

\[ + Q_i L_{\text{max}} (m - 1) \]

Which can be finally rewritten as:

\[ = C_{sce_i} + \sum_{\tau_j \in h_p(i)} \left[ \frac{R_{i j}}{T_j} \right] C_{sce_j} + Q_i L_{\text{max}} (m - 1) \]

Under this formulation, \( C_{sce} \) can be rewritten as:

\[ C_{sce} = C + \hat{\mu} L_{\text{size}} \left( \frac{m}{B W_{\text{min}}} - \frac{1}{B W_{\text{max}}} \right) \] (3.11)

This formula highlights that the term \( B W_{\text{max}} \) can be overapproximated, or considered as \( B W_{\text{max}} \rightarrow \infty \), in case a precise estimation cannot be performed. Thus, the formula can be rewritten without the dependency from \( B W_{\text{max}} \) as follows:

\[ C_{sce} = C + \hat{\mu} L_{\text{size}} \frac{m}{B W_{\text{min}}} \] (3.12)

### 3.5.5 Experimental Validation

In this section, we present some of the results obtained on a commercial multi-core platform when the SCE framework is deployed and the analysis is carried out as described in Section 4.2.4. Specifically, we show how the workload for a specific task can be characterized and how WCET(\( m \)) can be derived.

We have performed an integrated implementation of Colored Lockdown, MemGuard and PALLOC on a Linux kernel. For our experiments we use a commercial multi-core platform that provides the necessary hardware support to deploy the discussed techniques. Specifically, we have used a Freescale P4080 platform. The P4080 features \( m = 8 \) PowerPC cores with 2 levels of private cache, 2 MB of shared L3 (last-level cache), and 4 GB of DRAM. Each core operates at 1.5 GHz, while the minimum (guaranteed) DRAM bandwidth has been calculated from specifications and validated through benchmarking to be at 1.2 GB/s. Similarly, we have determined a peak bandwidth of 2.5 GB/s. Next, the MemGuard budget replenishment period has been configured as \( R = 1 \ ms \), which represents a good trade-off between regulation granularity and introduced overhead \[163\]. Thereby, under the current configuration, the value of the remaining parameters necessary
Table 3.7: Characterization of SD-VBS benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Profile Pages</th>
<th>Input Res.</th>
<th>PeakVM</th>
<th>Exec. Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>disparity</td>
<td>173</td>
<td>128x96</td>
<td>7736</td>
<td>2.29</td>
</tr>
<tr>
<td>localization</td>
<td>80</td>
<td>128x96</td>
<td>2988</td>
<td>1.61</td>
</tr>
<tr>
<td>mser</td>
<td>115</td>
<td>128x96</td>
<td>3304</td>
<td>2.11</td>
</tr>
<tr>
<td>tracking</td>
<td>217</td>
<td>128x96</td>
<td>3468</td>
<td>1.88</td>
</tr>
<tr>
<td>multi-nct</td>
<td>87</td>
<td>33x44</td>
<td>2996</td>
<td>1.19</td>
</tr>
<tr>
<td>sift</td>
<td>930</td>
<td>128x96</td>
<td>6528</td>
<td>2.04</td>
</tr>
<tr>
<td>texture</td>
<td>404</td>
<td>352x288</td>
<td>4616</td>
<td>2.25</td>
</tr>
</tbody>
</table>

for the WCET analysis are the following: \( L_{\text{size}} = 64 \text{ bytes} \); \( L_{\text{min}} = 2.38 \times 10^{-8} \text{ s} \); \( L_{\text{max}} = 4.96 \times 10^{-8} \text{ s} \); \( Q_i = 2520 \). In addition, the selected platform meets the hardware requirements for the main SCE components described in Section 3.5.1.

In order to perform locking and unlocking of cache lines, we use the `dcbtls` and `dcblc` atomic instructions. In this evaluation, we manage the shared L3 cache, and keep the lower cache levels unavailable for allocation. Thanks to its large size, the L3 allows more flexibility in the allocation strategy. Unfortunately, in the P4080 platform, this level of cache is not significantly faster than the DRAM subsystem, when the latter is used at full bandwidth. This characteristic is evident from the benchmarks, which only experience about a 2x performance improvement when full cache allocation is performed. This means that by managing all the cache levels, significant performance improvements can be obtained without violating SCE invariants. This and similar optimizations, however, are left for future work as we are mostly interested in enforcing performance isolation.

In our integrated implementation, MemGuard directly monitors the DRAM activity in order to take access control decisions at the granularity of a single core. This can be done on the selected platform by relying on the on-chip event processing unit (EPU). This unit is able to internally collect and process Nexus debug messages generated at the DRAM controller. Moreover, the unit can be configured to increment different hardware counters according to the ID of the core that has originated each DRAM transaction. Finally, each counter can be programmed to generate an interrupt when a threshold in the number of collected events is reached.

In our evaluation, time samples have been obtained using the core’s internal timestamp counters in order to have cycle-accurate measurements. On the selected platform, this can be done through the instructions that provide access to the time base register: `mftbu` and `mftb`.

In order to validate our implementation and the SCE theoretical model described in Section 4.2.4, we have used the San Diego Vision Benchmarks Suite (SD-VBS) \[166\]. The suite includes a number of applications that implement image processing algorithms and are good examples of memory-intensive workload. In addition, since the suite includes motion tracking, object localization and image feature detection algorithms, it represents a realistic example of data-centric applications deployed on modern aircrafts for real-time synthetic vision.
For each of these benchmarks, we have performed profiling using the technique described in [6]. Table 3.7 reports a summary of their characteristics, such as: number of memory pages in the produced profile (“Profile Pages”); resolution of input images in pixels (“Input Res.”); peak virtual memory footprint across execution expressed in KB (“PeakVM”); ratio between runtime with no allocated pages and full L3 assignment (“Exec. Ratio”).

We were able to observe the same performance trend across all the considered benchmarks. Thereby we show in the next section the detailed curves for one benchmark in particular, tracking, and summarize the rest in Table 3.8. The tracking benchmark extracts motion information from the input image-set, which involves feature extraction and feature movement detection. Thus, besides the avionic domain, this application is relevant for robotic vision, autonomous vehicles and surveillance. As mentioned in [166], the considered benchmark implements the Kanade Lucas Tomasi (KLT) tracking algorithm.

As reported in Table 3.7, the complete profile for the tracking benchmark is comprised of 217 memory pages, for a total of 868 KB of memory. As mentioned in Section 3.2.3, the correspondent progressive lockdown curve can be obtained by measuring the WCET when an increasing number of profile pages are allocated in cache. Since in the final system the leftover cache space may be assigned to different tasks, once the desired pages are prefetched and locked, we make the rest of the L3 cache unusable for the task by locking arbitrary portions of system memory.

![Progressive Lockdown Curve for Tracking Benchmark](image)

Figure 3.15 depicts the resulting curve and compares the resulting WCET time to the case in which all the L3 is left unmanaged and the benchmark is able to potentially allocate over L3’s entire size. The continuous line represents the measured WCET, while the negative error bars report the difference between maximum and minimum observed runtime. Each data point summarizes 50 different observations. Three main aspects emerge from the plot: a) by allocating about half of the most frequently accessed profile pages, it is possible
to consistently reduce by 75% the WCET of the considered benchmark; b) increasing the amount of allocated pages quickly reduces the fluctuation of the measured execution time; and c) by allocating only a subset of critical pages, the benchmark exhibits performance that are comparable to the case where the entire L3 is available for the application.

Albeit not shown in the plots, the number of residual generated DRAM transactions is associated to each data-point in the progressive lockdown curve. These correspond to the parameter $\mu$ in the analysis and can be experimentally captured or derived from profile-time data. In this case we follow an experimental approach and simply rely on EPU counters to produce the correspondent $\mu$ for each execution. Once the task has been characterized with a progressive lockdown curve, the value of WCET($m$) with $m = 8$ active cores (i.e. $C_{sce}$) can be derived according to Equation 3.10.

![Figure 3.16: $C_{sce}$ calculation and experimental MemGuard runtime for tracking.](image)

In Figure 3.16, the continuous line at the top of the graph represents the obtained value of $C_{sce}$ for each step in the progressive lockdown curve. As a reference, the original progressive lockdown curve (see Figure 3.15) is reported and visible at the bottom of the plot. Finally, the dotted line represents the measured WCET when MemGuard is activated, even bandwidth assignment is enforced and memory-intensive benchmarks are deployed on all the remaining cores. For each data-point, the maximum and the observed fluctuation is reported. In this plot, three main features can be observed.

First, when MemGuard is activated and memory interference from other cores is generated, a component of noise appears in the measurements. This noise results from different components, such as: interleaving of memory transactions from different cores on the bus; and overhead, in terms of DRAM transactions and timing, introduced by the OS routines. The latter are particularly visible since L1 to L3 caches are not available for allocation. In our current setup, we were able to eliminate a portion of this noise by allocating MemGuard periodic routines into the L1 cache. In addition, as a part of our future work, we plan to perform
Table 3.8: Runtime and $C_{sce}$ estimation with fixed cache allocation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Solo</th>
<th>Exp. w/ SCE</th>
<th>$C_{sce}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>disparity</td>
<td>171.1</td>
<td>339.4</td>
<td>598.7</td>
</tr>
<tr>
<td>localization</td>
<td>63.3</td>
<td>95.2</td>
<td>158.5</td>
</tr>
<tr>
<td>mser</td>
<td>15.1</td>
<td>17.3</td>
<td>22.5</td>
</tr>
<tr>
<td>tracking</td>
<td>108.1</td>
<td>212.9</td>
<td>335.9</td>
</tr>
<tr>
<td>multi-ncut</td>
<td>670.1</td>
<td>703.6</td>
<td>761.7</td>
</tr>
<tr>
<td>sift</td>
<td>471.3</td>
<td>640.2</td>
<td>967.5</td>
</tr>
<tr>
<td>texture</td>
<td>440.1</td>
<td>893.5</td>
<td>1465.6</td>
</tr>
</tbody>
</table>

an extensive analysis to identify the set of critical routines/data structures of the OS that need to be retained in cache to reduce the DRAM noise.

Second, despite the noise, it can be observed that by combining MemGuard, Colored Lockdown and PALLOC it is possible to enforce strict resource allocation to prevent inter-core interference with a reasonable loss in performance. In fact, after about 140 allocated pages, we observe that some of the collected samples present a runtime that is very close to what observed in isolation. On the other hand, when not enough critical pages are allocated in cache, a significant worst-case slowdown is experienced. This effect is expected since an $m = 8$ times slower DRAM subsystem is exported by SCE.

Third, it is easy to note that the estimated $C_{sce}$, calculated according to the equations in Section 4.2.4, always upper-bounds the experimentally produced WCET. This property confirms the validity of the proposed model and analysis. As previously mentioned, the same result has been obtained on all the benchmarks of the suite. Due to space constraints, the results are summarized in Table 3.8. In this table, we have fixed the allocation at half the number of profile pages and report the measured WCET in isolation (“Solo”); measured WCET under SCE while memory-intensive tasks are active on different cores (“Exp. WCET w/ SCE”); and calculated value of $C_{sce}$. All the times are in milliseconds.

In the plot a certain degree of pessimism is visible. This is not surprising since at least two main sources of pessimism can be identified: a) in order to be conservative, the analysis assumes the memory access pattern that realizes the worst-case regulation-induced stall (Lemma 3). However, this does not adhere to real benchmarks where non-memory instructions and memory accesses are not clustered, but rather interleaved; b) the additional noise in the system increases the number of DRAM transactions that the task is being accounted for. The latter effect can be alleviated by performing aforementioned OS-level optimizations. Moreover, additional knowledge of task-specific memory access patterns could be derived using static analysis tools and leveraged to lower the current pessimism.

Finally, we have mentioned how the simpler Equation 3.12 can be considered for practical purposes instead of Equation 3.11 if an exact estimation of the parameter $L_{min}$ (i.e. $BW_{max}$) cannot be performed. Thereby, in our last plot we show how sensitive is the derived $C_{sce}$ to variations of the discussed parameter. For this purpose, Figure 3.17 compares the $C_{sce}$ curve reported in Figure 3.16 with the same calculation performed under the unrealistic assumption that $BW_{max} = 1000$ GB/s (dotted curve). Despite the large gap in the...
Figure 3.17: Sensitivity of \( C_{sce} \) to variations in \( L_{min} \) (i.e. \( BW_{max} \)) parameter.

value of this parameter, however, only a 7% increase is visible in the resulting curve.

3.5.6 Complete SCE Example

In this section, we provide an integrated, qualitative example of SCE. The goal is to show, using a simplified workload, how the SCE methodology described in Section 3.5.1 can be applied. We assume that steps 1 and 2 have been performed, in order to skip the details about hardware selection, task-to-partition and partition-to-core assignments.

Consider task \( \tau_1 \) with period 36. When analyzed in isolation:

After the allocation of 3 pages in cache for task \( \tau_1 \):

Figure 3.18: Progressive Lockdown Curve for Task \( \tau_1 \) and resulting WCET(1) for different amount of memory pages allocated in last-level cache.
Figure 3.18(a) shows that after the profiling for Task $\tau_1$ is completed, the lockdown curve can be used to select the desired runtime in isolation for a given task. In our example, Task $\tau_1$ has a period of 36 time units and its execution time is 13 time units when no pages are locked. The same curve needs to be derived for all the tasks in the system.

Next, a cache assignment for the three most frequently accessed memory pages is performed for this task, as depicted in Figure 3.18(b). This determines a reduction in the execution time of the task from 13 to 7 time units when executed in isolation. However, as shown in Figure 3.19(a), when the interfering cores are turned on, unregulated contention on main memory is generated. As a result, memory accesses performed by Task $\tau_1$ to all those locations that were not allocated in cache cause a fluctuation on the execution time of the task under analysis. The execution variance can be severe and represents a major source of pessimism when computing the worst case execution time of tasks running on multicore. MemGuard is used to enforce a predictable regulation of core accesses to main memory. Specifically, we enforce an even memory bandwidth partitioning across the cores of the system (see Section 3.5.1) to obtain safe and tight bounds on task execution times. Since MemGuard performs an even distribution of the available bandwidth, each core will see a slower (but predictable) memory subsystem. The inflated execution time ($C_{sce}$) for each task can be computed as described in Section 3.5.4. In our example, Task $\tau_1$ will have an inflated worst-case execution time of 8 time units, as depicted in Figure 3.19(b).

This step completes the procedure to derive the final parameters for a given task. In our example, Task $\tau_1$ will have a period of 36 time units, with a WCET($m$) (i.e. $C_{sce}$) of 8 time units and 3 memory pages allocated in last level cache. A complete view of the workload deployed on one of the CPUs of the system under analysis is presented in Figure 3.20(a). Specifically, two IMA partitions (Partition 1 and Partition 2) are active on CPU 1, with a period of 18 and 36 time units respectively, and a reservation of 6 and 12 time units respectively. Inside Partition 1, two tasks are running: Task $\tau_3$ with period 18 and execution...
time 3; and Task $\tau_4$ with period 36 and execution time 3. Similarly, Task $\tau_1$ (analyzed in Figures 3.19) and Task $\tau_2$ are running inside Partition 2. Such tasks have periods 36 and 72, respectively, and have an execution time of 8 and 3 time units, respectively.

Inside each IMA partition, tasks are scheduled rate-monotonically and at each partition activation, prefetch and lock (Colored Lockdown) of the allocated cache pages is performed for all the tasks running inside the partition. The prefetch and lock operations are highlighted in the figure using a striped pattern. Figure 3.20(b) shows the top-level scheduling of IMA partitions on CPU 1 and CPU 2 in the considered system. Once tasks are assigned to partitions and partitions are assigned to CPUs (see Section 3.5.1), the schedule of IMA partitions can be done offline (cyclic executive). Since each partition is synchronized with one or more I/O devices, each instance of a partition will be activated at a constant offset from the beginning of its period. Moreover, since prefetch and lock of allocated memory pages is performed at the beginning of each partition instance, IMA partitions execute non-preemptively on the assigned CPU. The final schedule of IMA partitions is determined using a methodology similar to what described in [20].

Adopting the methodology in [20], conflicts among different cores accessing the same physical I/O devices are resolved offline by serializing access to all the devices on a special core, called I/O Core. For each IMA partition, say Partition 1 on a certain core, there are two events that occur before a Partition 1’s instance begins executing on the CPU. First, if the tasks in Partition 1 need data from a certain device, say Device A,
then Device A will have to internally produce the data. This is called physical input. Second, the I/O Core will interface with Device A and make the data available to Partition 1. This is called input partition. Apart from completing by the beginning of any Partition 1’s instance, input partitions cannot begin earlier than physical inputs for the same device. Two similar events occur at output, namely physical output and output partition. Clearly, output partition has to begin after the completion of Partition 1’s instance and before the beginning of the physical output for the same device. Note that since physical inputs and outputs are enforced by the devices, they have a constant offset within their periods.

I/O requirements for system partitions are considered and a table of I/O transactions performed on the I/O-Core is derived. Figure 3.21 shows a possible schedule for these transactions, assuming that both input and output of data from/to peripherals can be completed at most within one unit of time. In the figure, physical inputs (outputs) are denoted as ‘I” (“O”) on the last row; input (output) partitions are denoted as ‘I” (“O”) on the third row. As depicted, following precedence constraints need to be honored by each partition and its related I/O peripherals: (i) data transfers can be performed on the I/O-core only between the end (beginning) of the physical input (output) of data at the device; and (ii) the activation (completion) of the next (previous) partition instance.

3.6 Verification of Colored Lockdown

Real-time hardware resource management aims at mitigating inter-core performance interference, but as we have extensively discussed, the proposed solutions often involve OS-level modifications. A part from what
presented in this chapter, a number of other works [123, 120] have proposed OS-level mechanisms to explicitly manage those hardware components that, if unregulated, represent major sources of unpredictability: i.e. shared CPU caches, DRAM memory, and I/O subsystem. Management techniques proposed in the literature have been shown to achieve substantial real-time benefits. Yet, industry is reluctant to widely adopt such solutions due to a fundamental lack of confidence about the correctness of their implementation. The fear is justified considering that hardware management mechanisms often operate at high-privilege level, and thus their misbehavior can lead to substantial failures.

The work presented in this section represents a first step toward the verification of system-level components that implement hardware management techniques for real-time purposes. In fact, we demonstrate that it is possible to verify the logic of a kernel-level component at the source code level in a modular way; i.e. without verifying the entire OS that can be assumed verified or trusted. Specifically, we hereby present a verification approach for Colored Lockdown [6] presented in Section 3.2 which is part of our SCE framework (see Section 3.5).

As increasingly higher level of assurance is required from safety-critical systems, there has been an uptrend in the popularity of verification methodologies. A consistent body of works has used the “verified by design” approach. In this context, the SPARK language and toolkit [167] provide extensive capabilities to reason about the correctness of applications at a source code level. In the SPARK environment, verification is performed with a combination of static analysis and deductive verification. Deductive verification on the other hand, has been widely used on industrial use-cases [168, 169, 170]. Similarly, the level of assurance provided by formal static analysis based on abstract interpretation often represents a good trade-off in terms of scalability [171, 172].

Automated assertion checking is often used as an alternative to deductive verification. With this approach, it is typically possible to confine the explored state space to a manageable subset that is fundamental for the considered properties/assertions. Among the different techniques for assertion checking, bounded verification is often used for source code debugging. A number of consolidated tools implement assertion checking, e.g. SLAM [173], TASS [174], and CBMC [175] used in this work.

Recent works have explored the use of verification techniques to validate application-level software in the domain of control systems [176], aerospace and avionic software [177], and railways systems [179]. In seL4 [180], the design and verification of an entire OS is proposed. While closely related to [180], we take a fundamentally different approach: we consider certified systems where new kernel-level functionality can be introduced to improve/optimize performance and demonstrate how modular verification of OS-level code can be performed. Finally, many works perform verification of the interaction between kernel modules and OS routines [181, 182]. Conversely, we focus on the verification of kernel-level logic that interacts with (i) kernel sub-routines, (ii) virtual memory, and (iii) CPU cache space.
3.6.1 Verification Approach

This section provides an overview of the approach followed to verify the main properties of Colored Lockdown. We first establish the boundaries of the performed verification; next, we discuss what memory model is being considered; and finally we describe what components of the hardware/OS are abstracted.

Verification Strategy We perform source-level verification of the main block of code that is responsible for the allocation of memory pages in last-level cache within the Colored Lockdown module. The considered code is compiled as a Linux kernel module and runs at the highest level of privilege in the target platform. Verifying its correctness is therefore of great value.

In order to perform cache allocation, the Colored Lockdown module tightly interacts with the rest of the Linux kernel in two main ways: (i) it uses data from many descriptors used in the kernel; (ii) it invokes memory manipulation/translation procedures provided by the Linux kernel. The code base of the entire Linux kernel is too large and complex to be formally verified. For this reason, we restrict the verification to that portion of the cache allocation logic that is directly related to Colored Lockdown.

In order to focus the verification on the important components, we abstract the behavior of any invoked kernel routine, as detailed in Section 3.6.2 For instance, a routine used to allocate a new generic memory page is abstracted as a function that returns an unsigned integer. The return value is non-deterministic, and such that: (i) it is aligned to the memory page size; and (ii) it is within the range defined by the bit-width of the considered memory layout.

Similarly, only sub-fields of kernel data structures that are relevant to verification are initialized by the verification routines. A portion of the initialization procedure is parameter-dependent, so that different cache allocation scenarios can be analyzed.

Verification Boundaries and Assumptions The hardware-level properties that are abstracted mostly concern the behavior of a typical cache controller that allows per-line cache locking. The assumptions required to verify Colored Lockdown are a subset of what discussed in Section 3.1 First, we assume that the initial status of the cache is unknown. This reflects the status of a cold cache at the time of Colored Lockdown allocation. Second, we assume that the considered cache is physically indexed Third, we consider that the bits of the physical address that encode the index of a cache line correspond to the least significant bits following the cache offset bits. Hence, the structure of a physical address from the cache controller’s perspective from most to least significant bit is: tag bits, index bits, offset bits (see Figure 2.3(b)). In this case, we consider cache controllers that support per-line locking hence we assume that a special instruction is available to set a lock bit on a per-line basis. Once the lock bit has been set, the cache line cannot be evicted from cache. Finally, we assume that a cache look-up for a locked line will result in a cache hit.

\footnote{Shared last-level caches in multi-core platforms are typically physically tagged and indexed.}

\footnote{The Colored Lockdown implementation that has undergone verification is the one used for integration within SCE and it targets PowerPC platforms that support cache locking by-line, like the Freescale/NXP P4080 used in our evaluations.}

82
We verify an implementation of Colored Lockdown as a Linux kernel module. The same logic, however, can be ported across different OS’s, assuming that they provide kernel-level routines with similar semantics. In order to focus our attention on the target module, we assume that the descriptors belonging to the OS and used by Colored Lockdown have been correctly initialized (see Section 3.6.2). Next, we assume that profile information about the process under consideration have been correctly passed from user-space to kernel-space. Finally, we assume that all the virtual memory pages of the process have a valid mapping in physical memory. The latter assumption is typically verified in RTOS’s that do not perform demand-paging. Under Linux, this behavior can be achieved using the `mlockall` system call.

**Memory Layout Specification** Our verification is parametric with respect to the memory layout and cache controller configuration. Thus, it is possible to re-run the verification procedure on a specific memory/cache configuration and with a variable number of pages to be allocated, i.e. profile pages. The following five parameters suffice to fully define the considered memory subsystem as well as the address structure from the cache controller’s perspective:

1. $P_s$: Number of bits in a virtual address that encode the offset of a byte in a memory page, also known as *page shift*;
2. $B_w$: Bit-width of a physical address in the considered platform, e.g. 32 for 32-bit architectures; 48 for 64-bit architectures$^7$;
3. $O$: Number of bits in a physical address that encode the offset of a byte within a cache line;
4. $I$: Number of bits in a physical address that encode the index in cache of a cache line;
5. $W$: Associativity – i.e. number of ways of the cache.

Given the five parameters above, the rest of the parameters used to perform cache locking can be derived: size of a memory page; size of a single cache line; number of lines and pages (i.e., available colors) per way; number of cache sets; bit-width of the cache tag; and total size of the cache.

One more parameter controls the amount of memory that is allocated in cache for the process under consideration. This parameter defines a generic number of pages that is prefetched and locked in cache as a result of the coloring/locking logic. By default, all the pages are considered as process’ heap pages. This however does not affect the generality of our approach since there is no difference in the way pages belonging to various regions are handled.

**Verified Properties** A set of core properties of Colored Lockdown was successfully verified. The target of the verification is twofold: (i) that cache allocation is correctly performed when profiling data are correctly

---

$^7$Despite the bit-width of CPU registers is 64 bit, the memory subsystem typically works with 48 bit addresses. This results in 256 TB of addressable memory and a 4-level page tables layout is used.
specified from user-space, and the amount of memory to be locked in cache is smaller than the cache size; and (ii) that the status of the system and cache is overall consistent. Note that using the current verification infrastructure, additional system/cache properties can be verified. The verified properties can be summarized as follows:

1. If the number of pages to be allocated in cache is less or equal to the number of available cache space in pages, cache allocation for the considered process is entirely performed. Otherwise, no cache allocation is performed;

2. If cache allocation is performed, then all the physical memory mapped to each virtual address within the range selected for allocation will result locked in cache;

3. No more than the total number of locked pages are set as locked in cache at the end of the Colored Lockdown procedure;

4. All the temporary kernel-level resources required by Colored Lockdown to execute are released at the end of the procedure.

Verification Challenges Hereby we summarize the challenges that had to be addressed to perform source-level verification of Colored Lockdown as a OS-level component. One of the first challenges we encountered in the attempt to verify a Linux kernel module was the large number of dependencies with the kernel source code that a module can exhibit. Three main type of dependencies exist: data type dependencies, procedural dependencies, and logic dependencies.

A Linux kernel module uses several types that are defined and exported by the kernel. Many of these types are complex C-language structures interconnected via pointers. Obviously, only a subset of the fields in such structures are required for focused verification. cbmc v. 5.2 [175], the source code verification tool we used, employs slicing to eliminate unused variables and reduce verification complexity. However, we found this to be inadequate for our target system. The first challenge was to manually prune the definitions of kernel-level structures to exclude all the irrelevant fields. In order to overcome this issue, we have incrementally transferred into the verification sandbox a number of kernel headers and systematically stripped them of unneeded data types and fields. For instance, one of the imported files was sched.h that in the Linux kernel defines constants and types relevant for process management. The file is about 2700 lines long in a typical Linux source tree. In the first pruning, we only maintained the process descriptor definition, reducing the file length to about 370 lines. Next, we identified the only two fields required for verification out of the 170+ fields included in a typical process descriptor.

The second type of dependency is procedural dependency. The code that needs to be verified uses at top level a set of routines defined in the kernel code. To reduce the state space and the amount of code logic to be verified, one challenge consists in abstracting the semantics of the invoked procedures (if possible) and
making a reasonable assumption on their output. In Section 3.6.2 we describe as an example the abstraction performed on the kernel procedure `get_user_pages`.

Finally, many logic dependencies exist between the state of the kernel and the verified module. This problem sets our verification approach apart from verification of standalone components. In fact, the Colored Lockdown module expects the status of a number of kernel-level descriptors to be initialized and valid. Some of these descriptors are created at boot-time, while others are constantly updated upon system events. Hence, it would be unfeasible to verify the code responsible for their initialization. To tackle this challenge, we have first identified all the logic dependencies. Next, we have introduced an initialization routine that either explicitly sets each referenced variable to its expected value or assumes its value to be within the expected range. A closer look at the initialization procedure is provided in Section 3.6.2.

Overall, CBMC revealed a good maturity in handling C source code. However, when verifying kernel-level code, we have encountered a few glitches that need to be carefully addressed to avoid false negatives in the verification process. Relatively simple workarounds have been found for all the encountered glitches. Such problems, however, can represent a serious overhead in the verification process when reasoning over a large base of system-level code.

The first problem we encountered regards the way `void` pointers are handled in CBMC. The standard C semantics enforces that the increment of a `void *` data type is performed at the granularity of a single byte. Consider the following code:

```c
int void_test(void)
{
    void * ptr = (void *)(1 << 12);
    ptr += 0x100;
    return (ptr == (void *)0x00001100UL);
}
```

The code compiles without warnings/errors under a standard GCC compiler. The expected return of the `test` function is always 1 under standard C-pointer arithmetic. However, a CBMC verification instance that relies on this behavior will fail. Running CBMC 5.2 on the considered procedure, produces the following output:

Counterexample:

```
State 21 file ./cbmc_test.c line 18 function void_test thread 0
-------------------------------
ptr=NULL (00000000000000000000000000000000)

State 22 file ./cbmc_test.c line 18 function void_test thread 0
-------------------------------
ptr=NULL + 4096 (00000000000000000001000000000000)

State 23 file ./cbmc_test.c line 19 function void_test thread 0
-------------------------------
ptr=NULL + 3840 (00000000000000000000111100000000)
```

85
Clearly, State 23, which should reflect the pointer’s status after the increment in the considered code extract, reports a wrong pointer value. This triggers a verification failure. A possible workaround consists in performing the pointer value increment after a conversion to unsigned long\(^8\).

The second issue requires a longer explanation but can be summarized as follows. CBMC seems to exhibit a glitch in the propagation of a variable value after it has been assigned using a bitwise operator. Consider the following snippet:

```c
int retval = 1;
for (...) {
    retval &= bool_function(...);
}
if (retval) ...
```

In this case, CBMC produced verification counterexamples that reported the execution of the if block even though the state value of the `retval` variable was 0 (false);

In our verification, we found that many subtle interactions in system-level code are hard to fully capture at a source level. Consider the following case. Colored Lockdown performs re-coloring of a process page. For this purpose, a new physical page is allocated and its content copied from the original page, appropriately modifying the page tables of the process under consideration. This behavior is “correct” as far as Colored Lockdown is concerned. However, if no action is taken to de-allocate the original page correctly, Colored Lockdown can indirectly trigger a fault somewhere else in the system as the original page descriptor remains in an inconsistent state. Similar interplay problems can occur when a module accesses a data structure without acquiring the required lock. In a typical multi-threaded application, this problem would be easy to detect since all the execution flows are known. The problem is however significantly harder to solve without knowing where in the kernel potential data races can arise.

Finally, a challenge that affects source-level verification at large, is the quick increase in complexity as the state-space expands. In our verification attempt, we were able to overcome the vast majority of challenges described in this section. In spite of this, verification settings with realistic parameters required significant computational resources. We provide additional insights on the feasibility and limits of our verification approach in Section 3.6.2.

---

\(^8\)The unsigned long type has typically the same bit-width of a pointer in the considered architecture.
3.6.2 Verification Details

In this section, we provide additional details about the performed verification. First, we discuss how the cache hardware is modeled; next, we discuss the initialization of kernel structures and OS state. A detailed overview about how cache and memory layout are initialized is also provided. Finally, we detail the structure and verification statements used to verify the core properties of Colored Lockdown.

**Cache Model** Traditional source-level verification tools, including CBMC, do not provide primitives to model platform hardware behavior. For this reason, we use a supporting data structure to maintain the cache state and to perform assertions on its state. Colored Lockdown allows deterministic allocation of memory pages in cache. Thanks to coloring, the mapping set is explicitly controlled. Conversely, the decision about the allocation way is left to the cache controller. The key insight, however, is that when the replacement policy attempts to allocate a line with a certain set, and the line for that set is marked as locked in a given cache way, the way cannot be selected for eviction. Thus, as long as a number of lines less or equal to the cache associativity is locked, each locking request can be satisfied. It follows that the logical view of a cache is a 2D structure (sets vs. ways). One index (set index) is derived from the physical address being allocated; while the other index (way index) is non-deterministically determined by the replacement policy.

Following this structure, the cache status is defined as:

```c
typedef struct {
    void * addr;
    char locked;
} cache_line_t;

typedef cache_line_t cache_set_t [CACHE_ASSOC];
typedef cache_set_t cache_t [CACHE_NSETS];

cache_t cache;
```

In the listing above, `CACHE_ASSOC` and `CACHE_NSETS` refer to the number of sets and to the number of ways (associativity), respectively. Note that there is no need to record the value of the cached data, as we are only concerned with hit/miss behavior. Hence, only cached address and locked status are being tracked.

The assumption we make about the initial state of the cache is that no line is currently locked. As such, we initialize the locked state on all the cache elements as 0, and assign a non-deterministic value to the address field.

**Profile Structure and Initialization** As stated in Section 3.6.1, we assume that profiling information has been passed from user-space to kernel-space before the lockdown procedure is invoked. Hence, for verification purposes, we explicitly initialize the kernel structures that hold kernel-side profile data. In Colored Lockdown, profiling data is provided via the Linux CGROUP virtual file-system interface. For a task for which a profile has been loaded via the CGROUP interface, a custom structure, namely `struct`
**Task Profile**

The most relevant fields of the structure are: (i) number of memory regions with pages to be locked; (ii) list of descriptors for memory regions with data to be locked; (iii) total number of pages to be allocated in cache; (iv) list of descriptors for pages to be locked.

Since the state of the `struct task_profile` object is assumed to be valid, an initialization routine was added. The routine allocates enough data to contain the full list of memory regions and memory pages. These parameters are set at profile loading time, hence they are known at the time Colored Lockdown is invoked. In our approach, they are treated as parameters for the creation of a verification instance. A default scheme is used to associate memory pages to areas. This choice however does not compromise the generality of the verification, as there is no different in the way pages in different areas are handled.

Within each memory region’s descriptor, only the index that the considered region has in the list of kernel-maintained virtual memory areas (VMA) is initialized. The logic that resolves such a (relative) index into an absolute range of virtual memory addresses is part of the Colored Lockdown logic. Hence, it is part of the verification.

**Task Descriptor Setup**

When Colored Lockdown is invoked as a system call by a task, it heavily relies on information contained within the kernel-maintained task descriptor `struct task_struct` to perform cache allocation. Whenever any system call is invoked in the kernel, a globally visible expression, namely `current`, expands to a pointer to the `struct task_struct` object for the calling process. For verification purposes, the object pointed by `current` needs to be initialized. The following is an extract of the task descriptor setup routine:

```c
int pages;
struct vm_area_struct * prev_vma;
struct vm_area_struct * cur_vma;
/* ... */
prev_vma->vm_start = 0x08048000UL;
current->mm->mmap = prev_vma;
pages = nd_int();
__CPROVER_assume (pages >= AREA_MINPAGES && pages <= AREA_MAXPAGES);
prev_vma->vm_end = prev_vma->vm_start + (pages << PAGE_SHIFT);
/* Link VMAs */
cur_vma->vm_start = prev_vma->vm_end;
prev_vma->vm_next = cur_vma;
/* Use cur_vma to setup next VMA */
```

The first area in the list of VMAs is typically the text (i.e. the executable code) section of a process. The start of the first area is taken as the default address at which code is logically placed in compiled executables (line 5). The address of the first VMA descriptor is recorded inside the `current` object (line 6). Next, a non-deterministic number of pages between the established boundaries is generated in lines 7–8, and the end of the first VMA is set accordingly (line 9). As VMAs are initialized, they are placed in an unidirectional linked list (lines 11–12).
**Colored Lockdown Procedure**  The Colored Lockdown module also performs a series of initialization routines as soon as it is loaded (once) into the kernel. The routines mostly initialize cache parameters and buffers required to perform page coloring. Due to space constraints, we omit the details about how initialization is performed inside the verification environment.

When Colored Lockdown core logic is invoked as a system call, the sequence of operations can be summarized as follows:

1. Access to profile structure and validation of `current` object – to make sure Colored Lockdown is performed on the right task;
2. Derivation of virtual addresses for each memory page in the profile to be allocated in cache;
3. Resolution of virtual addresses into physical addresses and cache color calculation;
4. Check of color availability in cache and assignment of first available color;
5. If each page has been assigned a color, perform page re-coloring (as needed) and lockdown.

Hereby, we highlight those portions of listing that are relevant to understand the interaction with CBMC. The first point is trivially verified because we assume that profile data passing and Colored Lockdown invocation is performed correctly. The second step largely uses data in the `current` descriptor initialized as described above. Next, in order to translate the virtual addresses of pages to be allocated, Colored Lockdown uses a kernel routine, namely `get_user_pages`. The `get_user_pages` routine represents an entry point for a number of page-wide kernel operations that can be selected via a `flag` parameter. When invoked with no flags, the function takes as input a range of (virtual) addresses and a task descriptor and returns an array of pointers to `page descriptors`. Each page descriptor corresponds to a page in the selected range. In Linux, the value of the pointer to a page descriptor is always a linear translation of the described page’s physical address. Hence, knowing the pointer to the page descriptor for a page is equivalent to knowing its physical address. The `get_user_pages` logic is fairly complex, but since it is part of the kernel, it sits beyond our verification boundaries. As such, we have abstracted much of its functionality as follows.

```c
long get_user_pages(struct task_struct *tsk, struct mm_struct *mm, unsigned long start, unsigned long nr_pages, int write, int force, struct page **pages, int *locked)
{
    struct page *page_ptr;
    assert(nr_pages == 1);
    assert(write == 0);
    assert(force == 0);
    assert(tsk == current);
    assert(mm == tsk->mm);
    page_ptr = __CPROVER_uninterpreted_void_ptr(tsk, mm, start);
    __CPROVER_assume(page_ptr >= mem_map && page_ptr < (mem_map + MAX_PAGES));
    __CPROVER_assume(((unsigned long)page_ptr & ((1 << sizeof(struct page)) - 1)) == 0);
    *pages = page_ptr;
```
First, a set of asserts on the passed parameters is performed (lines 4-8), to verify the expected value of a number of parameters when `get_user_pages` is called within Colored Lockdown. An uninterpreted function is used (line 9) to construct a valid return value for the routine. In general, the returned value can be any pointer to a `page_struct` object (line 11) with a value between `mem_map` and the end of that portion of kernel memory where page descriptors are stored (line 10). For any specified parameter value of `tsk`, `mm` and `start`, the same page pointer should be returned by successive invocations of `get_user_pages`. Hence the use of an uninterpreted function at line 9. The derivation of physical addresses from page descriptor pointers follows a similar logic.

In the following step the availability of colors is checked. The check is performed using an internal structure that remembers the color associated to each page to be allocated. The step is performed with minimal kernel interaction. When a “conflict” page is encountered, i.e. a page with an unavailable color, the module selects the closest available color. It also marks the internal descriptor for the page to reflect the change. At this stage, no recoloring is performed, hence no final changes are carried out.

If the procedure has determined that there exist enough available space to perform cache allocation, the following actions are performed. First, the module performs re-coloring of all the conflict pages. Second, it executes a cache lockdown operation on each line of each profile page. In the considered architecture, the lockdown is performed using a dedicated assembly instruction, namely `DCBTL`\(^{10}\). In order to perform verification, however, we also update the status of the structure used to model the cache. More in detail, we invoke the `lock_line` procedure on each address corresponding to every line in a page being allocated. The `lock_line` procedure is reported below.

```c
void lock_line(void * addr)
{
    unsigned int index = get_index(addr);
    unsigned int way = nd_int();
    __CPROVER_assume(way >= 0 && way < ACHE_ASSOC);
    __CPROVER_assume(! cache[index][way].locked);
    cache[index][way].addr = addr;
    cache[index][way].locked = 1;
}
```

The procedure is invoked on physical addresses, hence it is easy to calculate the cache index of the line, i.e. the cache set where the line will map (line 3). Since no specific cache replacement policy is assumed, the way selected for the allocation is generated as a non-deterministic integer (`nd_int()`; line 4) between 0 and the number of available ways (line 5). The ways where a line has been previously locked in the same set are excluded (line 6), as per assumed hardware behavior. Finally, with selected set/way, line locking is carried out as in lines 7 and 8.

\(^{9}\)In a Linux kernel, this symbol represents the beginning of the array of page descriptors.

\(^{10}\)This instruction is common to PowerPC-based platforms, such as Freescale MCPxxx and QorIQ P40xx platforms.
To complete the verification, after Colored Lockdown is invoked, we check that: (i) every physical address (at the granularity of single cache lines) in pages to be allocated, as per the profile, can be found in our cache structure; and that (ii) no more locked lines that what calculated from the profile is marked as locked.

3.6.3 Evaluation

In this section, we provide a brief evaluation of the time required to perform verification using the proposed approach. The evaluation has been performed under two memory/cache layout scenarios using CBMC version 5.2 on a workstation machine featuring a 28-core Intel Xeon E5-2658 CPU running at 2.10 GHz with 32 GB of RAM. Unfortunately, CBMC only uses only one core and it is not possible to parallelize the verification effort due to the large amount of memory required to acquire each sample.

In the first scenario, we consider a 32-bit system ($B_w = 32$) with the following memory layout: memory pages of size 256 bytes ($P_s = 8$); a cache line size of 64 byte ($O = 6$); and a way size of 512 bytes ($I = 3$), so that each cache way can entirely hold 2 memory pages. We study the length of the verification for an increasing number of profile pages and cache associativity. Moreover, we set the timeout for the verification to 2 hours. The results for this setup are reported in Figure 3.22.

In the figure, we use logarithmic scale to visualize in a compact way the runtime of the considered scenarios. As can be seen, the verification runtime can require from few milliseconds to entire hours, depending on the complexity of the system. For a low number of pages and higher associativity, we consistently observe peaks in execution time. We believe that these peaks originate from the increased flexibility of in-cache
placement, which negatively impacts the size of the state space. In general, as the number of pages is increased with a fixed associativity, the increment in runtime follows a regular trend and is exponential in time. Intuitively, this arises from the exponential increase in state space size to be explored by CBMC. It can also be noted that the verification time sharply decreases in those instances of verification that are not supposed to succeed. These cases, highlighted in the figure, correspond to those setup where the cache space is insufficient to carry out allocation, and where verification fails as it should. In this cases, CBMC stops after encountering a verification counter-example, hence it does not perform a complete exploration of the state space. Unfortunately, cases beyond associativity 6 consistently timeout in our evaluation.

In a second scenario, we evaluate the verification time for a more complex memory/cache layout by fixing the associativity to 1 and varying the number of pages. We consider a 32-bit system with 4 KB memory pages ($P_s = 12$), 64 byte cache line size ($O = 6$), and a way size of 64 KB ($I = 10$). In this layout, a single cache way can contain up to 16 memory pages. The results are depicted in Figure 3.23.

As shown in the figure, a sharp increment in runtime is observed at 6 profile pages. Although not included in the graph, any verification attempt for pages beyond that boundary runs longer than the selected 2 hours timeout threshold. Nonetheless, even with the current approach, verification is feasible on a general-purpose machine for a limited number of profile pages.

3.7 Summary

To achieve predictability on commercial multi-core platforms it is fundamental to explicitly control how shared hardware resources are being utilized by real-time and non-critical applications. In this chapter we
have described a set of hardware resource management techniques that aim at achieving predictability on COTS cache-based multi-core architectures. More specifically, we have focused on the key components of the shared memory hierarchy: shared caches, DRAM memory controller and DRAM memories.

First, we described Colored Lockdown, which represents a cache management framework which integrates a technique to perform deterministic cache allocation at an OS level. In Section 5.2 we describe how application memory behavior can be analyzed to optimize the benefit of last-level cache allocation performed via Colored Lockdown. While we have discussed Colored Lockdown in the context of multi-core systems, the same technique can also be used to manage the content of a cache in: (i) single-core platforms; and (ii) different levels of cache, including private caches as long as they provide support for line locking. Colored Lockdown can be used to avoid critical tasks from suffering interference coming from self-eviction, preemption, asynchronous kernel flows (i.e. ISRs), and other tasks running on the same core or on a different one. However, the most important property exported at an analysis level is that, by combining page coloring and cache lockdown, it is possible to enforce a deterministic cache hit rate on a set of known frequently accessed memory regions.

As a second step, we presented PALLOC, a DRAM bank-aware memory allocator, for performance isolation on multi-core platforms. It exploits the page-based virtual memory system to allocate memory pages on specific DRAM banks. This allows us to configure systems in a way to minimize bank sharing among concurrently executing applications. Using PALLOC, we performed an extensive set of experiments to investigate the performance impact of the private DRAM banking strategy on two COTS hardware platforms with a set of synthetic and SPEC2006 benchmarks. Our finding is that private DRAM banking significantly improves the quality of performance isolation and real-time performance on COTS multi-core platforms. However, we also found that partitioning DRAM banks (and caches) is still far from ideal performance isolation due to contention in other shared resources, including the memory bus, in the memory hierarchy.

In order to improve performance isolation, DRAM controller bandwidth management is key. For this reason, we have integrated a previously developed DRAM controller bandwidth partitioning mechanisms, called MemGuard [16]. The integration of all the developed components and MemGuard constitutes a framework an unified framework, namely Single Core Equivalence. With SCE, we have consolidated the proposed techniques onto the same platform and OS. On one hand, we demonstrate that SCE can be implemented on commercially available hardware. On the other hand, we discuss how SCE can contribute to lower the cost of real-time system development on multi-core platforms by allowing to reuse a large set of consolidated industrial practices. For this framework, we have proposed a set of theoretic results to analyze the application workload and perform schedulability analysis.

Finally, we focused our attention on verification of kernel-level cache management logic. We have demonstrated that it is possible to perform verification by reasoning directly on the system-level C code of the target module. Remarkably, we show that key properties for additional kernel-level features can be verified in a modular way with respect to the rest of the OS logic. In our approach, we relied on bounded model
checking via CBMC. The work opens many possibilities for improvement and expansion. As a part of our future work, we will investigate how to include elements of deductive verification to allow verification of more complex scenarios. Additionally, we will attempt verification of complementary real-time hardware kernel logic with the goal of establishing a industry-ready, verified real-time resource management framework.

3.8 Future Work

So far the majority of our efforts have gone in the direction of achieving inter-core performance isolation in spite of hardware resource sharing. An important step forward, however, is allowing a reliable and predictable mechanism for tasks on different cores to communicate. Inter-core communication, in way, represents a violation of perfect inter-core partitioning. Hence, the challenge is developing a predictable communication framework, where the introduced amount of interference can be controlled and bounded at analysis time. At the same time, it is important that the overhead arising from an explicit control of the communication flow do not surpass the benefit of parallel processing.

The methodology detailed for SCE, and the use of Colored Lockdown in particular, only provide directives about the use of the last level cache space. Nonetheless, multiple cache levels can co-exist in modern multi-core platforms. For this reason, an important step forward is advancing the framework to support both the profiling and allocation of data in multiple levels of caches. The problem is complicated by the difficulty to observe the behavior of all the cache levels at once. We envision that by extending out profiling techniques to fuse data from different performance counters and from frequency-based analysis, it is possible to create multi-level cache profiles.

There is a large margin of improvement in the interaction between Palloc and MemGuard. In the current implementation, MemGuard only performs regulation using a generic counter of DRAM transactions. However, read requests and write requests are handled very differently in the DRAM subsystems (see Section 2.3). Read requests are prioritized over write, the latter being processed in background (unless the write buffers are fulfilled). As a result, MemGuard could be augmented to exploit this extra bit of information when deciding whether or not the assigned fraction of guaranteed bandwidth has been reached at each core. This, however, can be done only assuming that the target platform provides an way to distinguish between read and write requests, and to separately accumulate transaction events related to the two types of requests.

Augmenting the SCE framework to use more profiling information about the scheduled tasks can also introduce large performance benefits. For instance, at the moment cache locking is performed only at the beginning of the IMA partition instance where the considered task runs. This means that at prefetch&lock time, a selection of frequently accessed memory pages needs to be loaded in cache. However, the memory working set of the task instance may change during execution, making the locked data a non-optimal selection of memory locations. If changes in tasks’ working sets are known and can be predicted, SCE could be extended to allow cache re-loads for the same task instance. Another example is exploiting known per-task
memory bandwidth requirements. The amount of bit per second requested by application to the memory subsystems vary as the task executes. As such, each task can be analyzed to extract its characteristic memory bandwidth demand profile. The OS could use this information to better assign MemGuard budget to applications. In fact, if task A executes for 100 ms requesting only 20% of the assigned budget, then the OS can reassign the unused 80% to improve the performance of tasks on different cores. Spare budget re-assignment is currently done in MemGuard following an heuristic. If deterministic bandwidth usage properties can be extracted via task profiling, the benefit in performance could be embedded into the analysis to lower its pessimism.
CHAPTER 4
PREDICTABILITY ON SCRATCHPAD-BASED
ARCHITECTURES

To cope with lack of performance isolation in multi-core platforms, the approach described in Chapter 3 aims at transparently de-conflicting access to shared hardware performed by mostly unmodified user-space applications. The regulation approach was proved to represent a viable solution. However, performing regulation on a platform designed for average performance optimization can lead to large partitioning-induced overhead.

The source of overhead is not to be identified in the regulation technology itself, but rather on the lack of assumption on the task behavior and structure. In fact, if the resource access pattern of tasks is allowed to be unrestricted, no assumption can be made about the behavior of interfering cores when analyzing the behavior of a task. This in turns leads to pessimistic assumptions and potentially large resource over-provisioning. In order to overcome the aforementioned drawbacks, a change in philosophy is necessary. In this chapter, we consider platforms that were designed following real-time oriented paradigms, as we detail in Section 4.1.

On the target platforms, we introduce a framework of techniques to enforce deterministic hardware resource management leveraging on additional information about the behavior of the tasks. We say that tasks are structured meaning that they to adhere to a well specified execution model.

4.1 System Model and Assumptions

This section summarizes the task model that we use and the hardware assumptions we rely on for the design of proposed OS-level techniques.

**Scratchpad Memories** The first assumption we make is the presence of scratchpad memory (SPM). We assume that each core in our system features a block of private scratchpad memory. Moreover, in this work we assume that the size of each per-core scratchpad memory is big enough to fully contain the footprint of any two tasks in the system. Hence, the footprint of the largest task in the system is at most half the size of the scratchpad memory. Although this assumption may appear restrictive, we make the following considerations. First, modern scratchpad-based micro-controllers provide scratchpad memories that have a size in the same order of magnitude as the main memory. For instance, in the Freescale/NXP MPC5777M that we use for our evaluation, each core includes 80 KB of scratchpad with a total main memory size of about 400 KB.
Second, hard real-time control tasks typically are compact in terms of memory size. Third, if a task violates this size constraint, known methodologies exist \cite{184, 185} to split a large application into smaller sub-tasks that are individually compliant with the imposed constraint.

Before tasks can be executed from SPM, their code and data need to be transferred from main memory. Thus, we adopt a task model that is composed of three phases: a **load phase**, an **execution phase** and an **unload phase**. First, during the load phase, the code and data image for the activated task is copied from main memory to the SPM. Next, during the execution phase, the loaded task executes on the CPU by relying on in-scratchpad data. Finally, the portion of data that has been modified and needs to remain persistent across subsequent activations of the task is written back to main memory during the unload phase.

**DMA Engines** To avoid to stall the CPU when load/unload operations are performed, we assume that copy operations toward/from the scratchpad memories can proceed in parallel with task executions. This can be achieved as long as execution and load/unload phases belong to two distinct tasks. In order to parallelize load/unload operations with task execution, we rely on direct memory access (DMA) engines. We assume that the hardware provides DMA engines that are able to transfer data from the main memory into the scratchpad and vice versa. By exploiting (i) the capability of parallelizing load/unload operations together with task execution, and (ii) the assumption that any task image can fit in half of the scratchpad memory, it is possible to hide task loading/unloading overhead during task execution, as we discuss in Section 4.2.1.

**Dedicated I/O Bus** The next made assumption is about the organization of the I/O subsystem. Since the activity of I/O devices is typically triggered by external events, it is inherently asynchronous. Unfortunately, unregulated I/O activity on the system bus can lead to unpredictable contention with CPU activity \cite{19}. Hence, unarbitrated I/O traffic represents one of the major sources of unpredictability in real-time systems. To deconflict the inherently asynchronous activity of I/O devices from application cores’ activity, we assume that a dedicated bus exists to route I/O traffic without directly interfering with CPU-originated memory requests. The idea of co-scheduling CPU activity and I/O traffic is not new and specific solutions have been proposed in \cite{19, 16}. However, the increased awareness of chip manufacturers about this problem has resulted in the design of COTS platforms that use dedicated buses to handle I/O transactions. Table 4.1 shows a non-exhaustive list of COTS platforms with this feature. In this work, we assume that suitable hardware exists to enforce a separation between I/O and CPU-originated memory transactions. Furthermore, traffic transmitted over the dedicated I/O bus needs to be handled, pre-processed and scheduled before reaching the application cores. Thus, we assume that an I/O processor exists, which we hereafter refer to as **I/O core**. Just like the application cores, the I/O core features a scratchpad memory that is used to buffer I/O data before they are delivered to applications.

Typically, devices that support high-bandwidth operations are DMA-capable. Instead, slower devices expose memory-mapped input/output buffers that can be read/written using generic platform DMA engines.
Without loss of generality, we assume I/O data transfers from/to the I/O core are performed by DMA engines and that data from I/O devices can directly be transferred into the I/O core’s scratchpad memory. In other words, I/O devices are not allowed to initiate asynchronous transfers directly towards main memory. As previously discussed, this design choice allows us to perform co-scheduling of CPU and I/O activities to achieve higher system predictability. A summary of the architectural assumptions discussed so far is provided in Figure 4.1.

### Memory Organization
As micro-controllers evolve into complex multi-core systems, more advanced support of memory protection schemes is provided. However, for the purpose of this work, no specific assumption needs to be made about platform memory protection features. Hence, the presence of a memory management unit (MMU) is not a necessary requirement. We discuss in Section 4.2.5 how task relocation from main memory to scratchpads can be achieved without MMU support. Intuitively, MMU support allows for a straightforward implementation of task relocation by relying on page table manipulation. Usually, systems without MMU include a memory protection unit (MPU). MPUs support the definition of per-core access permissions based on linear ranges of physical memory addresses. Although they are not necessary to implement our system, MPUs can be easily supported within our design.

The hardware assumptions described so far represent desirable features that are becoming increasingly common in modern COTS micro-controllers used for safety-critical applications. Table 4.1 provides a list of some of the available COTS platforms that satisfy the described assumptions. The MPC5777M and

### Error Detection On SRAM, Flash and SPM
We assume that hardware error detection logic is available for different kinds of memories, such as SPM, SRAM and Flash. For the purpose of our evaluation, we considered error detection capabilities of the Freescale MPC5777M processor. Our platform implements Hsiao codes that provide single bit error correction and double bit error detection (SEC-DED). Hsiao
Table 4.1: Suitable Commercial Multicore COTS platforms

<table>
<thead>
<tr>
<th>Features</th>
<th>NXP MPC5777M</th>
<th>NXP MPC5746M</th>
<th>TI TMS320C6678</th>
<th>ST SPC58NE84C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratchpad</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMA engines</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dedicated I/O bus</td>
<td>✓</td>
<td>✓</td>
<td>✓ (priority-based)</td>
<td>✓</td>
</tr>
<tr>
<td>SRAM ECC</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Code [186] for correction and detection is popular in modern embedded platforms.

Although for our implementation we use double bit error detection, our approach works with any kind of error detection logic, as long as it is possible for the OS to determine the location of the error in physical memory.

**Task Model** For the proposed design, we consider a partitioned and fixed priority scheduling policy; additionally, each core has a set $\mathcal{\Gamma}$ of $N$ sporadic tasks, $\{\tau_1, \ldots, \tau_N\}$, each with different priority whereby $\tau_1$ has the highest priority and $\tau_N$ has the lowest priority. The deadline of each task is assumed to be less than or equal to its minimum inter arrival time. The tasks in our system follow a three-phases model, composed of: load, execution, unload. In order to satisfy their temporal constraints, the last phase (unload) of a task needs to complete before the deadline. As we detail in Section 4.2.1, load and unload phases are performed inside a TDMA schedule. Since by design task load/unload duration is shorter than a single TDMA slot $\sigma$, the length of these operations is not explicitly considered in the task model. For ease of implementation, this work assumes non-preemptive tasks, although we plan to relax this assumption as part of our future work.

4.1.1 Summary of Parameters

Table 4.2: Summary of parameters for resource co-scheduling techniques in scratchpad-based platforms.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>Number of application cores in the system</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of tasks assigned to the core under analysis</td>
</tr>
<tr>
<td>$\tau_i$</td>
<td>Task $i$, with $1 \leq i \leq N$</td>
</tr>
<tr>
<td>$T_i$</td>
<td>Minimum inter-arrival time (MIT) for task $\tau_i$</td>
</tr>
<tr>
<td>$C_i$</td>
<td>Worst-case execution time of task $\tau_i$, including all overheads</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>TDMA slot size for the DMA scheduling</td>
</tr>
</tbody>
</table>

Table 4.2 summarizes the notation used for task and system parameters. These parameters will be used throughout Chapter 4 to describe the design and analysis of the proposed SPM-centric OS.
4.2 Scratchpad-Centric OS

We propose to re-design a RTOS to enforce inter-core isolation via explicit co-scheduling of shared hardware resources. As described in Section 4.1, we focus on emerging embedded scratchpad-based multi-core platforms. Scratchpad memories, in fact, have been proven to provide better temporal isolation when compared to traditional caches [11, 12]. Alongside, we exploit additional hardware features that vendors are now including in some modern families of multi-core platforms designed for the embedded market, such as: separate I/O and memory buses, the presence of dual-port memories with DMA support, and core specialization.

The key insight is that, in order to prevent unregulated access to shared components of the memory hierarchy, we exploit local SPM. Hence, we refer to our design as scratchpad-centric OS, or SPM-centric OS. In our scratchpad-centric OS, resource control has been extended to include a fully deterministic I/O subsystem. In fact, we rely on the presence of a dedicated bus to perform I/O transactions. Hence, asynchronous activity remains confined within the I/O subsystem. Upon task load/unload only task-relevant I/O data are transferred from/to the I/O subsystem. The benefit of this design is twofold. First, it allows high-level scheduling of accesses to main memory, ultimately achieving conflict-free execution of tasks from local memories. Second, performance benefits derived from the usage of fast scratchpad memories are exploited, ultimately combining better performance with higher temporal determinism.

4.2.1 OS Design

The central idea of the proposed SPM-centric OS is resource specialization. As previously mentioned, a specialized I/O core and I/O bus are used to handle peripheral traffic. Similarly, a specific role is assigned to different memory resources in the system. Specifically, three types of memory resources exist in our system, as depicted in Figure 4.1. First, flash memories are used to persistently store application/OS code, read-only data, as well as initialization values of read-write portions of main memory. Next, the SRAM (main) memory contains writable application and system data that represent the time-variant state of the system. Finally, scratchpad memories temporarily store a copy of code and data images for those tasks that are currently being scheduled for execution.

In our solution, applications are never executed directly from main memory, thus we adopt the following strategy: (1) task images are permanently stored in flash and loaded into main memory at system boot; (2) a dedicated DMA engine is used to move task images to/from SPM upon task activation; (3) a secondary DMA engine is used to perform I/O data transfers between devices and I/O core; (4) tasks always execute from SPM; (5) only task-relevant I/O data are transferred upon task load from the I/O subsystem. The benefit of this design is twofold. First, it allows high-level scheduling of accesses to main memory, ultimately achieving conflict-free execution of tasks from local memories. Second, performance benefits derived from the usage of fast scratchpad memories are exploited, ultimately combining better performance with higher temporal determinism.
We refer to the capability of our SPM-centric OS to dynamically move applicative tasks in and out of the SPM memories as support for relocatable tasks. As mentioned in Section 4.1, if hardware MMU support exists, task relocation can be achieved using page table manipulation. Otherwise, advanced compiler level techniques can be exploited to generate position independent code, as described in Section 4.2.5.

In proposed SPM-centric OS, a DMA engine is used to position the image of a relocatable task inside a SPM for execution. We refer to this DMA engine as application DMA. Similarly, we refer to the platform DMA used for I/O transfers as peripheral DMA. Typically, a single DMA engine is capable of utilizing the full main memory bandwidth in micro-controller platforms. Nonetheless, the design constraint that imposes the use of a single applicative DMA can be relaxed if the main memory subsystem allows two or more DMA engines to transfer data concurrently without saturating the main memory bandwidth.

### 4.2.2 Scratchpad and CPU Co-scheduling

Load/unload operations for tasks running on the $M$ applicative cores need to be serialized to prevent unregulated contention over the memory bus. Hence, only a single DMA is required as application DMA for all the $M$ applicative cores. Several schemes are known to fairly share a single resource across different consumers. For the scope of our design, we employ a time division multiple access (TDMA) scheme to serialize task load/unload operations among $M$ applicative cores. The main advantage of the TDMA scheme lies in its simplicity of implementation. Although in this work we restrict our discussion to TDMA sharing of the applicative DMA, the proposed OS can be extended to consider round-robin policies as well as budget-based schemes.

In order to perform TDMA-based scheduling of the application DMA, time is partitioned into slots of fixed size. In each slot, only a single DMA operation can be performed, either a task load or unload. The slot size is chosen to ensure that the task with the largest footprint in the system can be loaded within the slot time window. Figure 4.2 depicts the sequence of operations in our TDMA scheme for a system with $M = 2$ application CPUs. Note that the TDMA enforcement needs to be centralized. Hence, in our design, the I/O core is responsible for interfacing with application cores’ schedulers through active/ready queues, programming the application DMA as well as enforcing the time-triggered TDMA slots. In particular, Figure 4.2 depicts three tasks scheduled on one core. Up arrows in blue color represent the arrival times of the considered tasks; we use colors for two different partitions. A task can only run after its load operation has been completed and the previous task on the other partition has completed, (see $\tau_2$ to $\tau_3$ and $\tau_1$ to $\tau_2$ for example of the two cases). There might be slots where no load/unload is performed. This happens at time 8: $\tau_1$ finishes right after the beginning of the slot, so both partitions are full at the beginning of the slot and the I/O core can neither load nor unload any applicative core scratchpad. Effectively, the slot is wasted.

Since tasks need to be loaded/unloaded in parallel with respect to CPU activity, two partitions are created
on the scratchpad. There is logically no difference between the two scratchpad partitions. Thereby, tasks may execute from either one of the two, depending on their arrival time. Interchangeably, one of them contains the image of the task which is currently being executed, while the second half is used to load (unload) the image of the next (previous) task to be executed (that was completed). Note that when a task is executing on the CPU while a second task is loaded/unloaded in background, CPU and DMA contend for scratchpad access. However, the impact of this contention on the timing of the tasks is typically negligible for two main reasons. First, scratchpads are often implemented as dual-ported memories; thus, they are able to support stall-free CPU and DMA operations. In fact, on the considered MPC5777M platform we have verified this by experimentation and found that both the core and the DMA module do not suffer any delay when they access the SPM simultaneously. Second, in a system with $M$ CPUs, DMA-CPU contention over scratchpad involves only two masters, as opposed to the traditional approach where up to $M$ masters could contend for main memory.

As depicted in Figure 4.2, the application DMA is alternatively assigned to transfer data for a specific core. Within a single slot, either an unload operation for a previously running task or a load operation for the next scheduled task is performed. The specific operation to be performed is decided as follows:

**Rule 1:** If a load operation can be performed, a load operation is programmed on the application DMA;

**Rule 2:** If a load cannot be performed and there is a previously running task to be unloaded, an unload operation is programmed on the application DMA.

Note that Rule 1 can be activated by the following conditions: (i) at least one of the two SPM partitions is available (i.e. has been previously unloaded), and (ii) a task has been released and is ready to be loaded. Similarly, Rule 2 can be activated if no load can be performed, at least one partition is not empty and the task loaded on that partition has completed.

In the proposed design, the next task to be executed is loaded in background while the foreground running task is not interrupted until its completion. The described mechanism allows to hide the DMA loading

![Figure 4.2: Scheduling CPU, DMA and local memory.](image-url)
overhead, avoiding contention in main memory and exploiting performance benefits deriving from SPM usage.

The work-flow followed by an applicative core and the I/O core at the boundary of each TDMA slot is depicted in Figure 4.3. Specifically, at each time slot, the I/O core checks the status of the queue of active tasks belonging to the considered core. If a task that is active for execution but not ready (i.e. not relocated in scratchpad) is found, the I/O core checks which SPM partition (P1 or P2) is empty on the application core. If any partition is found to be empty (Slot #1), the I/O core programs the application DMA to load the topmost active task to the empty partition. Once the load is complete, the I/O core updates the active and ready queues of the considered application core. The latter operation allows the application core to begin the execution of the task (Slot #2). Note that since only one task can be in running state on the CPU, there is always a SPM partition that is available for load/unload operations.
4.2.3 I/O Subsystem Design

Together with memory resources, applications typically need to communicate with peripherals and thus require I/O data to operate. We propose an I/O subsystem design that enforces a complete separation between task execution and the asynchronous activity of I/O peripherals: this goal is achieved by offering to application tasks a synchronous view of I/O data. It is achieved by distinguishing between data production and their dispatch to/from tasks. In fact, we allow I/O data to flow from/to I/O subsystem to tasks only at the boundary of load/unload operations.

As mentioned in Section 4.1, we assume that a dedicated bus connects the SPM of I/O core with peripherals. Hence, asynchronous peripheral traffic can reach the I/O subsystem without interfering with task execution. For each device used in the proposed system, the OS defines a statically positioned device buffer on the I/O core scratchpad. A device buffer is further divided into an input device buffer and an output device buffer. The input (output) device buffer represents the position in memory where data produced by devices (tasks) is accumulated before being dispatched to tasks (devices).

In our design, peripheral drivers can operate with an interrupt-driven or polling mechanism. For DMA-capable peripherals supporting interrupt-driven interaction, the driver only needs to specify the address in SPM of the device buffer from/to where data are transferred. The driver is also responsible for updating device-specific buffer pointers to prevent a subsequent data event from overwriting unprocessed data. For interrupt-driven interaction with non-DMA-capable devices, the driver uses the platform peripheral DMA to perform data movement. Similarly, the device driver is periodically activated and the peripheral DMA is used to perform data transfer for polling-based interaction with devices.

In general, device-originated interrupts as well as timer interrupts for device driver activations are prioritized according to how critical is the interaction with the considered device. Nonetheless, all the device-related events are served with priority levels that are lower than task-scheduling events, such as: (i) TDMA slot timer events and (ii) completion of application DMA loads/unloads.

In order to interface with a peripheral, application tasks define subscriptions to I/O flows. A subscription represents an association between a task and a stream of data at the I/O device. For instance, a given task could subscribe for all the packets arriving at a network interface with a specific source address prefix. Task subscriptions are metadata that are stored within the task descriptor.

For each task in the system, a pair of buffers (for input and output respectively) is defined on the SPM of the I/O core to temporarily store data belonging to subscribed streams. Since the content of these buffers will be copied to/from the application cores upon task load/unload, we refer to them as task mirror buffers. Consider the arrival of I/O data from a device. As soon as the interaction with the driver is completed, the arrived data is present in the corresponding device buffer. According to task subscriptions, the OS is responsible for copying the input data to all the mirror buffers of those tasks subscribed to the flow.

The advantage of defining mirror buffers lies in the fact that when a task needs to be loaded, all the
peripheral data that need to be provided are clustered in a single memory range. Consequently, during
the loading phase of a task, the application DMA is programmed to copy the content of the mirror input
buffer together with task code and data images to the application core. The reverse path is followed by
task-produced output data during the task unload phase.

Since I/O data are delivered to applicative tasks at the boundary of load/unload operations, the approach
presented in Section 4.2.4 for the calculation of tasks’ response time can be reused to reason about end-to-end
delay of I/O-related events.

4.2.4 Schedulability Analysis

Given the scheduling strategy described in Section 4.2.1, we can calculate a safe bound on the worst case
execution time based on all tasks’ parameters in an approach similar to [69]. Note that we assume that the
task’s execution time, $C_i$, is actually the adjusted execution time in which all the overheads are included,
such as the context-switch and the DMA setup routines. Also note that for simplicity we discuss the case
with $M = 2$ cores, since it is used in our prototype, but the analysis could be trivially extended to account
for any number of cores by changing the length of the DMA operations.

Figure 4.4 depicts an illustrative example of the worst case scheduling scenario (critical instant) for an
example task set where $\tau_3$ is the task under analysis. The schedule depicts a busy period where $\tau_3$ suffers
interference from two higher-priority tasks, $\tau_1$ and $\tau_2$. As in [69], we consider the busy period as composed
by a sequence of scheduling intervals $Interval_1, Interval_2, Interval_3, Interval_4$ (each bounded by bold vertical
lines in the figure), followed by a final interval $Interval_F$. During each scheduling interval, only one blocking
or interfering task runs. During the final interval, the task under analysis runs. Each scheduling interval always starts with a CPU execution and ends either when the CPU finishes executing the task or when the next task finishes being loaded by the DMA, whichever happen last; at this point, the next interval starts with the execution of the loaded task. The final interval starts with the execution of the task under analysis and finishes when the task under analysis is unloaded.

We say that a scheduling interval is CPU-bound when it ends with CPU execution (ex: Interval$_1$, Interval$_3$ and Interval$_4$ in the figure), and I/O-bound when it ends with DMA load operation (ex: Interval$_2$). The length of a scheduling interval is the maximum between the execution time of the task running in the interval and the DMA operations required to load the next task. We denote the size of the TDMA slot as $\sigma$; since in the worst case a load/unload operation can occupy the entire slot, we upper bound the length of DMA operations as a multiple of $\sigma$.

Response Time Calculation Building on the above-mentioned definitions, we can use the same algorithm detailed in [69] to compute the worst case response time for the task under analysis, by showing that the problem is equivalent to the one in [69]. The algorithm in [69] computes the response time by adding three components: (1) the blocking time $B$ caused by a lower priority task that starts executing before the beginning of the busy period; this is Interval$_1$ executing task $\tau_5$ in the figure; (2) the interference $H$ comprising the remaining scheduling intervals in the busy period, which are Interval$_2$, Interval$_3$ and Interval$_4$ in the figure. The number of such intervals is equal to the number of interfering higher priority jobs plus one, since an extra lower priority job that starts loading before the beginning of the busy period ($\tau_4$ in the figure) can execute within the busy period itself; (3) the computation of the task under analysis ($C_i$). The algorithm builds a list of DMA times and computation times for tasks executed in $H$, then it derives a provably safe bound on the length of $H$ using standard response time iteration.

Compared to [69], our solution differs in three aspects. First, in this work we use fixed-size DMA operations, while [69] employs dynamic-size DMA operations. Therefore, we need to discuss how to compute the length of the DMA operations that are inserted in the list of DMA times. Second, we need to recompute the length of the blocking time $B$ since the next task to be loaded is determined at a different time. Finally, unlike [69], we consider the task under analysis finished when the task is unloaded, at the end of Interval$_F$. Consequently, we can use the same algorithm to compute the worst case response by replacing $C_i$ with the length of Interval$_F$. We address each point in sequence.

Scheduling Intervals in The Busy Period ($H$) When the system is busy with both SPM partitions occupied and at least one pending task, within each interval we need to first unload the previous partition and then load it with the next task. Therefore, for any scheduling interval, it will require four TDMA slots ($4 \cdot \sigma$) to load the next task if the interval was preceded by another I/O-bound interval, such as for Interval$_3$ in the figure. On the other hand, if the interval is preceded by a CPU-bound interval, it might require up to
five TDMA slots ($5 \cdot \sigma$) to finish loading the next task in the worst case, as for Interval$_2$. This is because the CPU-bound interval can induce an unused empty TDMA slot in the next interval (slot [4:5] in the figure).

As a result, the length of any scheduling interval can be computed as either $\max(C, 4 \cdot \sigma)$ or $\max(C, 5 \cdot \sigma)$. We now formally prove that for any CPU-bound interval to cause the worst case scenario, with the exception of the first interval Interval$_1$, the CPU execution has to be strictly longer than four TDMA slots ($4 \cdot \sigma$).

**Lemma 4** For any scheduling interval in $H$, no extra empty slot will be induced in the next interval unless the length of the CPU execution is strictly greater than four TDMA slots ($4 \cdot \sigma$).

**Proof:**

We show that any scheduling interval in $H$ with CPU execution less than $4 \cdot \sigma$ cannot induce an empty slot in the next interval. By considering the figure above, the execution of $\tau_x$ in the middle interval is greater than $4 \cdot \sigma$, and it induces an empty slot in the next interval. Since both partitions must be full during the execution of intervals in $H$, it follows that the interval to which $\tau_x$ belongs must include both a load and an unload operation; in the worst case showed in the figure, the interval could start with the unload operation. Still, clearly if the execution time of $\tau_x$ is reduced to less than $4 \cdot \sigma$, the interval will finish before the next slot assigned to the core under analysis is reached, and hence no empty slot will be induced. Note that if the execution of $\tau_x$ is reduced further, it could make the interval into an I/O-bound interval, which would end right after finishing the load of the next task; thus, the next interval would not suffer from an empty induced slot either.

Based on Lemma 4, we can construct the list of DMA times used by the algorithm as follows: we insert in the list a number of $5 \cdot \sigma$ time values equal to the number of tasks executed in $H$ with length greater than $4 \cdot \sigma$, plus one task (to account for the task in Interval$_1$, which can cause an extra empty TDMA slot as in the figure). The remaining DMA times in the list are equal to $4 \cdot \sigma$.

**Critical Instant and Blocking Time (B)** At the beginning of the example schedule in Figure 4.4, the system has two free local SPM partitions at time zero. In Interval$_1$, the task under analysis $\tau_3$ is released along with all higher-priority tasks after an arbitrarily small time ($\epsilon$) when all free partitions have been loaded or have started loading lower-priority tasks ($\tau_5$ and $\tau_4$); this is $\epsilon$ after time 2 in the figure. The task under analysis $\tau_3$ cannot run until the pre-loaded lower-priority tasks ($\tau_5$ and $\tau_4$) plus all higher-priority tasks ($\tau_1$ and $\tau_2$) finish execution. We now prove that the discussed scenario is indeed the critical instant for our system, leading to the worst case response time for the task under analysis.

**Lemma 5** The critical instant is produced when the task under analysis $\tau_i$ and all higher priority tasks arrive immediately after a lower priority task has started loading into a partition, and the other partition was loaded with another lower priority task as late as possible (i.e., two slots before).

107
Proof: We first show that in the worst case, both \( \tau_i \) and all higher priority tasks must arrive \( \varepsilon \) time after the beginning of a slot where a lower priority task is loaded. If either \( \tau_i \) or a higher priority task would arrive at or before the beginning of the slot, then such task would be loaded and executed in place of the lower priority task. Hence, the length of the busy period would decrease by one scheduling interval, which cannot produce the worst case response time for \( \tau_i \). If instead \( \tau_i \) arrives some \( \delta \) time later during the busy period, then the finishing time of \( \tau_i \) would not change, but the response time of \( \tau_i \) would decrease by \( \delta \). Finally, if a higher priority task arrives later during the busy period, the number of interfering jobs of the task could only be lower or equal compared to releasing it immediately after the beginning of the slot. Hence, the described activation pattern must lead to the critical instant.

For what concerns the lower priority task pre-loaded in the other partition, it suffices to notice that loading the task as late as possible (i.e., two slots before \( \tau_i \) arrives, which is slot [0:1] in Figure 4.4) maximizes the amount of execution of the task within the busy period.

Based on Lemma 5, the worst case blocking time \( B \) can be obtained as the length of \( Interval_1 \) minus \( \sigma \), where the length of \( Interval_1 \) is bounded by \( \max(C_u, 2 \cdot \sigma) \); here, \( C_u \) is the WCET of any low priority task \( \tau_u \) executed in \( Interval_1 \), while the length \( 2 \cdot \sigma \) accounts for the fact that the next task is loaded in the second slot of the interval (slot [2:3] in the figure). Similar to [69], since we can make no assumption on which lower priority tasks execute in \( Interval_1 \) and \( Interval_2 \), the algorithm simply considers the two lower priority tasks with the longest execution times.

**Final Interval (F)** The length of the final interval \( Interval_F \) can be computed as \( \max(C_i + 5 \cdot \sigma, 7 \cdot \sigma) \), where \( \tau_i \) is the task under analysis. In the example depicted in Figure 4.4, the length of \( Interval_F \) is \( C_3 + 5 \cdot \sigma \). The other case can happen when \( C_3 \) is short enough and slot [22:33] is utilized, in the worst case, to load the next task. In this situation, up to seven TDMA slots are required to finish unloading \( \tau_3 \), as formally proven below.

**Lemma 6** The length of \( Interval_F \) is upper bounded by \( \max(C_i + 5 \cdot \sigma, 7 \cdot \sigma) \).

**Proof:** Similar to scheduling intervals in \( H \), we need to consider two cases: (1) the length of the interval is bounded by \( C_i \) plus the time required to unload the task; (2) the length of the interval is bounded by the time required to unload/load the other partition before unloading \( C_i \). For the first case, note that differently from scheduling intervals in \( H \), there might be no pending task at the start of \( Interval_F \), since the last task in the busy period (task under analysis) is running. Therefore, a new job of any task could be release later during \( Interval_F \) and start a load operation. In particular, the new job could arrive just before the unload of the task under analysis (\( \tau_i \)), as shown in Figure 4.4. In this case, since there is one free partition and load operations have priority over unload operations (Rules 1,2), the new job has to be loaded first; thus, the unload of \( \tau_i \) is delayed by up to \( 5 \cdot \sigma \) in the worst case (one empty slot plus four other slots, as shown in Figure 4.4 for slots [22:27]); no more than 5 slots are possible since after the load at [24:25], both partitions
Table 4.3: Characteristics of Freescale MPC5777M SoC

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip Name</td>
<td>MPC5777M (Matterhorn)</td>
</tr>
<tr>
<td>Manufacturer</td>
<td>Freescale</td>
</tr>
<tr>
<td>Architecture</td>
<td>Power-PC, 32-bit</td>
</tr>
<tr>
<td>CPU Unit</td>
<td>2x E200-Z710 + 1x E200-Z709 + 1x E200-Z425 (I/O)</td>
</tr>
<tr>
<td>Processing Unit</td>
<td>CPUs, DMA, Interrupt Controller, NIC</td>
</tr>
<tr>
<td>Operational Modes</td>
<td>Parallel + Lockstep (on one applicative core)</td>
</tr>
<tr>
<td>ECC Protection</td>
<td>Cache, RAM, Flash Storage</td>
</tr>
<tr>
<td>Cache Hierarchy</td>
<td>L1 (Private Instructions + Data) + Local Memory</td>
</tr>
<tr>
<td>Local Memory (SPMs)</td>
<td>Instructions (16 KB) + Data (64 KB)</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>Instructions (16 KB) + Data (4 KB)</td>
</tr>
<tr>
<td>Main Memory</td>
<td>404 KB, SRAM</td>
</tr>
<tr>
<td>Flash Size</td>
<td>8 MB</td>
</tr>
<tr>
<td>Main Peripherals</td>
<td>Ethernet, Flexray, CAN, I2C, SIUL</td>
</tr>
</tbody>
</table>

are full and thus an unload must happen next). In this case the length of $\text{Interval}_F$ is upper bounded by $C_i + 5 \cdot \sigma$.

The following figure shows the other case where the length of $\text{Interval}_F$ is $7 \cdot \sigma$ in the worst case. This case happens when the execution of the task under analysis is very small to the point that $C_i + 5 \cdot \sigma$ is smaller than the required number of TDMA slots to actually unload $\tau_i$. When CPU execution of $\tau_i$ is sufficiently small, the load of the next task has to be after $5 \cdot \sigma$ at most regardless of the release time of the next task, otherwise $\tau_i$ would be unloaded by the fifth slot. If the next task is indeed loaded before the unload of $\tau_i$ as shown in the figure, then in the worst case it takes two more slots to unload $\tau_i$ (given that both partitions are full after loading the next task), hence resulting in a bound of $7 \cdot \sigma$. To conclude, by taking the maximum of the two cases we guarantee to capture the worst case.

![Figure showing the different cases for $\text{Interval}_F$](image)

4.2.5 Implementation

In this section, we provide the details of SPM-centric OS implemented using a COTS platform that supports the hardware assumptions described in Section 4.1.

**Architectural Overview of Considered Platform** For the implementation, we used a Freescale MPC5777M micro-controller unit (MCU). This MCU is the most advanced SoC in the Freescale MPC line as of Q4 2015. A brief summary of the architectural features of the MPC5777M MCU is provided in Table 4.3. The chip
includes four processors: two E200Z710 application cores operating at 300 MHz and a single E200Z425 I/O core. An additional non-programmable core is included for delayed lockstep operation.

Each core features private, globally accessible scratchpads for instructions and data, with a size of 16 KB and 64 KB respectively. No MMU is available on this platform. Hence, there is no support for virtual memory. Application cores can directly access the SRAM through a dedicated bus. A separate and slower bus is dedicated for transferring peripheral data to/from the I/O core.

**Implementation of SPM-centric OS using Erika Enterprise** Proposed SPM-centric OS was implemented using Evidence Erika Enterprise[^1]. Erika Enterprise is an open-source RTOS that is compliant with the AUTOSAR[^2] (Automotive Open System Architecture) standard. AUTOSAR is an open standard for automotive architectures providing a basic infrastructure for vehicular software. Erika Enterprise features a small memory footprint, supports multi-core platforms and implements common scheduling policies for periodic tasks. We performed a porting of Erika Enterprise on the MPC5777M MCU, adding support for UART communication interface, interrupt controller, caches, memory protection unit (MPU), data engines (DMA), and Ethernet controller.

In order to implement our SPM-centric OS, we have augmented Erika Enterprise to support position-independent (relocatable) tasks. We rely on the compiler[^3] support for *far-data* and *far-code* addressing modes. In this way, tasks are compiled to perform program-counter-relative jumps and indirect data addressing with respect to an OS-managed base register. We have extended the default task loader to exploit DMAs for transferring task images from SRAM to local memories and vice-versa. Similarly, the OS scheduler has been adapted to implement the strategy discussed in Section 4.2.1.

In Erika Enterprise, tasks are compiled and linked directly inside the image of the OS. For each task in the system, Erika-specific meta-data need to be defined. Additionally, meta-data that extend the task descriptors for SPM-centric operations are required. Manually configuring these parameters is tedious and error-prone; hence, we developed an OS configurator. The tool uses high-level task definitions and generates the final configuration for our SPM-centric OS. Specifically, each core is associated with a set of configuration files that describe: number of tasks, their priority, task entry points, initial status and so on. When a task is added, these files need to be configured accordingly.

First, the body of all the tasks is placed in an ad-hoc file. Similarly, task-specific data that need to be preserved across activations are defined in different files and surrounded with appropriate compiler-specific *PRAGMA*. This is fundamental to ensure that: (A) specific linker section is used to store task code and data images; and (B) position-independent data and instructions are generated. A separate file also defines the relocatable task table, which stores the status of each relocatable task. This structure includes: (A) position

[^1]: [http://erika.tuxfamily.org/drupal/](http://erika.tuxfamily.org/drupal/)
[^3]: Applications and OS are compiled using the WindRiver Diab Compiler version 5.9.4 - [http://www.windriver.com/products/development-tools/](http://www.windriver.com/products/development-tools/)
in SRAM of the task code and data images; (B) position of the task’s I/O data buffers; (C) current status of the task (e.g. loaded, completed, unloaded); (D) SPM partition of last relocation.

4.2.6 Evaluation

To validate the proposed design and implementation, we performed a series of experiments, whose results are summarized in this section. First we investigate the overhead of SPM management. Next, we consider the performance and predictability benefits of our approach with synthetic as well as real benchmarks. The achievable I/O bandwidth supported by our design is also measured. Finally, we investigate the schedulability results of the proposed strategy.

**Overhead Evaluation** A crucial parameter of proposed system is the size of the TDMA slot. This should be long enough to allow the completion of a load (or unload) operation for the task with the largest footprint in the system. However, in order to derive an upper-bound, we assume that a task footprint is constrained by the size of an SPM partition. Thereby, we measured the time to copy from/to half SPM (one partition) of an applicative core and derive the TDMA slot size accordingly. The results are reported in Table 4.4.

The application DMA needs to be programmed by the I/O Core to perform task relocation. Hence, DMA programming time represents an overhead introduced by our design. The time required to program the DMA has been measured and is reported in Table 4.4. Similarly, Table 4.4 reports the measured context-switch overhead of the implemented scheduler.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition load time</td>
<td>432</td>
</tr>
<tr>
<td>Partition unload time</td>
<td>432</td>
</tr>
<tr>
<td>DMA setup</td>
<td>3.16</td>
</tr>
<tr>
<td>Context switch</td>
<td>0.46</td>
</tr>
</tbody>
</table>

**Results on Achievable I/O Bandwidth** The performance of the proposed I/O subsystem (see Section 4.2.1) depends on the frequency of load/unload operations. In order to measure the achievable I/O bandwidth of proposed design, we have implemented support for the onboard Fast Ethernet Controller (FEC). The FEC is capable of transmitting data at the highest bandwidth among all the devices of the considered MCU. Hence, it represents the best I/O component to stress-test our design.

We have connected the FEC to an external node which generates constant-rate traffic. Specifically, the traffic source generates a 1 KB packet every 100 µs (1000 Hz, about 82 Mb/s). The payload of each packet contains a flow-ID chosen from 4 different values in round-robin. On used MCU, each applicative core runs
two tasks that have subscribed to I/O data flows based on packets’ flow-IDs. Device buffers and task (mirror) I/O buffers have been dimensioned to accommodate a single packet per task, with an overwrite policy.

With this setup, we have derived the raw achievable bandwidth considering two different values of TDMA slot size. Specifically, we measured the data rate of packets that are processed and looped back on the network interface using the Wireshark packet analyzer. Our experiments revealed an achievable bandwidth for the outgoing traffic of 4 Mb/s with a TDMA slot of 800 µs, and 8 Mb/s with a TDMA slot of 400 µs. Although this represents a fraction of the physically available bandwidth (100 Mb/s), being able to sustain a bandwidth higher than 1 Mb/s constitutes a promising result given that the platform operates at a clock frequency of few hundred Hz.

Performance Evaluation – Synthetic Benchmarks We investigate the performance of SPM-based execution as opposed to a traditional execution model. For this purpose, we have developed a set of synthetic benchmarks that exhibit different memory access patterns. Figure 4.5 depicts the runtime for such benchmarks on one of the two applicative cores. The first cluster of bars refer to the runtime of the benchmark that exhibits good data locality. Hence, when it is executed from SRAM, caches are effective at hiding SRAM access latency and significantly reduce task execution time. The next two clusters of bars show that

\[\text{https://www.wireshark.org/}\]
misses suffered for only instruction fetches or only data fetches already induce a significant execution slowdown (around 2x). The need for accessing SRAM data also introduces runtime fluctuation (about 25%) as a result of inter-core interference. Such effect becomes even more severe with applicative code that experiences misses while accessing both instructions and data. If the cost of accessing SRAM memory together with the slowdown due to inter-core interference are considered, an overall 3.5x slowdown is experienced when compared to what has been observed in the ideal case (100% cache hits). Finally, notice that if a task is able to entirely execute from scratchpad, its execution time is comparable to the ideal case and inter-core interference is prevented. These results are a strong motivation to best use available scratchpads in order to improve performance and avoid inter-core interference.

**Performance Evaluation – EEMBC Benchmarks** Next, we investigate the behavior of EEMBC benchmarks on the selected platform. For this purpose, we have ported and measured the execution time of the full suite of automotive EEMBC benchmarks under two scenarios: traditional contention-based execution from SRAM and proposed SPM-based execution. The results of normalized execution times are shown in Figure 4.6. From the results, we note that computation intensive benchmarks do not benefit from SPM-based execution. Conversely, for memory intensive benchmarks SPM-based execution determines substantial speed-ups (up to 2.1x).

Table 4.5 shows the execution time of the full suite of EEMBC automotive benchmarks. Furthermore, Table 4.5 also provides the footprint size of the considered benchmarks. It can be noted that all the considered benchmarks fit into a single scratchpad partition. These results validate the applicability of proposed design in real scenarios.
Table 4.5: Details of EEMBC Benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SPM Time (µs)</th>
<th>SRAM Time (µs)</th>
<th>Code Size (bytes)</th>
<th>Relocatable Code Size (bytes)</th>
<th>Data Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tblook</td>
<td>1013</td>
<td>1015</td>
<td>1804</td>
<td>1892</td>
<td>10516</td>
</tr>
<tr>
<td>matrix</td>
<td>1053</td>
<td>1054</td>
<td>4430</td>
<td>4774</td>
<td>4488</td>
</tr>
<tr>
<td>a2time</td>
<td>1002</td>
<td>1029</td>
<td>2175</td>
<td>2538</td>
<td>1704</td>
</tr>
<tr>
<td>ptrch</td>
<td>1036</td>
<td>1145</td>
<td>1000</td>
<td>1398</td>
<td>4924</td>
</tr>
<tr>
<td>tsprk</td>
<td>383</td>
<td>425</td>
<td>4124</td>
<td>4772</td>
<td>8160</td>
</tr>
<tr>
<td>ifirf</td>
<td>1040</td>
<td>1189</td>
<td>3288</td>
<td>3512</td>
<td>1000</td>
</tr>
<tr>
<td>can rdr</td>
<td>1009</td>
<td>1359</td>
<td>1370</td>
<td>1562</td>
<td>12440</td>
</tr>
<tr>
<td>bitmask</td>
<td>990</td>
<td>1389</td>
<td>3152</td>
<td>3282</td>
<td>1116</td>
</tr>
<tr>
<td>rspeed</td>
<td>1012</td>
<td>1457</td>
<td>710</td>
<td>1208</td>
<td>13212</td>
</tr>
<tr>
<td>puwm</td>
<td>1036</td>
<td>1540</td>
<td>1716</td>
<td>2500</td>
<td>2412</td>
</tr>
<tr>
<td>aifirf</td>
<td>1005</td>
<td>1564</td>
<td>1554</td>
<td>2286</td>
<td>1552</td>
</tr>
<tr>
<td>aifitr</td>
<td>916</td>
<td>1642</td>
<td>3720</td>
<td>4458</td>
<td>8448</td>
</tr>
<tr>
<td>aiffft</td>
<td>1170</td>
<td>2092</td>
<td>2796</td>
<td>3540</td>
<td>9224</td>
</tr>
<tr>
<td>idct</td>
<td>1045</td>
<td>2126</td>
<td>4498</td>
<td>4690</td>
<td>244</td>
</tr>
</tbody>
</table>

**Schedulability Results** For the schedulability evaluation of our approach, we compare our system against the contention-based system, in which cores use caches but are left unregulated when accessing main memory. Standard response time analysis is applied on both our system and the contention-based system for the same simulated workload. We have considered the applications in Table 4.5 to generate sets of random tasks (workloads). Given a system utilization, each application is randomly selected and assigned a random period in the range between 10 ms to 100 ms. The task’s utilization is then computed based on the measured execution time of each application and its selected period. At every iteration a new task is randomly generated. The generation stops when the sum of the individual tasks’ utilizations reaches the required system utilization. After that, the overhead is added, such as context-switch and DMA setup. For the contention-based system, the execution times reported in SRAM column in Table 4.5 are used to represent the worst-case execution time including the contention overhead.

Figure 4.7 shows the result of the schedulability analysis when using proposed SPM-centric OS versus a contention based SRAM system. The figure shows the results in terms of proportion of schedulable task sets for both approaches. Each point in the graph represents 1000 task sets. The results show that the schedulability of the system increases significantly when the proposed SPM-centric approach is used. Hence, the described SPM-centric OS not only improves the predictability of task execution, but it also improves task set schedulability by hiding the main memory access latency, especially for memory intensive applications.
4.3 Fault-Tolerant Scratchpad-Centric Scheduling

The ability to recover from soft errors and effectively to extend the Mean Time To Failure (MTTF) is particularly relevant when considering safety-critical systems. This is because real time embedded devices are often deployed in hostile environments such as production plants, aircraft, and satellites. In such environments, extended exposure to various kind of radiations, such as alpha particles, high and low energy cosmic rays, as well as strong electromagnetic fields, can increase the probability of temporary “bit flips”, i.e. soft errors, in the circuitry. The rate at which soft errors occur is called the Soft Error Rate (SER). The commonly used unit of measure for SER is the Failure In Time (FIT). One FIT is equivalent to one failure in $10^9$ device hours.

It has been estimated \cite{Loh2010} that the FIT/bit of SRAM memories is about 0.001, i.e. one soft error per-bit every $10^{12}$ hours of operation. Consider the platform used in our evaluation that features about 1 MB = $8 \cdot 10^6$ bits of SRAM memory including main memory and SPM. As such, each SoC will experience a single-bit soft error every $8 \cdot 10^{-6}$ hours, i.e. about 0.0002 errors per day. According to statistics about the worldwide population of automotive vehicles \cite{Statista2010}, about half a billion cars were present in 2010, with numbers expected to exceed the billion by 2020. The year 2020 also corresponds to the expected commercialization of self-driving technology, which will arguably determine an increase in the complexity of the computing infrastructure of automotive vehicles. Let us consider the current FIT for SRAM memories, and conservatively suppose that the worldwide car population does not exceed 0.5 billion. Even assuming an average daily operation time of 5%, about 5000 vehicles per day will be affect by a soft error, with potentially catastrophic consequences.

In this work, we consider the case of detectable SRAM memory errors and propose a set of OS-level strategies to recover affected real-time applications into a full operational state, without violating their timing constraints. The proposed strategies are designed and implemented on top of our previously proposed SPM-
centric OS \[13\] described in Section 4.2. Hereby, we describe how the SPM-centric OS and its analysis have been extended to recover from bit errors, in both main memory and SPM, that are detected but not corrected by the hardware logic via error-correcting code (ECC). The strategy that we follow largely leverages the existing redundancy in the employed multi-phase task model. A minimum amount of additional redundancy is introduced to protect data that do not exist as multiple copies in the original scheme.

4.3.1 Integrating Error Recovery in SPM-Centric OS

In our SPM-OS, any read-only data of a running task (in SPM) is duplicated in main memory. On top of that, two copies of the R/W portion of the task are kept inside main memory. Although, one can also keep two copies of the read-only data of the task inside main memory, this is not required because an additional copy of the read-only data is always available in flash.

The redundant copies of a task inside main memory are used to recover the system from a faulty state and allow to correctly handle one error every two periods of the same task in any of the memory modules. First, we describe how the overall system with redundant task copies works, then we explain how a fault in each of the memory modules (i.e. main memory, SPM and flash) is handled.

During normal operation of the OS, both the two copies of R/W data and one read-only copy of the task in the system are the working copies. The OS data structures are initialized with proper information about both copies as working copies and one of them is marked as currently being used. When a task becomes active on an applicative core and the TDMA slot for this core arrives on the I/O core. The I/O core first checks, if there is an empty partition available in one of the partition of the SPM. If an empty partition is available, the I/O core programs the DMA to move the task from main memory copy marked as currently being utilized into the SPM. Upon DMA completion interrupt, the I/O core sends an interrupt to the applicative core for which a load is being performed. If both of the partition are found to be full and none of the task on these partition is marked as completed, then the I/O core does nothing during this slot. However, if any of the SPM partition has a task that is marked as completed. The I/O core programs the DMA to unload the task from the SPM into main memory. Once this operation is successful, the I/O programs another unload DMA operation to download task from SPM into main memory to update the second copy of the task in main memory. An overview of the scheduling approach in case of normal operation is depicted in Figure 4.3.

Unlike what depicted in Figure 4.2, in case of memory errors, additional operations need to be performed as shown in Figure 4.8. In this case, the on-chip ECC modules detect memory errors only when a read is performed on block affected by the bit flip. Upon detection of an error, the ECC modules are programmed to (i) generate an interrupt to the application cores and to (ii) report the memory address where the error occurred. In our system, we consider the occurrence of memory errors in three different kinds of memories: main memory, SPM and flash. We now describe the proposed error recovery strategies.
Fault in SPM  Based on when a fault can be detected, the fault inside the SPM can be categorized into two types: the first case corresponds to a fault that happens in the read-only or read/write memory of the task during its execution; in the second case, a fault in read/write memory is detected during unload.

Whenever a fault is detected, all the applicative cores receive an interrupt from the error detection logic. Upon receiving the interrupt, all the applicative cores execute ISR 1 and check if the memory location that caused the error lies within their SPM memory range. Only the applicative core whose SPM was affected by the fault further executes the ISR 1. The affected core further checks if the error happened during the execution of the task or during the unload phase. This can be determined based on the SPM partition affected by the fault, since the OS keeps track of the state and location of each task.

In case the error is detected during the execution of the task, the applicative core marks the SPM partition from where the task was executing as empty and reschedules the task with the highest priority. This guarantees that the task is reloaded at the next TDMA slot for this particular core, thus improving the worst-case response time of the task as discussed in Section 4.2.4. This case is captured in ISR 1 at lines 3-7.

In the second case, the error is detected during the unload phase of the task. As previously mentioned, during an unload operation, only read/write data are copied from SPM to main memory twice. The error can occur during the first or the second redundant copy. In the first sub-case (A), the task had correctly terminated its execution phase, and the error is detected before the first copy of task R/W data from SPM to main memory is completed. In this case, the second copy in main memory is not updated, so that valid data from the previous task execution are not overwritten with faulty data. Conversely, the first (faulty) copy in main memory is marked as faulty, so that the backup copy will be used at next reload. Next, we mark the SPM partition from where the task was unloaded as empty and reschedule the task with highest priority, like in the previous case. This scenario is handled in ISR 1 at lines 9-13. In the second sub-case (B), the error is detected inside the SPM after the first copy was successfully unloaded, and while the redundant
MEMU ISR on Applicative Cores()

Algorithm 1: ISR on Application Cores From Error Detection Logic

copy was being updated. In this case, we mark the second copy in main memory as faulty and update
the OS data structures to use the first copy in main memory at next load. Since the task has successfully
completed, no task restart is required. This scenario is captured in ISR 1 algorithm at lines 14-17.

Fault in Main Memory As described earlier, there are two copies of the R/W data inside the main
memory and one copy of the read-only data inside main memory, whereas, a second copy of for the read-only
data resides in flash. An error in main memory can be detected only when a task is transferred from main
memory to SPM (load). Potentially, the fault could be directly reported to the I/O core by registering the
corresponding interrupt. For faults in main memory, however, we do not register and interrupt with the
memory error management unit. Instead, we follow a synchronous approach: at every DMA completion
interrupt, the OS checks if any error was reported by the ECC circuitry. In case of a positive outcome, the
faulty address is derived.

If the faulty address lays within the memory range being loaded from main memory, two cases are possible.
In case (A) the error affected the copy of R/W data used for the transfer, the task is not marked as “ready”
and its descriptor is updated to repeat another load at the next TDMA slot using the backup R/W data
copy. In the second scenario (B), the fault affects the read-only data of the task in main memory. In this
case, the I/O core directly copies the word that was corrupted by the error from flash to both main memory
and the SPM. No task re-load needs to be performed. The complete procedure handling cases A and B is
described in ISR 2.

DMA Completion Interrupt()

Algorithm 2: DMA completion Interrupt
Fault in Flash Memory The flash in our system is used during the bootup process. We keep two copies of the OS image in flash. At bootup, we bring the task read-only data from flash into main memory. Moreover, we also allocate the R/W data of each task in main memory. At anytime during the bootup, if an error is detected in the first working copy of the flash, we switch to the second copy of the OS image in flash. Next, we repeat the bootup procedure from the new location in flash.

4.3.2 Fault-tolerant Schedulability Analysis

Based on the description in Section 4.3.1 in this section we derive a safe bound on the worst case response time for the task under analysis \( \tau_i \). We follow the same approach detailed in [13] while focusing on proving the case of error/recovery handling, which is a special case of [13]. During the analysis we assume that \( C_i \) is the adjusted worst-case execution time of \( \tau_i \) which includes all overheads, such as the context-switch and the regular ISR handling in the applicative-core.

In this section, for simplicity, we discuss the case in which only one memory error can occur in two consecutive period of any task. However, the analysis could be easily extended to account for more frequent errors and with more than \( M = 2 \) cores.

Figure 4.9 depicts an illustrative example of the worst case scheduling scenario (critical instant at time \( t = 2 \) and following busy interval) for an example task set where \( \tau_3 \) is the task under analysis. The schedule depicts a busy period where \( \tau_3 \) suffers interference from two higher-priority tasks, \( \tau_1 \) and \( \tau_2 \). As in [13], we consider the busy period as composed by a sequence of scheduling intervals \( \text{Interval}_2, \text{Interval}_3, \text{Interval}_4 \) (each bounded by bold vertical lines in the figure), followed by a final interval \( \text{Interval}_F \). During each scheduling interval, only one blocking or interfering task runs. During the final interval, the task under analysis runs. Each scheduling interval always starts with a CPU execution and ends either when the CPU
finishes executing the task or when the next task finishes being loaded by the DMA, whichever happen last; at this point, the next interval starts with the execution of the loaded task. The final interval starts with the execution of the task under analysis and finishes when the task under analysis is unloaded.

We say that a scheduling interval is CPU-bound when it ends with CPU execution (ex: Interval$_1$, Interval$_3$ and Interval$_4$ in the figure), and I/O-bound when it ends with DMA load operation (ex: Interval$_2$). The length of a scheduling interval is the maximum between the execution time of the task running in the interval and the DMA operations required to load the next task. We denote the size of the TDMA slot as $\sigma$; in the worst case a load/two unload operations can occupy the entire slot, including the I/O-core ISR2 DMA handling which might copy from/to the secondary copy in SRAM, Flash and the SPM. For this reason, We upper bound the length of DMA operations as a multiple of $\sigma$.

Response Time Calculation
Building on the above-mentioned definitions, we can follow the same technique detailed in [13] to compute the worst case response time for a task under analysis $\tau_i$ ($\tau_3$ in Figure 4.9). In [13], the response time of $\tau_i$ is computed by adding three components: (1) the blocking time $B$ caused by a lower priority task that starts executing before the beginning of the busy interval; this is Interval$_1$ executing task $\tau_5$ in the figure; (2) the interference $H$ comprising the remaining scheduling intervals in the busy period, which are Interval$_2$, Interval$_3$ and Interval$_4$ in the figure. The number of such intervals is equal to the number of interfering higher priority jobs plus one, since an extra lower priority job that starts loading before the beginning of the busy period ($\tau_4$ in the figure) can execute within the busy period itself; (3) the worst case length $F$ for the final interval Interval$_F$ during which the task under analysis is executed, up to the finish time for the unload operation of $\tau_i$. Therefore, the response time of the task under analysis is $R_{\tau_i} = B + H + F$; since the length $H$ of the interfering intervals depends on $R_{\tau_i}$, this is computed using a standard iterative method. In particular, notice that the number of interfering higher priority jobs is computed based on $R_{\tau_i} - F$ rather than $R_{\tau_i}$: once $\tau_i$ starts executing in Interval$_F$, newly arriving higher priority jobs cannot delay its execution anymore.

As proved in [13], the critical instant is produced when the task under analysis $\tau_i$ and all higher priority tasks arrive immediately after a lower priority task has started loading into a partition, and the other partition was loaded with another lower priority task as late as possible (i.e., two slots before). Based on the critical instant, we then obtain $B = \max(Ci, 2 \cdot \sigma) - \sigma$, where $\tau_l$ is the lower priority task with the largest execution time. Finally, based on Lemma 3 in [13], the maximum length of the final interval is $F = \max(Ci + 5 \cdot \sigma, 7 \cdot \sigma)$.

Compared to [13], our solution differs as it accounts for the possible memory error/recovery that might lead to a task reschedule. We show that we can use the same response time iteration as in [13] to calculate the response time of the task under analysis after extending $H$ or $F$ depending on when the memory error/recovery takes place.
**Accounting for Error Recovery**  Since we assume that no more than one error can occur for any two consecutive periods of any task, it follows that during the busy interval of the task under analysis $\tau_i$, there can be at most one task that suffers one error. A failed task is then rescheduled within bounded time.

**Lemma 7** A failed task that is rescheduled with highest priority will be reloaded after at most $M \text{ TDMA slots}$.

*Proof:* Since the failed task will be raised to be the highest priority task in the system and the load has priority over unload, it is guaranteed to reload the failed task in the same partition during next TDMA slot of the corresponding core, which is once every $M$ slots. Therefore, the failed task is guaranteed to be reloaded after $M \text{ TDMA slots}$, 2 in the case of 2 cores as shown in Figure 4.9.

At the task level, the error might occur during the load, execute, or the unload of the task. However, to produce the worst-case workload induced by a task to the schedule, the memory error must happen as late as possible during the unload of the task.

**Lemma 8** A task generates the worst-case workload induced into the busy interval when the memory error occurs as late as possible, i.e. during the unload phase.

*Proof:* There are three cases in which the error can occur, (1) during the load, (2) during the execution, and (3) during the unload. Case (1) does not incur any extra overhead, since the error is recovered by ISR2 on the I/O core and the overhead is included in the TDMA slot size. However, in case (2), the execution of the task is aborted (hence partially wasted) and the task has to be rescheduled to load again after $M \text{ TDMA slots}$. Finally, in case (3) the task is fully executed and unloaded before being rescheduled and reloaded after $M \text{ TDMA slots}$; since this case results in the most (wasted) time added to the busy interval, it is the worst case.

Based on Lemma 8, we assume that a memory error always occurs as late as possible during the unload phase of a task to capture the worst case.

At the schedule level, we classify the memory error/recovery based on when it occurs with respect to the task under analysis, (1) prior to the final interval in which the task under analysis runs or (2) during the final interval.

**Error Recovery Prior to The Final Interval (Interval $F$)**

**Lemma 9** For an error that occurs prior to $F$, adding an extra interfering interval to $H$ executing the task in the system with the largest execution time other than $\tau_i$ leads to the worst case response time for the task under analysis.

*Proof:* By definition, the error has to be in $H$ or $B$ to affect the response time of the task under analysis. Based on Lemma 8, the error should occur during the unload of a task; hence, the recovery mechanism forces
the failed task to be rescheduled, i.e., a new scheduling interval is added to the scheduled. Furthermore, regardless of the tasks priority, the rescheduled (failed) task always runs as the highest priority in the system, thus this additional scheduling interval causes interference to the task under analysis.

Since we cannot make any assumption on which task might fail, lower-priority task or higher-priority task, it is safe to assume that the longest executing task in the system other than the task under analysis will be scheduled to run in the induced interval. Finally, note that even if the task that fails is the one executed in $B$, the rescheduled task will execute during $H$. Since the algorithm in [13] is able to correctly upper bound the interference in $H$ caused by any task, we then simply add the induced interval to $H$ to capture the worst-case response time for $\tau_i$.

Based on Lemma 9, let $H_{rec}$ be the computed length of interfering intervals including one restarted task.

**Error Recovery in The Final Interval ($Interval_F$)**

**Lemma 10** For an error that occurs within $Interval_F$, the maximum length of the interval with $M = 2$ is $F_{rec} = 2 \cdot \max(C_i + 5 \cdot \sigma, 7 \cdot \sigma) + \max(C_u, 4 \cdot \sigma) - 2 \cdot \sigma$, where $\tau_u$ is the task in the system with the largest execution time other than $\tau_i$.

*Proof:* As defined earlier, $Interval_F$ starts with the execution of the task under analysis $\tau_i$ and finishes with the end of the unload phase of $\tau_i$. Based on Lemma 8 in the worst case, the error occurs during the unload phase. Therefore, the error must occurs in the unload phase of $\tau_i$, otherwise $\tau_i$ will unload successfully and the interval finishes.

Based on Lemma 3 in [13], in the normal case, the unload operation for $\tau_i$ completes after at most $\max(C_i + 5 \cdot \sigma, 7 \cdot \sigma)$ time units from the beginning of the interval; in Figure 4.9 this occurs at time 27. However, in the case of memory error during the unload operation and task recovery, $Interval_F$ is extended. To simplify the computation of the worst-case length $F_{rec}$ of $Interval_F$ accounting for recovery, we divide the interval in three sub intervals. The first execution of $\tau_i$ is contained in the first sub interval $SubInterval_1$, which finishes with the first (failed) unload of $\tau_i$. The last sub interval $SubInterval_2$ starts with the second execution of $\tau_i$ after the reload and ends with the (successful) second unload, time 37 in the figure. In the worst case, there can be a middle interval $SubInterval_u$ in which another task $\tau_u$ is loaded into the other partition and executed.

Based on Lemma 7, $\tau_i$ will be reloaded after $M$ TDMA slots (2 in our case). $SubInterval_u$ finishes and $SubInterval_2$ starts either when the reload operation is complete (case shown in the figure) or when $\tau_u$ finishes, whichever happens last. Since in the worst case $SubInterval_u$ starts after $\tau_u$ has been loaded, and we need $M$ TDMA slots for the failed unload and $M$ TDMA slots for the reload of $\tau_i$, the length of $SubInterval_u$ can be upper bounded as $\max(C_u, 4 \cdot \sigma)$ when $m = 2$. The lengths of the sub intervals $SubInterval_1$ and $SubInterval_2$ are calculated the same way as in Lemma 3 in [13] as discussed above. However, as shown if Figure 4.9 $SubInterval_1$ and $SubInterval_U$ overlap by two TDMA slots (time 25 to
27 in the figure). This is because the length of SubInterval\textsubscript{1}, as in Lemma 3 in \cite{13}, is calculated up to the end of the unload phase, while SubInterval\textsubscript{U} starts \(M = 2\) slots before. To overcome this overlap, we subtract the overlapped time which is \(2 \cdot \sigma\) when \(M = 2\).

As a result, the length of Interval\textsubscript{F} is computed as:

\[
F_{\text{rec}} = \text{SubInterval}_1 + \text{SubInterval}_2 - 2 \cdot \sigma
= \max(C_i + 5 \cdot \sigma, 7 \cdot \sigma) + \max(C_u, 4 \cdot \sigma)
+ \max(C_i + 5 \cdot \sigma, 7 \cdot \sigma) - 2 \cdot \sigma,
\]

which is the same as the value in the hypothesis.

Finally, we have to account for the overhead of executing the recovering ISR 1 on the applicative core during the busy interval. If we let \(\rho\) be the maximum length for the ISR, we can compute a safe upper bound by simply adding \(\rho\) to the response time iteration. In general, we do not know which case will lead to the worst response time calculation for \(\tau_i\), when the error occurs before Interval\textsubscript{F} (case 1) or within Interval\textsubscript{F} (case 2). As a result, we independently calculate both iterations:

\[
R_{\tau_i}^1 = B + H_{\text{rec}} + F, \quad (4.1)
\]

\[
R_{\tau_i}^2 = B + H + F_{\text{rec}}, \quad (4.2)
\]

and take the maximum response time among \(R_{\tau_i}^1, R_{\tau_i}^2\). In particular, note that it is easy to see that \(F_{\text{rec}} - F\) is larger than the size of the additional interval added for case 2. This is the main reason why we chose to rescheduled failed tasks at the higher priority: it minimizes the worst case length of Interval\textsubscript{F} in case the task under analysis fails. On the other hand, rescheduling the task at higher priority means that we have to consider any task, rather than just higher priority tasks, for the extra interval in case 2, but as discussed this is generally not the worst case. However, note that we cannot formally avoid computing the iteration in Equation 4.1 because the interfering window for higher priority jobs is based on \(R_{\tau_i}^1 - F = B + H_{\text{rec}}\), which is larger than \(R_{\tau_i}^2 - F_{\text{rec}} = B + H\).

### 4.3.3 Implementation

In this section, we will describe the implementation of the recovery mechanism for SPM-centric OS using component-off-the-shelf (COTS) MPC5777M embedded platform. The architectural features of the considered SoC are compliant with the hardware assumptions made in Section 4.1 and summarized in Table 4.3.

Apart from the features highlighted in Section 4.2.5, the considered platform features ECC logic imple-
The ECC modules provide single-bit error correction and double-bit error detection (SEC-DED). ECC is implemented with Hsaio Codes for detection and correction, this means that both correction and detection can be simultaneously done. The chip also implements fake error injection mechanisms that are helpful to verify the reaction to various faults. We use these fault injection mechanisms to evaluate our system, as described in Section 4.3.4.

**OS-level Integration of Recovery Mechanisms**  
As previously mentioned, SPM-centric OS was implemented using Evidence Erika Enterprise[^1]. The implementation was extended to incorporate the described memory error recovery strategies. In the considered MPC5777M platform, there is a memory error management unit (MEMU) that is responsible for collecting and detecting the faults in different memory subsystems such as SRAM, SPM and Flash. The MEMU implements separate tables for reporting correctable and uncorrectable errors. There are separate tables for each kind of memories. These tables contain the address of the fault that caused error, moreover, there is a register inside the MEMU that tells if the fault that occurred is a correctable or uncorrectable fault. On MPC5777M, there is no way for the MEMU to send an interrupt to the CPU in case of a fault. There is a separate FCCU module present on the chip that collects all the errors that are forwarded to it from the MEMU. The FCCU module can be preprogrammed to take certain actions based on a particular error. Moreover, it is also responsible of generating interrupt to the processor to notify it in case any kind of errors that are being reported to it from the MEMU. Figure 4.10 shows how different modules are connected to each other.

In order to detect faults in SPM, we register a FCCU interrupt with application cores. This interrupt gets generated when an error is reported by the MEMU to the FCCU in one of the memory subsystem. Upon interrupt generation, each application core receiving the interrupt check if the address that caused the error falls within the address range of the local SPM. If not, the application core exists the ISR and resumes its normal operations. In case the address corresponds to a local SPM location, we further check if the fault

[^1]: http://erika.tuxfamily.org/drupal/
Table 4.6: Details of OS Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA Load time (Largest Code, R and R/W Data size)</td>
<td>209</td>
</tr>
<tr>
<td>DMA Unload time (Largest R/W Data size)</td>
<td>61.5</td>
</tr>
<tr>
<td>DMA setup</td>
<td>3.16</td>
</tr>
<tr>
<td>Context switch</td>
<td>0.46</td>
</tr>
<tr>
<td>Minimum ISR overhead on Applicative cores</td>
<td>0.70</td>
</tr>
<tr>
<td>Maximum ISR overhead on Applicative cores</td>
<td>8</td>
</tr>
<tr>
<td>Maximum ISR overhead on I/O core for DMA completion</td>
<td>2</td>
</tr>
<tr>
<td>TDMA slot size</td>
<td>215</td>
</tr>
</tbody>
</table>

is in the read-only or read-write data section(s) of the task. (A) In case of errors in read-only memory, the error is detected synchronously when the faulty instruction/constant data is being read. In this case, the core copies the data-word that caused the error from the copy in main memory and re-executes the faulting instruction again, as if the error was a recoverable segmentation fault. (B) For errors in writable sections, the application core marks the partition as empty and reschedules the faulty task. This procedure only applies for errors in SPM. Errors in main memory and flash are handled by the I/O core and they follow a procedure that is essentially what discussed in Section 4.3.1.

4.3.4 Evaluation

In order to validate the results of the proposed error recovery mechanisms, we performed a variety of experiments. We measured the overheads of different recovery handling routines to account for the OS overhead. We then present the results of the EEMBC benchmarks by running the tasks from the SRAM verses SPM. Based on the collected data we then calculate schedulability graphs.

**SPM-Centric OS Overhead Evaluation**  Just like SPM-OS without error recovery capabilities, one of the most important parameters in our system is the size of the TDMA slot. The slot needs to be long enough to account for the load/unload of any task in the system. In addition, the TDMA slot size must be long enough to allow two unloads of the completed task. Therefore the TDMA slot size is equal to $\max(\text{worst}\_\text{task}\_\text{load}, 2 \cdot \text{worst}\_\text{task}\_\text{unload}) + \text{I/O}\_\text{core}\_\text{ISR}\_\text{overhead}$. Table 4.4 shows the system parameters including DMA times and TDMA slot size.

We have measured the DMA modules configuring overhead. In addition, since the interrupt from the memory error management unit is delivered to all the application cores, we have measured the overhead of ISR1. This ISR has minimum and maximum overhead. The minimum overhead occurs when the ISR only checks if the error address is relevant or not and concludes by its irrelevance. On the other hand, the maximum overhead is experienced in the case of a relevant memory fault. We also measured the maximum overhead of I/O core ISR2 for DMA completion interrupt. All of these results are reported in Table 4.4.
Results of EEMBC Benchmarks  In order to evaluate our system, we run EEMBC benchmarks (automotive suite) on the target platform. We compared the execution time of running the reported applications in Table 4.5 out of the local SPM and the main SRAM memory. When the applications run from the main memory they suffer contention delay due to the shared access to the main memory, refer to Section 4.2.6 for more details about the comparison. Here we have updated the benchmark table to explicitly mark the sizes of the read-only data and read-write data of the applications as it is relevant to our recovery mechanism. The results are shown in Table 4.7.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SPM Time (µs)</th>
<th>SRAM Time (µs)</th>
<th>Relocatable Code Size (bytes)</th>
<th>Read only Data Size (bytes)</th>
<th>Read/Write Data Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tblook</td>
<td>1013</td>
<td>1015</td>
<td>1892</td>
<td>10916</td>
<td>60</td>
</tr>
<tr>
<td>matrix</td>
<td>1053</td>
<td>1054</td>
<td>4774</td>
<td>12188</td>
<td>124</td>
</tr>
<tr>
<td>a2time</td>
<td>1002</td>
<td>1029</td>
<td>2538</td>
<td>1704</td>
<td>148</td>
</tr>
<tr>
<td>pntrch</td>
<td>1036</td>
<td>1145</td>
<td>1398</td>
<td>4800</td>
<td>128</td>
</tr>
<tr>
<td>ttspkr</td>
<td>383</td>
<td>425</td>
<td>4772</td>
<td>2592</td>
<td>48/48</td>
</tr>
<tr>
<td>iirft</td>
<td>1040</td>
<td>1189</td>
<td>3512</td>
<td>888</td>
<td>248</td>
</tr>
<tr>
<td>candr</td>
<td>1009</td>
<td>1359</td>
<td>1562</td>
<td>12276</td>
<td>56</td>
</tr>
<tr>
<td>bitmp</td>
<td>990</td>
<td>1389</td>
<td>3282</td>
<td>72</td>
<td>1494</td>
</tr>
<tr>
<td>rspeed</td>
<td>1012</td>
<td>1457</td>
<td>1208</td>
<td>13200</td>
<td>40</td>
</tr>
<tr>
<td>puwm</td>
<td>1036</td>
<td>1540</td>
<td>2500</td>
<td>2400</td>
<td>180</td>
</tr>
<tr>
<td>aifirf</td>
<td>1005</td>
<td>1564</td>
<td>2286</td>
<td>1120</td>
<td>84</td>
</tr>
<tr>
<td>aifirr</td>
<td>916</td>
<td>1642</td>
<td>4458</td>
<td>2304</td>
<td>1912</td>
</tr>
<tr>
<td>aififf</td>
<td>1170</td>
<td>2092</td>
<td>3540</td>
<td>3072</td>
<td>1656</td>
</tr>
<tr>
<td>idct</td>
<td>1045</td>
<td>2126</td>
<td>4690</td>
<td>244</td>
<td>1788</td>
</tr>
</tbody>
</table>

Results of Schedulability Analysis  For the schedulability evaluation of our newly proposed scheme we first present the schedulability curve of normal case when there are no errors (i.e. Figure 4.7) and compare it with the case when we have errors. We also show the contention based approach where we have no error recovery. The case referred as “contention” corresponds to the case where no scratchpad management is implemented and in which tasks execute directly from SRAM contending for bus access.

We computed the response time of the same workload for the three cases: the traditional SPM-centric OS mechanism with no errors as proposed in [13]; the augmented SPM-centric OS with error recovery mechanisms using the analysis in Section 4.3.2 and with artificial error injection; and the contention based execution using standard response time analysis. For the evaluation, we have considered the EEMBC benchmarks in Table 4.7 and the overheads in Table 4.6.

For a given system utilization, each application is randomly selected and assigned a random period in the range between 10 ms to 100 ms. The tasks utilization is then computed based on the measured execution time of the applications and the selected period. Tasks are randomly generated until the sum of the individual tasks utilizations reaches the required system utilization.
Figure 4.11 shows the result of the schedulability analysis for our newly proposed SPM-centric scheme with error recovery and depicts a comparison with the case where no error recovery is performed. Based on the figure, we can conclude that there is limited degradation in schedulability for supporting the recovery mechanism. This degradation can be justified by the fact that the system is both predictable as well as fault tolerant. Moreover, from the Figure 4.11 we can also see that our SPM-centric approach with error recovery still outperforms the contention-based case where no error recovery is performed.

![Figure 4.11: Schedulability degradation with fault recovery mechanism.](image)

4.4 Multi-stage Scheduling and Optimization

One of the main drawbacks of multi-stage task models is the complexity of the resulting co-scheduling problem. Intuitively, since additional knowledge is available for each task, the number of dimensions to consider at scheduling time also increases. It follows that the problem of optimally scheduling multi-stage tasks is often intractable even for a small number of CPUs/DMAs involved.

In general, the class of scheduling problems for multi-stage tasks that execute on an ordered sequence of resources takes the name of “flow-shop scheduling”. This class of problems has been largely studied since the early ’50s because is also relevant to schedule resources and assembly phases in production plants. The problem of selecting the optimal schedule for flow-shop jobs with more than two stages, however, has been proven to be NP-hard in [189]. In [190], we focus on flow-shop tasks characterized by two stages and generalize the task model used in [13], such that: (i) each task stage requires to be executed on a specific type of resource, and (ii) one resource of each type exists in the system.

When two-stages real-time tasks with same rate and deadline are considered, Johnson’s algorithm [191]
provides an optimal solution when only one instance of each resource exists in the system. However, the properties of different scheduling strategies when these assumptions are relaxed is yet to be well understood. We are currently studying the behavior of some heuristic scheduling algorithms for special relaxations of the initial setup.

In [190] we introduce a novel and efficient (polynomial-time) algorithm that derives the optimal speed of resources, either memory, CPU or both, when single-rate periodic tasks that run across two stages of single-unit resources are considered. The selected speed allows optimizing power usage while ensuring that schedulability constraints are met. As a part of our future work we intend to provide additional insights about the hardness of two-stage flow-shop problem in the more general setting with intermediate deadlines. Additionally, we intend to discuss extensions to multiprocessor systems and to perform a quantitative evaluation of the proposed approaches.

4.5 Summary

The advantage of scratchpad-based platforms is that not only they provide multiple processing elements, just like cache-based architectures, but a certain degree of parallelism is provided for memory resources too. As mentioned in Section 2.1.2, the parallelism in memory arises from advanced multi-ported or high-bandwidth SRAM memories, or by coupling different local scratchpad memories on a per-core basis. Typically however, scratchpad memory has a high per-bit cost and its size directly impact the chip area. As a result, scratchpad memories are relatively small in size. It follows that being able to appropriately manage scratchpad resources is key to achieve predictability and optimize resource utilization on COTS scratchpad-based architectures.

In this chapter, we have proposed to re-design a real-time OS to specifically exploit scratchpad memory resources. A proof-of-concept implementation of the new design, namely SPM-centric OS, using a commercially available multi-core platform was also performed to assess the feasibility of our approach. In order to achieve this goal, we combined resource specialization, high-level scheduling of shared hardware resources as well as a three-phase task execution model. Theoretical results on how to perform schedulability analysis of the proposed scheduling strategy were presented. In order to validate the OS design and implementation, we have combined experimental results from synthetic and automotive EEMBC benchmarks on the considered platform. In addition to the strong temporal predictability achieved by enhancing inter-core isolation, we are able to exploit the performance benefits of scratchpad memories. Hence, a schedulability improvement over traditional contention-based approaches was obtained.

Predictability and performance optimization alone are not enough to ensure practical adoption in industry. This is because the hardware complexity of multi-core platforms is significantly higher compared to single-core microcontrollers. As such, they exhibit higher soft error rates. Regulations (see Section 2.5) in the automotive and avionics domain impose strict requirements for functional safety. Thereby, the existence of appropriate safety nets to counteract the increased risk of hardware errors, and to achieve large-scale
reliability and robustness is a fundamental requirement in next-generation safety-critical systems.

We have presented error recovery mechanisms to correct detectable hardware bit-flips that cannot be recovered by platform ECC modules. We integrate these strategies into our SPM-centric OS. Since SPM-centric OS uses a multi-phase copy-execute scheme for scheduling real-time tasks, it already embeds a certain degree of redundancy, because several copies on the same task memory can exist in the system. The integrated error recovery techniques largely rely on existing redundancy and only introduce a limited amount of extra redundancy. We have discussed how the original analysis can be extended to take into account different sources of overhead and task re-executions to analyze the schedulability of real-time tasks in spite of errors and recovery procedures. Our evaluations highlight that the loss in schedulability arising from the additional recovery-induced workload is acceptable considering the gain in robustness.

4.6 Future Work

A number of challenges require to be addressed to bring mature the proposed OS-level resource management scheme and for its adoption on a wide scale. Inter-process and inter-core communication represents an important feature that needs to be carefully designed and implemented to integrate seamlessly within the scratchpad management scheme. On one hand, inter-core communication opens a number of challenges because commonly used standards (e.g., AUTOSAR) often define a number of communication models. Each model would require a careful design and important integration efforts. On the other hand, inter-core/process communication already represents a structured interaction between system entities, because it is typically performed via a set of Application Programming Interfaces (APIs). Thus, it is possible to intercept API calls and implement appropriate intermediate communication logic.

Being able to lift the current restrictions on the size of applications represents another important goal for future research. Currently, the footprint of any real-time task needs to be entirely contained within a local core scratchpad. Even considering that future platforms will feature larger scratchpad memories, the size of applications can easily outweigh SPM memory size. A possible approach consists in using compiler-level techniques to divide the task into a sequence of execution intervals. For each interval, the data required for its execution shall be known, so that it could be loaded using before the execution of the interval onto the SPM.

One of the most important characteristics that sets the automotive domain apart from classic real-time systems is the presence of adaptive variable-rate (AVR) tasks. AVR tasks feature activation times that depend on hardware-related events (e.g., engine revolutions). Such tasks not only vary their period depending on the physical system’s state, but also exhibit variations on the amount of computation/memory required. We intend to extend our RTOS design so to allow the definition of AVR tasks. This includes extending meta-data and scheduling algorithms to account for the new class of tasks.

Finally, a required extension is the development of an accurate end-to-end delay analysis for the developed
predictable I/O subsystem (see Section 4.2.3). In the current design, input traffic at the peripherals is asynchronously received at (transmitted from) the I/O Core and synchronously transmitted to (received at) the application cores. Two questions are thus important: (i) what is the volume of traffic that can be processed at the I/O Core without experiencing information loss, given constraints on channel bandwidth, I/O scratchpad size and DMA slot length? and (ii) how can we model and analyze the timing properties of a data-stream when several platforms, each running an instance of SPM-centric OS, co-operate on a local network?
CHAPTER 5

WORKLOAD PROFILING

Modern multi-core architectures are characterized by heavily parallelized computational resources and a mostly centralized memory hierarchy. It follows that being able to efficiently manage the components of the memory hierarchy is a crucial aspect to address toward practical resource regulation and control. In order to attain this goal, two are the main requirements: (i) a good understanding about the functional and timing behavior of the underlying hardware components; and (ii) some degree of knowledge about the behavior of the tasks under analysis. The former can be acquired via the hardware manufacturer in the form of documentation or through platform benchmarking and represents a required per-platform knowledge base. Conversely, the latter is strictly dependent on the characteristics of the application workload and thus needs to be acquired on a per-task basis.

In this chapter, we briefly review aspects and methodologies for application workload profiling, i.e. the process of extracting behavioral knowledge out of existing applications. The problem of determining the memory access pattern of a task under analysis for workload characterization purposes has been approached from several perspectives, resulting in solutions with different advantages and trade-offs, as we briefly discuss in Section 2.7. In our work, we focus on profiling performed through the observation of live execution of the target program.

First, we introduce a novel mechanism to perform memory access detection on general purpose applications that can be used on COTS platform with standard hardware support. Next, we discuss how profiling information to drive efficient cache allocation can be extracted from memory access traces. Finally, we discuss how automatic code re-factoring can be performed by exploiting memory traces. This allows us to re-organize the memory access pattern of a task to make it suitable for resource control strategies.

5.1 Extracting Application Behavior

In [15] we propose a novel technique to perform memory access detection that relies on memory protection mechanisms, namely MadT. The methodology and proposed implementation is aimed at demonstrating how it is possible to acquire accurate memory traces by introducing controlled memory faults and trap exceptions. In this way, we (1) do not rely on advanced debugging hardware capabilities, (2) do not need to instrument any of the instructions in the observed binary, (3) do not necessarily need to instrument the source code: in
fact, we can directly perform memory tracing of a binary executable by dynamically linking a shared library containing MadT’s code.

MadT is capable of recording memory accesses performed by an application in a fully symbolic form. In addition, our tool is capable of accurately tracking dynamic memory allocations and releases, forging when necessary new symbol names that simplify the process of identifying accesses to dynamic memory locations. By using MadT the user can accurately relate all memory accesses, including those to dynamically allocated blocks, to the source code of the application under analysis. Compared to other profiling tools, MadT shows similar performances, and in some cases it is significantly faster. The main advantage of MadT over traditional approaches such as source-code level instrumentation and dynamic binary instrumentation are the following: (i) it is minimally intrusive on the memory layout of the program under analysis; (ii) its implementation is lightweight with a very small portion of the code being platform-specific. Hence, it can be ported easily to all those platforms that provide memory protection and basic code debugging primitives.

5.1.1 Methodology Overview

MadT is basically a shared library that is linked to the target program at compilation time. Whenever the target program is executed, the code in MadT’s library sets up the execution context so that data memory accesses are detected and properly resolved. MadT is based on three basic components:

A) A profiler that detects memory accesses by natively executing the target program.

B) A symbolic resolver that finds the symbolic name associated with any numerical address used by the target program.

C) A dynamic memory tracker that intercepts the calls to the C library functions related to dynamic memory handling.

MadT Profiler The profiler is the component of MadT devoted to detect and record the data memory accesses performed by the target program. MadT does not emulate or instrument the machine code of the target program. Rather, MadT sets up the target process so that it executes natively and self-detects its memory accesses. This approach turns out to be particularly convenient for programs in which the number of machine instructions that load or store values in the memory cells is a small fraction of the total count of executed instructions.

The general idea is to set up the target process so that any access to data in a target’s memory page shall result in a page fault. Consequently, the program receives a SIGSEGV signal, and the corresponding signal handler records the address of the machine instruction that tempted the access, the type of access (load or store), and the address of the memory cells being accessed. The mechanism for generating a page fault for
any access consists of removing any access right from the memory pages containing the data of the target program.

Once the memory access has been recorded, the target program should resume and execute again the machine instruction that triggered the fault. This time the instruction must not cause a page fault, otherwise we would be in an infinite loop. Therefore, the handler of the SIGSEGV signal restores the proper access rights to the accessed memory pages. The problem is that any further access to the same pages performed by the following machine instructions would then go undetected. It is thus necessary to remove the access rights immediately after the single machine instruction that triggered the page fault has been successfully terminated. In order to achieve this goal, the page fault handler of the SIGSEGV signal modifies the hardware context of the target process saved on the stack so as to set the hardware “trap flag”. When the trap flag is set, the CPU control unit generates a trap exception for each machine instruction that terminates. Right after the SIGSEGV handler terminated, the target’s machine instruction that accessed the memory is executed again, this time without generating page faults. Then, the CPU control unit raises the trap exception. The OS kernel reacts to the trap exception by sending a SIGTRAP signal to the target process. The MadT's handler for the SIGTRAP signal removes the access rights from the pages, and modifies the hardware context of the target process saved on the stack so as to clear the “trap flag”. When the target process resumes its normal execution, instruction trapping is disabled and all data pages are “protected” again.

MadT’s profiler currently exploits a couple of features of the x86 architecture. However, we consider it to be quite portable. On one hand, any general-purpose CPU coupled with a Memory Management Unit (MMU) must support memory protections and page faults. On the other hand, the trapping instruction mechanism is present at least in any architecture based on a CISC ISA, because it makes much easier to implement debuggers and monitors. The trapping mechanism is also present in several RISC-based architectures: prominent examples are the various ARM architectures.

Finally, observe that because the MadT profiler operates at the machine instruction level, it is not difficult to extend it so as to support target programs written in high-level languages different than C.

**MadT Symbolic Resolver** One distinctive feature of MadT is that it can output the memory accesses in a symbolic form. Usually, each time a program is executed, the virtual memory addresses used by the process change. Basically there are two reasons for this behavior:

1. In each run, the kernel loads the segments of the shared libraries used by the process at different addresses. This mechanism is called “address space layout randomization” and is essentially aimed at making more difficult to exploit security flaws in the program. Although address space layout randomization can usually be disabled by the user, the effective load addresses of the shared libraries may still change between different kernel versions, dynamic loader versions, or shared libraries versions.

---

1. In a RISC CPU all machine instructions have opcodes of fixed length. Moreover, very few instructions access memory. Thus, even if the trapping mechanism is not available at the hardware level, it is not too hard to devise a mechanism that execute the very single instruction that caused the page fault, and then resumes normal execution.
2. The virtual addresses returned by dynamic memory allocator procedures cannot be easily predicted in advance and may change at each run. The C standards explicitly mention that this behavior is allowed and expected.

Thus, a memory access detection tool that returns only a list of numerical addresses is not very practical, because the user faces the difficult task to relate those addresses with the corresponding function and variable names of the target program and of the shared libraries.

Instead, in MadT each recorded memory access yields one line in the output file that includes the symbolic names of the function, variable, or dynamically allocated block of the target program. Actually, MadT forges a new symbolic name for each dynamic memory block allocation (see below). In the other cases, MadT associates any virtual address with the corresponding symbol as known at compilation time or at dynamic loading time.

To fully enable the address resolution mechanism, the target program must be recompiled in such a way to include debugging information in the executable file. It is thus possible to extract from the executable file some tables that relate virtual addresses with symbolic names (even for the symbols that are “private” to the target program). These tables are read by MadT when the target program is launched. Moreover, MadT translates virtual addresses related to shared libraries functions and variables by invoking suitable helper functions of the dynamic loader. The dynamic loader is included in the memory space of any process being executed and deals with shared libraries loading and shared symbols resolving.

MadT’s symbolic resolver is quite portable across different architectures and even across different operating systems. It relies on the GCC toolchain [192] and on the dynamic loader coupled with the “glibc” GNU C library [193], thus any architecture to which these programs have been ported to is basically supported by MadT’s symbolic resolver.

Extending the symbolic resolver to support target programs written in high-level languages different than C is also feasible, as long as two main requirements are satisfied: a) it must be possible to instruct the compiler to insert debugging information in the target executable; and b) the target program must use the same system dynamic loader used by C programs. Typically, programs written in high-level languages supported by the GCC toolchain satisfy these requirements.

**MadT Dynamic Memory Tracker** Virtual addresses returned by dynamic memory allocators have no symbols associated with them, neither at compile time nor at run time. To overcome this limitation, MadT forges a new symbol for any dynamic memory allocation.

The name of any forged symbol has a structure that helps the user to recognize the kind of allocation and the exact position of the call to the allocator procedure in the target’s source code. For instance, if the 14th dynamic memory allocation of the target program has been triggered by a call to the malloc() C library procedure, and the call is included in some function funcI() of the target program, then the forged symbol name would be something like “malloc0014@funcI+230>”. Here, “230” represents the offset in bytes of
the call to `malloc()` inside `funcI()`. Observe that the role of the counter is to distinguish between different allocations performed by the same allocator call; for example, consider a loop iterated several times and containing an invocation to `malloc()`.

In order to forge new “dynamic” symbols to be associated with dynamically allocated memory blocks, MadT has to detect when the target process invokes a C library procedure that allocates memory for the process itself. This is simply done by defining a set of “wrapper” functions in MadT that are invoked in place of the “original” C library procedures. Currently MadT install wrappers for the C library functions `malloc()`, `calloc()`, `realloc()`, `free()`, `mmap()`, `mremap()`, and `munmap()`.

Any wrapper records internally the dynamic memory event so that “dynamic” symbols can be properly forged. The dynamic event is also written in the output file produced by MadT. The wrapper function also takes care of invoking the “original” C library procedure.

The wrapper mechanism used by MadT is rather portable because it is based on features provided in all modern operating systems. Basically, whenever a process invokes a given procedure defined in a shared library, the dynamic loader looks for a matching procedure name starting from the first loaded library. The trick, therefore, is to load MadT’s shared library before any other shared library used by the target program. The dynamic memory tracker of MadT can thus be easily ported to any platform that allows the user to override the loading order of the shared libraries. Currently, MadT supports Linux systems, where the loading order of the shared libraries can be controlled, for example, by defining some environment variables.

The dynamic memory tracker can also be extended to support programs written in high-level languages different than C. For example, in order to support a C++ target program, new wrappers for C++ library procedures that handle dynamic memory, like the `new` operator, should be included in MadT’s shared library.

**Limitations**  It is important to note that currently MadT does not trace memory accesses caused by instruction fetches. Although the same mechanism used to trace data accesses can also be applied to trace instruction fetches, the performance impairment would likely be very high. Moreover, from a practical standpoint, individually tracing instruction fetches is not necessary due to the inherently sequential nature of code execution flow. Conversely, an analysis based on basic blocks is sufficient and many tools have been described in literature to perform control-graph extraction either at runtime or offline.

Currently, MadT does not trace single memory accesses to the stack. This is mainly because, similarly to instructions fetches, stack usage can be easily analyzed without performing tracing at runtime. In fact, stack boundaries are known at compile-time and can also be extracted by parsing the executable file. Nonetheless, MadT already detects memory block operations (e.g., `memcpy`, `memset`) that access the stack, and we plan to extend MadT to include stack single accesses detection in a future release of the tool.

Finally, MadT does not provide a component to analyze the collected trace. The main reason is that MadT is meant only to provide a method to detect memory accesses and to map them to the original variables defined by the programmer. Moreover, the output file produced by MadT can be easily converted in the
format of other tools like Valgrind \cite{valgrind} and OProfile \cite{oprofile}. Therefore, any graphical analyzer suitable for the mentioned tools can also be used for MadT.

5.1.2 Implementation Details

The implementation of MadT relies on many different techniques, which will be discussed in this section. As a general rule, MadT performs two kinds of operations: it collects the memory accesses, and it dumps to file those accesses by translating them into symbol names and offsets. We are going to discuss the following topics:

1. how the target program is compiled for MadT tracing and linked to the MadT dynamic library (libMadT.so);
2. how the target program is launched;
3. how the MadT library is initialized;
4. how MadT keeps track of memory accesses;
5. how MadT translates a numerical address into a symbol name and an offset;
6. how MadT keeps track of dynamic memory handling;
7. how MadT writes the recorded events in the output file.

Compilation of the target program As a general rule, a target program that has to be profiled by MadT must be recompiled\footnote{It is possible to use MadT on programs that have not been specifically recompiled; in that case, however, the tool is much less flexible and accurate.} MadT depends on the GCC toolchain \cite{gcc}, because it makes use of a few peculiar extensions provided by the GCC compiler and loader.

No change to the source code of the target program is strictly required: MadT may start profiling right before executing the main() function, and it may stop right before terminating the process. However, the user may define a smaller portion of the target program execution to be profiled. Specifically, the MadT\_start() and MadT\_stop() functions can be added to the source code to, respectively, start and stop the profiler.

When compiling and linking the target executable, the user must specify some command line flags that instruct gcc to include debugging information, to link with the libMadT.so shared library, and to override the main() function of the target program with a specific MadT’s function (this is used when initializing the library, see below).

Together with the target executable file, the user must also generate two files that include the contents of the relocation section and of the symbol table of the target program. This can be easily done by using the readelf utility program (included in the GNU Binutils \cite{binutils} toolset).
Launching the target program MadT collects accesses by monitoring a live execution of the target program. The target must be run on a Linux-based system, because MadT extracts some information about the target process from the Linux-specific /proc/self/maps virtual file.

In order to start profiling the target program, the user typically sets a few environment variables that control MadT’s behavior. Next, the user launches the target program by passing the same command line arguments as in a normal execution. The memory accesses detected by MadT are written on a dedicated output file. By using some environment variables the user may force the profiler to start recording memory accesses right before executing the target’s main() function. (It is always possible to start and stop the profiler by inserting calls to MadT::start() and MadT::stop() in the target source code.) The user may also decide the format of the addresses recorded in the output file: numerical virtual addresses, symbolic addresses, or both.

Library initialization The MadT profiler is linked to the target program as a shared library. This means that at launch time the dynamic loader maps code and data of MadT in the memory space of the target process and ensures that specific library initialization functions are executed.

As explained in Section 5.1.1 MadT defines wrappers for some of the procedures defined in the C library and possibly in other shared libraries used by the target program. Therefore, MadT must initialize some of its data structures as early as possible: after the other shared libraries have been loaded but before the initialization code of other shared libraries run. This can be achieved by (1) imposing the dynamic loader to load MadT’s shared library before the other shared libraries, and (2) using a particular extension of the GCC C compiler that allows the programmer to define a given library function as a “constructor” procedure to be run before main().

In particular, when MadT is being initialized, the code checks whether libMadT.so has been effectively loaded first. If not, MadT sets some environment variables that modifies the libraries loading order, and restart the whole target program. This is done in a wrapper function invoked in place of target’s main(), because the target program has to be restarted by specifying the same command line arguments as the original execution, and these strings are made easily available only to main().

The initialization code of MadT also reads from the files generated at compile time the dynamic relocation table and the symbol table of the target executable. It also reads from the /proc/self/maps virtual file, present in any Linux system, the list of virtual memory regions currently defined for the target process. Finally, the initialization code installs signal handlers for the SIGSEGV and SIGTRAP signals, and invokes the main() function of the target program.

Collecting memory accesses As explained in Section 5.1.1 MadT forces a page fault exception whenever a machine instruction performs an operation that produces a memory access. In order to achieve this, MadT uses the mprotect() POSIX system call to remove all access rights to the set of pages containing the memory
cells whose accesses are being profiled. In particular, when the profiler is being activated, MadT scans the list of all virtual memory regions of the target process, and it removes the access rights from the pages in the regions corresponding to:

- the data segment of the target program or shared libraries
- the bss segment of the target program or shared libraries
- the heap of the target process

Access rights are never removed for pages of the MadT library and of the dynamic loader, and for pages containing executable code and stack. Thus, accesses to those pages are not recorded.

When a page fault exception occurs, the Linux kernel determines whether the faulty address is included in a memory region whose access rights forbid the tempted access. As usual, in this case the kernel sends the SIGSEGV signal to the program. Since MadT has installed a signal handler for SIGSEGV, the kernel saves the hardware context of the target program in the User Mode stack and forces the invocation of the signal handler. The handler analyzes the hardware context saved on the stack and determines the faulty address that caused the fault, the address of the instruction that tried the memory access, and the type of access (load or store). The handler can thus record these data in a log buffer. It then invokes the mprotect() system call to restore the proper access rights to the pages including the faulty address. This is done because MadT must ensure that the machine instruction that is accessing memory is executed once again without generating a page fault. Finally, the page fault handler modifies the hardware context saved on the stack so as to activate the hardware tracing mechanism when normal execution resumes.

When the SIGSEGV handler terminates, the machine instruction that caused the page fault is executed again. This time the pages including the memory cells accessed by the instruction should have proper access rights, so that likely the memory access can be successfully completed. Right after the execution of the machine instruction the CPU’s control unit raises a “trap” exception. The kernel thus raises a SIGTRAP signal and activates the corresponding handler defined in the libMadT.so library.

The trap handler removes any access rights to the pages of the virtual memory region including the address involved in the previous page fault event, Then, it modifies the hardware context of the target program saved on the stack so as to disable the hardware tracing mechanism. Finally, it checks whether the log buffer containing the events recorded by the profiler is nearly full; if so, it calls a procedures that empties the buffer by writing the events on the output file (see below).

When eventually the trap handler terminates, the normal execution flow of the process is resumed. Tracing is disabled, but pages to be profiled have no access rights, so that the profiler shall catch the next memory access.

**Translating virtual addresses to symbol names** When MadT initializes, it reads from some files generated at compile time the dynamic relocation table and the symbol table of the target executable. The
symbol names found in these tables are inserted in a red-black balanced binary tree indexed by the starting address of the symbol. MadT also reads from the \texttt{/proc/self/maps} virtual file the list of virtual memory regions currently defined for the target process, and insert each memory region in another red-black balanced binary tree indexed by the starting address of the region. Each node of the red-black tree is augmented with the size of the corresponding memory region, thus the binary tree allows MadT to quickly discover the memory region containing a given address.

The log buffer containing the events recorded by the profiler has limited size, thus it must be periodically flushed by writing its contents into the output file. Each record in the log buffer includes the virtual address of the machine instruction that triggered the fault, as well as the address of the memory cell being accessed. The record also includes a reference to the virtual memory region containing the page being accessed, and the type of access (i.e., load or store).

When flushing the log buffer, MadT translates the virtual addresses into meaningful symbol names and offsets relative to the symbols. In order to translate a raw virtual address into meaningful symbol name and offset, MadT looks up the raw address in the red-black balanced binary tree including all canonical symbols of the target program. If a match is found, it outputs the symbol name and the difference between the raw address and the “start” value associated with the symbol. If no match is found in the symbol tree, MadT invokes the \texttt{dladdr()} function of the dynamic loader to resolve the address at run-time\(^3\). If \texttt{dladdr()} returns a valid symbol, it is inserted in the symbol red-black tree. Otherwise, if MadT does not succeed in finding a symbol associated with the raw address, it outputs the name of the memory region including the raw address, as well as the relative offset with respect to the start of the region.

**Dynamic memory handling** The mechanism just described for translating virtual addresses into symbol names does not work when the addresses belong to memory blocks that have been dynamically allocated by means of C library procedures like \texttt{malloc() or mmap()}. Therefore, MadT tracks dynamic memory events so as to write meaningful information in the output file when these memory blocks are accessed.

The key idea is to establish proper wrappers for any C library function that handles dynamic memory. Currently MadT knows about the following functions: \texttt{malloc()}, \texttt{calloc()}, \texttt{realloc()}, \texttt{free()}, \texttt{mmap()}, \texttt{mremap()}, and \texttt{munmap()}. It is easy to extend MadT and add wrappers for other library functions, if required.

Once started, any wrapper routine typically checks if the profiler is active; if so, the wrapper disables profiling by restoring the access rights of the target memory pages. Then, the wrapper invokes the original procedure. Next, it records in the log buffer some data about the dynamic memory event: the address returned by the library function and the block size, in case of memory allocation or re-allocation; the address of the block being freed, in case of release. Finally, the wrapper removes the access rights to the target pages if the profiler was active when it started.

\(^3\)The \texttt{dladdr()} function is a non-POSIX extension of the “Glibc” library.
For performance reasons, symbols related to dynamic memory events are not forged when the events occur, but later when the corresponding entries stored in the internal buffer are written to the output file. Some dynamic memory events also modify the list of virtual memory regions of the process; in this case, MadT also updates the internal data structure that keeps track of those regions.

**Interpreting profiler’s output** Each line in the output file produced by MadT starts with a character that encodes the type of the recorded event. The type encoding is summarized in Table 5.1. The output file may include “raw” lines and “symbolic” lines. The second character in any line is “#” for “raw” lines, or “$” for “symbolic” lines.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>memory access of type “load”</td>
</tr>
<tr>
<td>S</td>
<td>memory access of type “store”</td>
</tr>
<tr>
<td>Y</td>
<td>block memory “copy” (memcpy()-like operation)</td>
</tr>
<tr>
<td>W</td>
<td>block memory “store” (memset()-like operation)</td>
</tr>
<tr>
<td>G</td>
<td>block memory “fetch” (write()-like operation)</td>
</tr>
<tr>
<td>M</td>
<td>dynamic memory allocation — malloc()</td>
</tr>
<tr>
<td>C</td>
<td>dynamic memory allocation — calloc()</td>
</tr>
<tr>
<td>P</td>
<td>dynamic memory allocation — mmap()</td>
</tr>
<tr>
<td>R</td>
<td>dynamic memory reallocation — realloc()</td>
</tr>
<tr>
<td>E</td>
<td>dynamic memory reallocation — mremap()</td>
</tr>
<tr>
<td>F</td>
<td>dynamic memory release — free()</td>
</tr>
<tr>
<td>U</td>
<td>dynamic memory release — munmap()</td>
</tr>
</tbody>
</table>

“Raw” lines show the numerical virtual addresses collected by the profiler. These lines may also include a string identifying the virtual memory region that includes the memory cells being accessed.

“Symbolic” lines show the addresses with the format “symbol.name+offset”: “symbol.name” is the symbolic name associated with the address, and “offset” is the difference between the address and the “start” value associated with the symbol. These lines may also include sizes (memory block lengths), and strings that identify the virtual memory region including the memory cells being accessed.

Symbol names forged by MadT to track dynamic memory blocks can be easily recognized because they always start with the character “<”: as this character cannot be used in valid C symbol names, there can be no confusion. The format of these symbol names is shown in Table 5.2. As already explained, MadT keeps a counter for the number of dynamic memory allocations. The current counter value is included inside the symbol name, as it helps in recognizing different memory allocations triggered by the same source code instruction.

Moreover, the allocation counter helps in recognizing accesses to memory blocks that have already been released. When MadT detects an invocation to free() or munmap(), it updates the name of the corresponding symbol by overwriting the first seven characters with the strings “<freed:” or “<unmap:”. The
Table 5.2: Symbol name format for dynamic memory events

<table>
<thead>
<tr>
<th>Symbol Name Format</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;malloc number@symbol+offset&gt;</code></td>
<td>malloc()</td>
</tr>
<tr>
<td><code>&lt;calloc number@symbol+offset&gt;</code></td>
<td>calloc()</td>
</tr>
<tr>
<td><code>&lt;realloc number@symbol+offset&gt;</code></td>
<td>realloc()</td>
</tr>
<tr>
<td><code>&lt;freed:number@symbol+offset&gt;</code></td>
<td>free()</td>
</tr>
<tr>
<td><code>&lt;memmap number@symbol+offset&gt;</code></td>
<td>mmap()</td>
</tr>
<tr>
<td><code>&lt;mremap number@symbol+offset&gt;</code></td>
<td>mremap()</td>
</tr>
<tr>
<td><code>&lt;unmap:number@symbol+offset&gt;</code></td>
<td>munmap()</td>
</tr>
</tbody>
</table>

The rest of the symbol name, however, is left unchanged, thus it is still possible to identify the dynamic memory block. Accesses containing symbol names starting with “<freed:” or “<unmap:” should never appear in the output file, unless the target program is accessing a dynamic memory block that has been previously released. Usually, this is a target program bug.

5.1.3 MadT use case

In order to provide a description of the capabilities of our memory access detection tool, we analyze in detail a trace fragment obtained by the execution of a benchmark program.

When MadT is used to observe the memory access pattern of a given target program, the produced output is a collection of trace messages corresponding to a set of specified events following the encoding in Table 5.1. As can be noted, the produced output trace features a very high event granularity, going down to the single memory access (loads and stores). We first show how the details about the collected events can be aggregated to produce a high-level view of the memory access pattern of the target program; next, we demonstrate how a one-to-one correspondence can be established between collected trace events and original source code.

For the analysis carried out in this section, we consider the “disparity” benchmark from the San Diego Vision Benchmark Suite [166]. This benchmark computes the 3-dimensional depth of a scenery provided in input as a pair of images taken from slightly different positions (stereoscopic view). Table 5.3 contains a summary of the number of events observed during the execution of the considered benchmark. The benchmark is executed with in input a stereoscopic image in Common Intermediate Format (CIF), i.e., having a resolution of 352x288.

Table 5.3: Summary of memory events for disparity benchmark

<table>
<thead>
<tr>
<th>Event</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of process memory region</td>
<td>30</td>
</tr>
<tr>
<td>Number of performed stores</td>
<td>2540679</td>
</tr>
<tr>
<td>Number of performed loads</td>
<td>5491371</td>
</tr>
<tr>
<td>Number of performed block-copy</td>
<td>0</td>
</tr>
<tr>
<td>Number of performed block-store</td>
<td>608288</td>
</tr>
<tr>
<td>Number of performed block-fetch</td>
<td>4</td>
</tr>
</tbody>
</table>
One way to read the memory trace produced by MadT is to use a visual approach. Figure 5.1 has been generated directly by post-processing the obtained memory trace for the disparity benchmark. This figure plots the virtual memory space of the analyzed process. Lower memory regions correspond to the top of the figure, while the bottom of the figure represents higher memory addresses. Gray horizontal lines denote the end of a given memory region (text, heap, ...) and the beginning of the next. Memory pages are ordered by starting address and each one is represented with a circle inside the corresponding memory region. Memory pages inside regions are ordered from left to right and from top to bottom. We use a color intensity code to show the access frequency of each page in the observed memory region (the darker, the more accesses). Gray and white pages represent respectively, memory pages that were not traced, and for which no access was performed. Subsequent non-traced regions or regions where no accesses were observed have been aggregated.

From the arrangement of accesses in the profile, some features of the benchmark are evident. First, the benchmark makes a heavy use of the heap section. Specifically, 99 pages marked in light red experience an uniform access frequency. These pages correspond to the buffer containing the stereoscopic image in input. The number of accesses recorded on the pages of the input buffer is on the order of $10^4$. One page of the heap, marked in darker red right after the input buffer, instead, experiences a number of accesses that is three orders of magnitude higher. As confirmed by source code analysis, this page stores the progressively built disparity map. Hence the large number of accesses. The frequently accessed pages at the top of the memory addressing space represent the data (bss, rodata) region. Moreover, the frequently accessed pages depicted in the second half of the figure belong to dynamic linker (libdl.so), C library, and stack. However,
Listing 5.1: Body of `findDisparity()` function

```c
#define subsref(a,i,j) a->data[(i) * a->width + (j)]

void findDisparity(F2D* retSAD, F2D* minSAD, I2D* retDisp, int level, int nr, int nc) {
    int i, j, a, b;
    for (i=0; i<nr; i++) {
        for (j=0; j<nc; j++) {
            a = subsref(retSAD, i, j);
            b = subsref(minSAD, i, j);
            if (a<b) {
                subsref(minSAD, i, j) = a;
                subsref(retDisp, i, j) = level;
            }
        }
    }
}
```

only stack pages accessed through block operations are highlighted in the figure, while no text pages are highlighted. This is because, as discussed earlier, they are not being traced by MadT in the current setup. In this example we show aggregate data, however it is possible to do the same analysis filtering a subset of the collected records, for instance selecting just read or write accesses.

**MadT output analysis**   Another way to exploit the output produced by the MadT tool is to relate captured accesses (or fragments of them) to portions of source code. This allows the user to extract accurate information about the memory access pattern of a given code fragment in order, for example, to deduce data locality and memory-intensity. Considering the previously discussed disparity benchmark, we have isolated a single function used to find the disparity between two raster images by shifting them over each other. The function is called `findDisparity()`, and its C code is reported in Listing 5.1.

As can be observed, the function performs a nested loop over `nr` and `nc`, which represent the number of rows and columns of the image. For simplicity, we now analyze an execution of the function when a low resolution image is used in input to the benchmark. Specifically, we use a $10 \times 13$ image, so that `findDisparity` will be called with $nr=10$ and $nc=13$. In the considered invocation of the function, the parameter `level` is set to 0.

We now show that it is possible to relate a fragment of captured memory accesses with the execution flow of the benchmark. Three of the 130 iterations of nested loop are reported in Listing 5.2; specifically, they are the first, the second, and the last iteration. In order to follow the example, moreover, we report at the top of Listing 5.2 the dynamic memory allocations for the three structures used in the function (`retSAD`, `minSAD` and `retDisp`), as captured by MadT. For all the involved structures, the field `data` has a offset of 8 bytes. Finally, for all the considered iterations, the condition `(a<b)` is always satisfied.

It can be noted that the pieces of information that appear in the entries of the trace contain: (1) the location where `malloc()` was performed (e.g., `fMallocHandle()` for allocation of floats); (2) in which memory region the access falls, e.g., `[heap]`; (3) the offset in the function of the instruction performing the

---

4The names `<malloc number symbol+offset>` have been shorted in `<number symbol+offset>` in order to save space.
// Dynamic allocation of minSAD */
M$7 :< m06@mallocHandle +45>,528

// Dynamic allocation of retDisp */
M$8 :< m07@mallocHandle +45>,528

// Dynamic allocation of retSAD */
M$14 :< m13@mallocHandle +45>,528

... 

L$0 :< m13@mallocHandle +45>+0,[heap], findDisparity +55
L$1 :< m13@mallocHandle +45>+8,[heap], findDisparity +73
L$2 :< m06@mallocHandle +45>+0,[heap], findDisparity +90
L$3 :< m06@mallocHandle +45>+8,[heap], findDisparity +108
S$4 :< m06@mallocHandle +45>+8,[heap], findDisparity +127
L$5 :< m07@mallocHandle +45>+0,[heap], findDisparity +134
S$6 :< m07@mallocHandle +45>+8,[heap], findDisparity +147

... 

L$8 :< m13@mallocHandle +45>+0,[heap], findDisparity +55
L$9 :< m13@mallocHandle +45>+12,[heap], findDisparity +73
L$10 :< m06@mallocHandle +45>+0,[heap], findDisparity +90

... 

L$1032 :< m13@mallocHandle +45>+0,[heap], findDisparity +55
L$1033 :< m13@mallocHandle +45>+524,[heap], findDisparity +73
L$1034 :< m06@mallocHandle +45>+0,[heap], findDisparity +90

... 

access, in this case findDisparity+off.

The first memory operation in the function expands in an operation of the form: a = retSAD->data[0]. Hence two loads (L) are generated at offset 0 and 8 from m13, for retSAD and retSAD->data respectively. The next two loads are produced in an analogous way for minSAD and minSAD->data. Since the condition (a<b) holds, the symbol minSAD is accessed again in order to store the value of a at the memory location of minSAD->data. Analogously, with the last statement of the loop, the value of level is stored in retDisp->data, originating another pair of load and store operations.

The second iteration of the nested loop is identical to the first one, except for the value of j that has been incremented to 1. Thus, only the final addresses for the expressions of the type subsref(buffer, i, j) will change. Given the value of j=1, subsref(buffer, i, j) will expand in buffer->data[1]. Each element in the data arrays occupies 4 bytes. Thereby, accesses to retSAD, minSAD and retDisp will produce operations on data at offsets 12 from the respective structure.

In the final iteration of the inner loop, the values of i and j are 9 and 12 respectively. This means that the first operation will be of the form retSAD->data[9 * 13 + 12], i.e., retSAD->data[129]. Following the pointer arithmetic, this will result in an access to retSAD->data + 516 = retSAD + 524. A similar reasoning allows deriving the rest of the accesses in the iteration.

5.1.4 Evaluation

In this section, we compare the performance of MadT with respect to existing tools that are able to perform complete memory tracing of applications. Specifically, we focus our comparison on two well established
memory analysis tools, namely the Intel Pin tool [129] and the Valgrind Lackey tool [196].

Pin is a proprietary software component developed by Intel, which is free for non-commercial use, and implements a dynamic binary instrumentation framework for the IA-32 and x86-64 architectures. It provides a set of APIs that abstract the underlying ISA, supporting the creation of tools for dynamic program analysis. Pin performs its instrumentation at run-time on the compiled binary files without requiring recompilation. Its instrumentation framework also provide basic access to symbols and debug information.

In the context of this work, we focus on one of the sub-tools developed using the Pin APIs: the Pinatrace tool. This tool performs instrumentation only of instructions that read or write memory, producing in output the complete trace. Each entry in the trace reports three pieces of information: (1) address of memory instruction; (2) type of operation (read or write); (3) address of accessed memory location.

Similarly, Valgrind is a open-source binary instrumentation framework on top of which several dynamic analysis tools have been developed for code graph analysis and heap/stack profiling. The Valgrind framework is being actively developed and, unlike the Pin tool, features support for a large number of architectures, including IA-32, x86-64, PowerPC, ARM, and MIPS. One of the tools included in the Valgrind suite, called Lackey, performs a complete tracing of memory accesses and instruction fetches when executed with the option --trace-mem=yes. The style of the output is quite minimal, since per each memory access only two pieces of information are recorded: (1) access type (read, write or instruction fetch) and (2) address of the accessed memory location. However, each record corresponding to a data memory access appears in the trace immediately after the memory access corresponding to the fetch of the instruction that performed the data access.

We have executed the experiments on a set of real benchmarks that include applications from the MiBench suite [197] and the San Diego Vision benchmark suite [166]. The selected applications include object tracking, texture creation, jpeg compression/decompression, heavy mathematical processing, and sorting. We also tested the behavior of the considered tracing tools on simple synthetic benchmarks that stress single library functions that access memory in blocks and/or modify the memory layout of the application: calloc, realloc, read, fwrite and mmap. In addition, mandelbrot represents a computation intensive benchmark. The experiments have been performed on a server-grade machine featuring a Intel Xeon E5-2640 CPU operating at a frequency of 2.5 GHz, with 15 MiB of last-level cache and 16 GiB of DRAM. On this platform, the tools have been tested using a Linux 3.2 kernel and the GCC 4.7.2 compiler.

Figure 5.2 shows the comparison among run-times for a selection of benchmarks from the considered suites. The execution times have been normalized with respect to the MadT runtimes. Each bar shows the percentage of slow down (positive range) or runtime reduction (negative range) of Pinatrace and Lackey compared to MadT. As can be observed, in 5 benchmarks (stringsearch, qsort, math, disparity, and localization) MadT achieves significant performance benefits over both Pinatrace and Valgrind. In these examples, in fact, the slowdown introduced by Pinatrace and Lackey goes from 90% up to almost 800% compared to MadT. In general, this trend is visible with benchmarks that are mostly computation-intensive,
while MadT does not perform so well with memory intensive applications (i.e., `texture_synth`). Intuitively, this is because on one hand the handling time for the sequence of exceptions (`SIGSEGV + SIGTRAP`) is higher than instrumenting the single memory instruction, while on the other hand instructions that do not access memory are always executed natively with zero overhead on the CPU.

Due to space constraints, the summary for the results obtained on the complete set of benchmarks is reported in Table 5.4. As can be seen from the table, visible benefits are obtained on synthetic benchmarks that heavily use C library functions to operate on blocks of memory. This is because MadT is able to intercept library calls that perform block operations and treat their accesses in an aggregate way, instead of tracing their behavior instruction by instruction. Moreover, as discussed above, MadT behaves very well on computation intensive applications such as the Mandelbrot benchmark. Here MadT is able to provide performance that match the native execution because this benchmark performs a limited number of memory accesses, while intensely executing mathematical operations on few CPU registers.

Next, we consider a subset of benchmarks from the MiBench suite that reflect common workloads in automotive and industrial domain [197]. The measurements for this set of benchmarks is included in second section of Table 5.4. It is easy to observe that MadT always achieves better or comparable performance with respect to Pinatrace and Lackey tools. As in the case of basic benchmarks, Valgrind runs in about the same order of magnitude of the Pin tool. This reflects the fact that both the tools use the same approach of performing binary instrumentation. The slight disadvantage of Valgrind seems to be related to optimization problems, since it is designed to support a broad set of architectures.

Finally, the last section of Table 5.4 reports the achieved performance on the set of computer vision
Table 5.4: Summary of benchmarks normalized runtimes

<table>
<thead>
<tr>
<th></th>
<th>No Tracing</th>
<th>MadT</th>
<th>Pinatrace</th>
<th>Lackey</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synthetic Benchmarks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>calloc</td>
<td>1</td>
<td>50</td>
<td>1520</td>
<td>2040</td>
</tr>
<tr>
<td>realloc</td>
<td>1</td>
<td>1</td>
<td>1270</td>
<td>1730</td>
</tr>
<tr>
<td>read</td>
<td>1</td>
<td>10</td>
<td>1170</td>
<td>1530</td>
</tr>
<tr>
<td>mmap</td>
<td>1</td>
<td>20</td>
<td>2320</td>
<td>2980</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4949</td>
</tr>
<tr>
<td><strong>MiBench Benchmarks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stringsearch</td>
<td>1</td>
<td>290</td>
<td>1560</td>
<td>2300</td>
</tr>
<tr>
<td>susan</td>
<td>1</td>
<td>61520</td>
<td>25840</td>
<td>104490</td>
</tr>
<tr>
<td>jpeg</td>
<td>1</td>
<td>6527</td>
<td>4448</td>
<td>12378</td>
</tr>
<tr>
<td>qsort</td>
<td>1</td>
<td>15140</td>
<td>29410</td>
<td>75450</td>
</tr>
<tr>
<td>math</td>
<td>1</td>
<td>1021</td>
<td>2264</td>
<td>6832</td>
</tr>
<tr>
<td><strong>San Diego Vision Benchmarks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>disparity</td>
<td>1</td>
<td>2744</td>
<td>6216</td>
<td>5237</td>
</tr>
<tr>
<td>localization</td>
<td>1</td>
<td>587</td>
<td>5190</td>
<td>4410</td>
</tr>
<tr>
<td>mser</td>
<td>1</td>
<td>1907</td>
<td>2044</td>
<td>1617</td>
</tr>
<tr>
<td>multi_ncut</td>
<td>1</td>
<td>12732</td>
<td>4815</td>
<td>7413</td>
</tr>
<tr>
<td>sift</td>
<td>1</td>
<td>20940</td>
<td>21842</td>
<td>21123</td>
</tr>
<tr>
<td>svm</td>
<td>1</td>
<td>17221</td>
<td>16548</td>
<td>15593</td>
</tr>
<tr>
<td>texture_synth</td>
<td>1</td>
<td>24855</td>
<td>5534</td>
<td>3654</td>
</tr>
<tr>
<td>tracking</td>
<td>1</td>
<td>19720</td>
<td>18966</td>
<td>3299</td>
</tr>
</tbody>
</table>

benchmarks. These benchmarks accept as input a series of sensor data or images, read them into internal buffers and compute their output locally by working on dynamically allocated memory. As a result, they are memory-intensive, hence significant slowdown is introduced by all the considered memory tracing tools. From the obtained results, it can be concluded that MadT does not improve on binary instrumentation tools when normal instruction execution is closely interleaved with frequent memory accesses. Intuitively, it can be concluded that, from a pure performance standpoint, not much advantage can be achieved by executing non-memory instructions natively and triggering memory protection traps upon memory accesses. In fact, the penalty of handling frequent segmentation faults can come close to the slowdown introduced by adding an intermediate binary translation layer.

We argue however that when comparable performance are achieved, the proposed MadT still represents the most valid memory-tracing option for system designer or third-party tools. In fact, MadT provides two main advantages that can allow post-processing of the produced trace to be simpler and more detailed. (1) Unlike Pinatrace and Lackey, MadT performs symbol resolution on observed memory accesses. This way, it is always able to provide the memory region in which the access has been performed and in most of the cases the name of the variable it is related to. Furthermore, (2) because no sandboxing is performed on the executed task, no additional memory regions are added to the task at analysis time, apart from the minimal number of regions required to link the MadT library. This makes significantly easier to infer the
memory behavior of the task executing natively (i.e., outside the tracing environment) from the collected trace. Conversely, both Pin and Valgrind significantly rearrange the memory region layout of the analyzed task by almost doubling the total count of regions.

5.2 Profiling to Drive Cache Allocation

Being able to acquire the memory trace of an application is fundamental to use execution-driven profiling techniques. In [6] we demonstrate how information from memory traces can be exploited to build a task profile to drive cache allocation using Colored Lockdown (see Section 3.2). Specifically, as the task runs in the profiling environment (MadT could be used as well as a tracer) and memory accesses are traced, per-page access statistics are maintained. Next, (i) pages of the task’s addressing space are ranked by access frequency; and (ii) a profile is produced identifying frequently accessed (hot) pages by their relative position in the addressing space. Thanks to its relative positioning mechanism, derived memory profiles are valid from run to run despite the presence of memory address layout randomization.

For each task, by following its profile, it is possible to derive a curve that plots the WCET of a task as a function of the number of hot memory pages allocated in cache. We call this curve progressive lockdown curve. It relates cache assignment with resulting WCET, so that once the cache assignment has been performed, two parameters are derived: (1) a value of WCET \( C \) for the task running in isolation (single-core scenario); and (2) a residual maximum number of cache misses \( \mu \), corresponding to accesses to all those profile pages not allocated in cache. As previously discussed, the WCET \( C \) and the \( \mu \) parameter obtained at this step can be used to derive the value of WCET when \( m \) are the active cores in the system: WCET\((m)\) [8]. Finally, since the pages in the derived profile are ranked by access frequency, the progressive lockdown curve will exhibit a convex shape and constrained convex optimization methods can be used to obtain the optimal cache allocation for a group of tasks.

5.2.1 Memory profiling

The aim of the memory profiler is to identify the “hot” (most frequently accessed) virtual memory pages for a given executable. However, the detection procedure can not be based on absolute virtual addresses, because virtual addresses change from execution to execution. A common abstraction in operating systems is the concept of a memory region: a range of virtual addresses assigned to a process to contain a given portion of its memory. Thus, processes own several memory regions to hold executable code, the stack, heap memory and so on. Our goal is thereby exploiting such an abstraction to create a profile where a hot page is expressed as an offset from the beginning of the memory region it belongs to. In this way, the profile needs to be created only once, and it is possible to determine the effective address of the hot page at execution time. For instance, the hottest memory page might be the fifth page (offset) inside the third memory region.
of the process. In this case, the corresponding entry in the memory profile will look like: \(3 + 0x0005\).

Memory pages belonging to dynamically linked libraries are not included in the produced profile. Thus, if it is necessary to keep track of the memory accesses performed by a given task inside a library procedure, that library has to be statically linked inside the task executable.

Note that in the presented implementation we have generated the profiles considering a single input vector for each task. However, as a future work, we plan to improve the profiling technique to aggregate the data acquired from several task executions with different input vectors to enhance coverage.

If the assumptions made in the previous section hold, then, from the memory profile, it will always be possible to correctly calculate the position of each hot memory page. Finally, the obtained profile can then be handed off to the Colored Lockdown module, which can process this information and perform the desired operations on the correct set of pages.

5.2.2 Approach description

The proposed profiling technique can be divided in three phases. In the first phase, called “accesses collection”, the following operations are performed:

1. collect all of the virtual addresses accessed during the task execution;
2. create a list of all the accessed memory pages sorted by the number of times they were accessed;
3. record the list of memory regions assigned by the kernel in the profiling environment.

In the second phase, called “areas detection”, the analyzed task is run outside of the profiling environment, so that we can:

4. record the memory areas assigned by the kernel under normal conditions.

Finally, in the third phase, called “profile generation”, we:

5. link together the list of memory regions assigned during the profiling (obtained in step 3) with those owned by the process under normal conditions (step 4);
6. determine in which region of the list obtained in step 4 each entry of the list obtained in step 2 falls, according to the mapping determined in step 5;
7. generate the \textit{execution-independent} memory profile of the task.

The profile file obtained at the end of the procedure can be given as an input to drive the Colored Lockdown procedure.

In order to exploit the presented technique on a given executable, we need to insert, in the executable code, a call to a small procedure which generates the intermediate auxiliary files. In particular, it has to be
Figure 5.3: Profile creation for the benchmark “a2time01” and relations between the files involved, starting from the ranked list of accessed memory pages (a) to the final profile file (f).

inserted between the end of the start-up phase and the beginning of the periodic phase. In the final version of the executable, or in general once the profile has been generated, the procedure can be maintained (and called) as it is, to avoid altering the arrangement of the text section. Alternatively, its assembly instructions can be replaced with NOPs. We will refer to this procedure as: aux_files_out.

**Accesses collection** To collect information on which memory pages are most frequently used, the profiler must first intercept all memory accesses. To perform this step, we used a sub-tool of Valgrind called Lackey, which outputs (virtual) memory accesses as the program is running. Since the granularity of the Colored Lockdown mechanism is one memory page, in the first elaboration step, we trim the 12 least significant bits of the outputted addresses to obtain the (virtual) page number. These page accesses are recorded together with the number of times they are accessed. Doing so, it is possible to sort pages by the number of accesses to produce a ranking of the most frequently used “hot” memory pages.

The list produced in this step is called mempages.dat. The entries in this list are shown in Figure 5.3a, where an alphabetic ranking is associated with the page hotness. However, it contains a list of hot memory pages identified by absolute memory addresses. As previously stated, they have to be converted to relative values with respect to process memory regions.

Valgrind uses a library preload mechanism to instrument running processes. This means that, when an executable runs under Valgrind, it owns a completely different set of memory regions than when it runs outside the profiling environment. Moreover, the tool Lackey also records the accesses to some memory locations belonging to Valgrind itself. For these two reasons, the pages listed in mempages.dat have to be mapped back into memory regions owned by the process outside of the profiling environment.

In this access collection phase, the aforementioned aux_files_out procedure generates two additional files, called respectively memareas.profile and memaddrs.profile. The first file, shown in Figure 5.3b, contains the list of memory regions assigned to the analyzed task in the profiling environment. Each region is reported
as a pair of the form (start address, end address). This list is obtained relying on the standard Linux proc virtual file system. Specifically, `memareas.profile` is generated as a copy of the file `/proc/<pid>/maps`.

The second file `memaddrs.profile`, reported in Figure 5.3c, is generated by outputting the address of a set of variables located in different regions of the executable. Specifically, the addresses of a stack, a text, a data and a read-only data variable. As explained at the end of this section for sake of simplicity, tracing heap addresses requires additional logic.

**Areas Detection** As previously discussed, we need to map all the memory accesses performed by the process in the profiling environment back to the set of memory regions owned by the same executable running under regular conditions. To do this, we perform a second run of the task without Valgrind, in the native environment.

As in the access collection phase, the discussed `aux_files_out` generates a pair of files which are semantically identical to the previous case, in which the values are referred to the executable in its native environment. These files, reported in Figure 5.3e and Figure 5.3d, are given the names `memareas.real` and `memaddrs.real` respectively.

**Profile Generation** In this last phase, the actual profile is generated by merging all the pieces of information contained in the set of files previously created, as we show in Figure 5.3. For the first step, we leverage on the fact that there is a 1:1 correspondence between the lines of the two files `memaddrs.profile` and `memaddrs.real`, since they have been generated by the same procedure. We exploit this property to establish a mapping between the set of memory regions belonging to the task in the native and in the profiling environment, i.e. between the entries in the files `memareas.real` and `memareas.profile`.

Once the mapping between the two set of memory regions has been computed, it is possible to relate each accessed page in the profiling environment - i.e. each entry of the file `mempages.dat` - to the correspondent area in the set of memory regions assigned to the process in the native environment. Thus, it is finally possible to come out with a relative notation to address a given hot page that will be valid for every task execution.

Finally, the memory profile (Figure 5.3f) for the executable is generated. Each entry in this file identifies a hot page by a memory region index and an offset. The entries in this list respect the same alphabetical order as in Figure 5.3a. The Colored Lockdown module will read this file to determine which pages need to be locked. Note that some pages - such as those belonging to dynamically linked libraries - are not present in the final file because they have been filtered out.

**Heap Memory** As previously mentioned in the accesses collection phase, outputting the address of a single variable in each memory region of interest leverages on the assumption that a text variable, a stack variable, and so on, can be only contained in one memory region. Nevertheless, this is not the case with
the heap: the heap could actually be covered by multiple memory regions.\textsuperscript{5} Thereby, to find the heap memory regions and to make sure that we catch all the other memory regions that are created using \texttt{mmap}, we need to do a little extra work. In particular, we compile the executable \textit{wrapping} all \texttt{malloc} and \texttt{mmap} calls, which is easily done using a GCC extension (\texttt{-W}rap \texttt{ld}'s option), and does not involve re-engineering or rewriting the executable source code. The wrapper is implemented so as to intercept all \texttt{malloc} and \texttt{mmap} calls which create an anonymous memory region (\texttt{MAP_ANONYMOUS} flag). Finally, the starting address of every allocated memory area is appended to the already discussed files \texttt{memaddrs.profile} and \texttt{memaddrs.real} (Figure \ref{fig:memaddrs}).

5.2.3 Cross-platform compatibility

As previously mentioned, the produced memory profile can be used an arbitrary number of times on the same system where it has been produced. Moreover, it can be produced on a cross-platform and moved in the final system, given that the latter is a \textit{compatible} system. In particular, two platforms are compatible if:

1. they are based on the same architecture;
2. the memory region assignment policy is known to be the same (e.g. they run the same Linux kernel);
3. both use the same compilation tool-chain.

It is worth noticing that the proposed technique relies on task abstractions (such as memory regions) which are common to almost all the OSes. This means that the profiling mechanism can be ported to virtually any OS, provided that it offers as an invariant that the order of the memory regions assigned to a task does not vary from execution to execution. This is true for the great majority of UNIX-based systems.

Moreover, the memory access collection is performed using Valgrind and the dynamic memory allocation detection exploits a GCC extension. Both the tools are available for a wide range of platforms, thus we are confident that our technique applies to a variety of architectures, including x86, ARM and PowerPC.

5.2.4 Evaluation

The results of running the memory profiler on the seven EEMBC benchmarks can be seen in Table \ref{table:memory_profile}. The second column displays how many different memory pages were found to be accessed by a given benchmark. This number is equivalent to the number of lines in the profile file, similar to the sample one in Figure \ref{fig:sample_profile}. In our experiments, we have chosen to color/lock a subset of the resulting pages. In particular, the third column of Table \ref{table:memory_profile} shows how many pages per each benchmark have been selected. The fourth column reports the percentage of accesses that fall in the selected pages with respect to the total number of accesses.

\footnotesize{Under certain implementations of \texttt{malloc}, memory allocations over a fixed size are put into a new memory section using \texttt{mmap} instead of expanding the heap using the \texttt{brk} system call.}

\normalsize{5}
As it is shown in the table, we have selected the minimum number of pages so to cover at least the 80% of accesses.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total pages</th>
<th>Hot pages</th>
<th>% accesses in hot pages</th>
<th>Iters/Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2time</td>
<td>15</td>
<td>4</td>
<td>81%</td>
<td>167</td>
</tr>
<tr>
<td>basefp</td>
<td>21</td>
<td>6</td>
<td>97%</td>
<td>675</td>
</tr>
<tr>
<td>bitmnp</td>
<td>19</td>
<td>5</td>
<td>80%</td>
<td>43</td>
</tr>
<tr>
<td>cacheb</td>
<td>30</td>
<td>5</td>
<td>92%</td>
<td>22</td>
</tr>
<tr>
<td>canrdr</td>
<td>16</td>
<td>3</td>
<td>91%</td>
<td>498</td>
</tr>
<tr>
<td>rspeed</td>
<td>14</td>
<td>4</td>
<td>85%</td>
<td>167</td>
</tr>
<tr>
<td>tblook</td>
<td>17</td>
<td>3</td>
<td>81%</td>
<td>78</td>
</tr>
</tbody>
</table>

Table 5.5: Memory profiling results

5.3 Profiling to Reorganize Application Behavior

As described in Chapter 4, being able to enforce a structure in the way applications access shared resources is key to predictability and efficiency on multi-core systems. This is because co-scheduling strategies can be used to contemporarily manage CPU time, memory resources and I/O devices.

Suppose that we are able to structure the task behavior so that accesses to shared hardware resources are localized in clusters with known properties (position and length). In this case, the resource/CPU manager could schedule clusters of resource requests and parallelize blocks of pure execution. Fundamentally, this is the approach that we followed for the design of our SPM-centric OS (see Section 4.2). The PRedictable Execution Model [16] uses the same idea on cache-based architectures. PREM-compliant tasks can be obtained through code re-factoring. The resulting benefits in terms of I/O and memory scheduling, originally discussed in [19, 198] and [16], are substantial. Manually re-factoring the structure of a legacy application as suggested in [19], however, represents an extremely time consuming and error-prone procedure that requires a deep understanding of the application code.

In [17] we demonstrated that application profiling through memory tracing can be used to automate code re-factoring and we propose the implementation of a PREM-compliant re-factoring tool, namely Light-PREM, to support the adoption of the PREM model for complex applications. Light-PREM operates at the level of a single code function. At the end of the re-factoring process, memory accesses are clustered at the beginning of the target function using memory prefetching instruction, while the rest of the code executes relying on locally cached data. Note that since the source code is modified by the tool, re-certification and testing is required on the obtained PREM-compliant application. Although Light-PREM’s goal is to allow resource co-scheduling on cache-based architectures, many of the principles in Light-PREM can be reused in the context of scratchpad-based architectures.

Light-PREM is able to perform memory access clustering automatically by generating PREM annotations.
PREM annotations encapsulate prefetch statements for memory locations accessed inside the function under analysis. In this way, when the predictable function executes, the prefetch block (memory phase) placed at the beginning of the function brings into the local CPU cache those memory blocks that will be accessed during the execution phase. Light-PREM determines which memory accesses to prefetch by using memory profiling tools, as discussed in Section 5.1. Next, it determines how to construct such prefetches by translating the absolute memory addresses recorded at trace collection time into chains of pointer dereferences that always begin with an in-scope variable. Internally, Light-PREM intercepts memory allocation procedures and constructs a profile of the application’s memory layout. Such profile has the form of a graph where nodes are application’s memory blocks (e.g. stack, dynamically allocated blocks, text) and edges correspond to pointers between different memory blocks. The technique is compiler-independent.

5.3.1 Background on PREM

In [16], the PRedictable Execution Model (PREM) was proposed. The intuition behind PREM is that in the general case, the memory access patterns of tasks executed on commodity hardware can exhibit a high variance. In particular, predicting an exact sequence of fetches in main memory is extremely difficult, while, at the same time, assuming the worst case scenario leads to very pessimistic assumptions. Thereby, the key idea in PREM is to enforce, for a given set of tasks, a novel execution model with three main features:

1. jobs are divided into a sequence of non-preemptive scheduling intervals;

2. some of these scheduling intervals (named predictable intervals) are executed without cache misses by prefetching all required data at the beginning of the interval itself;

3. the execution time of the predictable intervals is kept constant by monitoring CPU time counters at run-time.

More in detail the code for each task \( \tau_i \) is divided into a set of \( N_i \) scheduling intervals \( \{s_{i,1}, \ldots, s_{i,N_i}\} \), which are executed sequentially at run-time. The timing requirements of \( \tau_i \) can be expressed by a tuple \( \{\{e_{i,1}, \ldots, e_{i,N_i}\}, p_i, D_i\} \), where \( p_i, D_i \) are the period and relative deadline of the task, with \( D_i \leq p_i \), and \( e_{i,j} \) is the maximum execution time of \( s_{i,j} \), assuming that the interval runs in isolation with no memory interference. A job can only be preempted by a higher priority job at the end of a scheduling interval. This ensures that the cache content can not be altered by the preempting job during the execution of an interval. The scheduling intervals are classified into compatible intervals and predictable intervals.

Compatible intervals are not characterized by any special property (they are backwards compatible). Cache misses can happen at any time during these intervals. The task code is allowed to perform OS system calls, but blocking calls must have bounded blocking time. Furthermore, the task can be preempted by interrupt handlers of associated peripherals. It is assumed that the maximum execution time \( e_{i,j} \) for a
compatible interval is computed based on static analysis techniques. However, to reduce the pessimism in the analysis, peripheral traffic is prohibited from being transmitted during a compatible interval. Ideally, there should be a small number of compatible intervals which are kept as short as possible.

Predictable intervals are specially factored to execute according to the PREM model shown in Figure 5.4: they are divided into two different phases and exhibit three main properties.

- First, during the initial memory phase, the CPU accesses main memory to perform a set of cache line fetches and replacements. At the end of the memory phase, all cache lines required during the predictable interval are available in last level cache. The memory phase of a generic predictable interval $e_{i,j}$ is denoted as $e_{i,j}^{\text{mem}}$.

- Second, during the following execution phase, the task performs useful computation without suffering any last level cache misses. Predictable intervals do not contain any system calls and can not be preempted by interrupt handlers. Hence, the CPU does not perform any external main memory access during the execution phase. This property allows peripheral traffic to be scheduled during the execution phase of a predictable interval without causing any contention for access to main memory. The execution phase of a generic predictable interval $e_{i,j}$ is denoted as $e_{i,j}^{\text{exec}}$.

- Third, at run-time, the time length of a predictable interval is forced to always be equal to $e_{i,j}$.

Note that in the original formulation of PREM, predictable intervals cannot contain any system call and cannot be preempted by interrupt handlers, to prevent the CPU from performing external main memory access during the execution phase. Not enforcing such constraint would result in two undesirable effects: (1) the OS would perform some transactions in main memory violating the separation between memory phase and execution phase; (2) the memory transactions performed by the OS could displace some memory locations that have been allocated in cache during the memory phase.
Such limitations can be addressed by relaxing the mentioned constraints on predictable intervals. Specifically: problem (1) can be solved by computing an upper bound on the number of memory accesses performed by the OS to satisfy a system call (or to execute an interrupt handler) and accounting them when computing the WCET; problem (2) can be addressed by using deterministic cache allocation mechanisms that have been largely studied [19, 193, 6]. We are currently investigating the exploitation of cited cache allocation mechanisms as part of our future work.

The big advantage of having a real-time task executing according to the PREM model is that it enforces a predictable memory access pattern. Thereby, it becomes possible to perform high-level coarse-grained real-time scheduling among multiple masters contending for access to a shared physical resource (like a shared bus and/or memory controller). For instance, it becomes possible to co-schedule memory transactions from a real-time I/O subsystem [19] in a way that they do not interfere with the execution of PREM-compliant real-time tasks. Recently, a Memory-Centric Scheduling framework [198] was also proposed that uses the PREM model to co-schedule memory accesses from contending CPUs of a multi-core platform: from the point of view of each core, the memory subsystem is seen as a slower but isolated one avoiding the experiencing of unpredictable temporal behavior due to a shared memory architecture. Finally, since the overall length of the predictable interval is constant, it is always possible to perform schedulability analysis of I/O transactions in the way it has been discussed in [16].

5.3.2 Light-PREM Overview

Configuring a system to exploit the benefits of the PREM model requires reworking application code so that tasks are split into scheduling intervals, which must either conform to the requirements of a compatible interval or a predictable interval. However, such legacy-to-PREM refactoring requires dividing tasks into scheduling intervals, and to manually add prefetch statements for all those memory locations that are needed in the execution phase of each predictable interval. This last step involves either having a perfect knowledge of the semantics or reverse engineering the code. Thus, as the size of the source code increases, so does the time required to manually perform this process. As such, this method does not scale with the size of the code base.

Light-PREM solves this problem by automatically generating prefetch statements for the memory phase of predictable intervals. In other words, it can be considered as a code refactoring tool that is able to enforce the separation between memory phase and execution phase inside what has been tagged by the programmer to become a predictable interval. It is worth highlighting one more time that such simplicity in performing legacy-to-PREM conversion of an arbitrarily large number of real-time tasks enables the use of novel scheduling techniques that can rely on a highly predictable memory access pattern to coordinate the access of several masters to the shared bus [19, 198, 16].

A possible solution to this problem could involve operating at a compiler level. Since the compiler al-
ready parses the source code, it generates data structures, such as the Abstract Syntax Tree (AST), which would allow much easier manipulation of the program. We investigated utilizing compiler passes such as those offered by the LLVM [200] compiler framework to automatically add prefetch statements at compile time. Unfortunately this method would introduce a dependency on the compiler tool-chain, thus breaking the property of Light-PREM to remain compiler-independent. This property is fundamental because industries often use proprietary compilers, so that: (1) reimplementing Light-PREM on a custom compiler would result in an unreasonable effort; (2) they would be unwilling to switch to third-party compilers. Thus, we implemented Light-PREM at the source code level instead. As we show, this technique yields excellent results, even though Light-PREM can achieve a knowledge of the code semantics which is significantly lower than what can be achieved by the compiler. As shown in Section 5.3.5, Light-PREM is able to achieve performance comparable to manual refactoring.

Light-PREM Workflow  The refactoring process performed by Light-PREM requires (1) analyzing the function to determine what memory accesses occur, and (2) generating prefetch statements to construct the memory phase (note that the memory phase is located at the beginning of the function). From an high-level point of view, the steps performed by Light-PREM are the following:

1. **access collection**: memory accesses performed inside the predictable function are recorded;
2. **chunk detection**: allowed memory ranges that can be accessed by the application are determined;
3. **handle detection**: in-scope variables (handles) at the beginning of a function are detected;
4. **graph construction**: chunks are interconnected detecting the existence of pointers from one chunk to another;
5. **relative expression construction**: prefetch statements are built for any memory access that was detected at Step 1.
6. **prefetch aggregation**: subsequent, contiguous prefetch statements are aggregated together to minimize the overall number of added statements.

**Access collection**  Since analyzing the behavior of a function from a semantic point of view requires either programmer or compiler knowledge, Light-PREM approaches the problem with a sample-based technique instead. Specifically, we perform a dry run of the function under analysis to record all the performed memory accesses. The output of this phase is a raw list of all the virtual memory addresses referenced by the predictable function. Such list represents the first step that is required to attain the goal of generating prefetch statements for all the memory misses in the execution phase.

The described memory access collection procedure can be performed using standard and architecture independent profiling tools, as detailed in Section 5.3.3. It is also important to underline that having a
perfect knowledge of the accessed memory locations is not enough to generate prefetch statements that can
be permanently added to the source code. This is because virtual memory addresses change from run to run
and thus, a prefetch on a constant memory address may result in an invalid addressing request in subsequent
runs.

**Chunk detection** Each application owns a set of ranges of virtual addresses that are accessible at exec-
tution time. Such ranges are assigned at loading time or obtained at execution time with dynamic memory
allocation. We refer to a valid range of virtual addresses as a *memory chunk*, or simply *chunk*. Detecting the
memory chunks belonging to the application under analysis has two main purposes. First, it is needed to
locate each observed memory access in the corresponding chunk. Second, it is required to understand how
chunks are interconnected between each other.

As it will be clear in the graph construction step, the majority of nodes in the memory graph are chunks.
They are defined in terms of starting address and size. To capture chunks, Light-PREM intercepts mem-
ory allocation routines and updates internal bookkeeping data structures accordingly. Light-PREM also
considers global variables, and other memory segments (e.g. stack, text, data) treating them as chunks.

**Handle detection** As previously defined, variables that are in-scope at the beginning of the function under
analysis are considered *handle* variables. Thus, from this definition it follows that a handle can be either
a global variable or a local function parameter. Since a final prefetch statement will explicitly reference a
handle, the names of such variables must be determined.

Retrieving function parameters is relatively simple and can be done by using code formatting tools and
regular expressions on the source file. Conversely, finding the names of global variables can be nontrivial if
several files are involved in the compilation process. For this reason, this step is performed by analyzing the
executable file produced at the end of the compilation process. Standard Linux tools can be used for this
purpose, as detailed in Section 5.3.3. Once the name of the handles is discovered, what gets recorded is the
 correspondence between handle name and memory address. This last information is important to detect in
which chunk a given handle is contained and thus which other chunk can be reached from it.

**Chunk linking and graph construction** Once the description of each chunk is available, in terms of
start address and size, Light-PREM attempts to link them together. For example, two chunks $C_1$ and $C_2$
are linked if in $C_1$ there exists a pointer to any location inside $C_2$. Performing this operation on each couple
of chunks allows building a directed multigraph which reflects the memory layout of the program under
analysis.

This graph is formally defined as $G = (V, E)$ where $V$ is the set of vertices and $E$ is the set of edges. Each
vertex $C_i$ represents a memory chunk, and it has additional data associated, such as the starting address $a_i$.

---

6This definition of a handle is valid for code written in C.
and its size $s_i$ in bytes. Assuming that there are $n$ memory chunks, $V = \{C_1 = \{a_1, s_1\}, \ldots, C_n = \{a_n, s_n\}\}$. Each edge in the graph represents a pointer that connects two vertices $C_i$ and $C_j$. However, more than one pointer directed to $C_j$ can be found in $C_i$. What differentiates two pointers that are outgoing from the same chunk and are directed to the same node is: (1) the relative position inside the source chunk and (2) the pointed location in the destination chunk. For this reason, each edge holds additional data to keep track of the discussed pieces of information. Respectively, we will refer to them as $\delta^s$ and $\delta^d$.

![Figure 5.5: Memory layout in a sample scenario](image)

Such graph is constructed at the beginning of execution of the target function. Figure 5.5 depicts the graph reflecting the memory layout of a sample application. In the figure, there are four vertex correspondent to four memory chunks. One of them - the vertex represented with a rectangle - is also an handle. For a given pair of vertexes, say $C_i$ and $C_j$ and a given interconnecting edge $\delta_k$ the following property holds:

$$[a_i + \delta^s_k] - \delta^d_k = a_j$$

Where the square brackets denote the “cast & dereference” operator, i.e. what is contained inside the brackets is interpreted as a pointer and accessed to obtain the value of the referenced memory location.

Note for instance that according to the chain of arrows highlighted in green in Figure 5.5, it is possible to reach chunk $C_4$ from the handle $C_1$ with the following expression:

$$[[[a_1 + \delta^s_1] - \delta^d_1 + \delta^s_3] - \delta^d_3 + \delta^s_4] - \delta^d_4 = a_4$$

**Relative expression construction** Once the graph has been built and all the interconnections between the chunks have detected, it is possible to generate the prefetch statements performing graph traversal. Specifically, the procedure for the creation of a prefetch statements accepts three inputs: (1) the memory address accessed inside the predictable function and recorded at run-time; (2) the set of variables that are in-scope (handles) at the beginning of the function; (3) the memory layout of the targeted application expressed as a directed multigraph as explained in the previous section.

Without loss of generality, a variable that is accessed inside a function is either a in-scope variable or a memory location that can be reached with an arbitrarily long chain of dereferences. Light-PREM is able to capture all the interconnections between the different chunks in the memory layout of an executable.
Thereby, for an observed memory access in one of the considered chunks, Light-PREM can build a valid dereference chain from a handle to any memory location recorded at execution time. Such a dereference chain, starting from a handle, is called a relative expression.

Provided the aforementioned oriented multigraph of the memory layout, finding the best dereference chain translates into navigating the graph upwards, i.e. in a reverse direction with respect to the orientation of the edges. As detailed in Section 5.3.3, a combination of two metrics is used to determine which chain is the preferred one in a pair of equivalent dereference chains. Considering the example reported in Figure 5.5, a memory access at a location $\lambda$ positioned at an offset $\delta_\lambda$ inside chunk $C_4$, will lead to the generation of the following prefetch statement:

$$\text{PREFETCH} (\text{handle} \xrightarrow{\delta_1} C_2 \xrightarrow{\delta_2} C_3 \xrightarrow{\delta_4+\delta_\lambda} \lambda)$$

Where the notation $\xrightarrow{\delta_k}$ denotes the dereference operation applied using the offsets $\delta^k_1$ and $\delta^k_2$ as previously explained.

As previously stated, the generated prefetch statements are placed back into the source code of the executable. However, if the memory layout graph of the application under analysis at run-time is different from what observed in the profiling stage, invalid memory references can be encountered while following a dereference chain. In general, it can be the case if there is a dependency between a) the input vector and the allocation order or the size of the chunks; b) the input vector and the existence or position of the interconnecting links (pointers). Unless the targeted code has a well known deterministic structure in this sense, the inherently heuristic nature of some of the processing steps involved in Light-PREM can lead to the generation of invalid memory references.

To address this critical issue, our approach consists of setting up a defensive mechanism. Specifically, a segmentation fault handler (i.e. a handler for the SIGSEGV signal) is installed at the beginning of the memory phase. Furthermore, at process setup, the permissions of those memory pages of the text section corresponding to the memory phase of predictable intervals are changed, so to allow write operations. In this way, if during the memory phase a broken dereference chain triggers a segmentation fault, it is caught by the handler which replaces the faulting instructions with NOPs, so to turn them inoffensive when re-executed after the handling procedure returns. The described handler is used after the refactoring has been completed to remove all those prefetch statements that are found to be faulty upon a change in the input vector. Moreover, since we have observed that it has a negligible overhead in terms of execution time and number of memory references (the handler has less than 10 lines of code), it can be left in production code. This ensures that the correctness of the refactored application is preserved.

**Prefetch aggregation** According to the locality principle, there is an high probability that, if a given memory location is accessed, subsequent accesses will interest surrounding memory locations. For this reason, it is recommended to aggregate prefetch statements that refer to memory locations within a certain radius.

---

<sup>7</sup>This can be done in Linux through the `mprotect` system call.
reason, instead of generating a prefetch statement for every single observed memory access, we aggregate single prefetches in longer prefetch sequences.

The advantage of performing the aggregation step is twofold. First, the number of lines that becomes part of the executable code is minimized. Second, it is important to consider that a single prefetch statement can result in several memory dereferences. Thus, by aggregating them it is possible to calculate the starting position once and to perform the prefetch sequentially with a remarkable run-time improvement.

5.3.3 Implementation Details

In this section, we will detail our implementation of Light-PREM to target general purpose applications written in a language that allow memory manipulation through pointers, like C. As discussed in Section 5.3.2, Light-PREM employs a multistage technique to produce PREM-compliant code out of a legacy application. Due to the fact that PREM, as a proof of concept, is currently implemented in C, the presented Light-PREM sets its focus on programs written in C.

**Access collection**  As previously discussed, the first step performed by Light-PREM is collecting a raw list of all the virtual memory addressed referenced inside the predictable function. For this step, the targeted code is compiled and run inside a profiling environment. Light-PREM can use freely available memory tracing tools (e.g. Lackey, a sub-tool of the Valgrind suite), or the previously described MadT(see Section 5.1) to analyze the task at run-time. The goal is to detect all the memory accesses that are performed inside the body of the function under analysis.

**Chunk detection**  In order to place the accessed memory locations in the appropriate chunk, it is fundamental to detect which memory regions are accessible by the application at the beginning of the predictable function. Such chunks will also represent the nodes of the memory layout graph. As previously mentioned, they are defined by a starting address and size.

At the beginning of the predictable function, there is a set of memory regions that are added to the list of chunks by default. Specifically, these are: (1) the stack area belonging to the predictable function (whose size is always known at run-time); (2) the text area of the executable; (3) .data, .rodata and .bss sections if not empty; (4) global variables and (5) predictable function parameters. Each chunk in the last two categories is also a handle.

Furthermore, to capture all the possible chunks created through dynamic memory allocation routines (malloc, mmap, etc.), Light-PREM intercepts such routines and updates its internal book-keeping data structures accordingly. Intercepting the aforementioned calls without modifying the executable code can be done by temporarily overriding the targeted primitives at dynamic linking time.
Handle detection  In a standard C program, only two kinds of variables are in-scope at the very beginning of a function (before any automatic variable is declared): function parameters and global variables. This is the reason why the only variables that need to be considered as handles are all those variables which fall in one of these two categories.

Since each generated prefetch statement will involve dereferencing a handle, the name of these variables must be determined. Retrieving function parameters can be done in a relatively simple way by using regular expression matching based on the function name. This procedure can be made resilient to different coding styles performing a preliminary pass in a code formatting tool (e.g. Uncrustify). To capture the name of all the global variables, we rely on the data that are contained inside the final executable. In particular, ELF[201] files define a symbol table in which objects with binding attribute set to GLOBAL and default visibility properties refer to global variables. Variables that are global but not exported at linking time (static) are also considered in a similar way. As a final step of the run-time data collection phase, handle variables are recorded in terms of name and memory address.

5.3.4 Prefetch generation

The remaining part of the process involves steps that can be performed offline relying on the information collected during the previous phase.

Graph Construction  As described earlier, while the chunks are the vertex of the memory layout graph, edges represent pointers, i.e. links from one vertex to another. Edges are generated after all vertices (chunks and handles) are discovered and created. This step involves recognizing the links between the chunks and adding the detected vertex with appropriate weights. The algorithm which is used to perform this operation is called “Pointer Probing”. It can be described as follows: let the size of a pointer in the targeted platform be $P$ bytes. For each chunk $C_i$, Light-PREM reads the value $v$ of every contained $P$ byte block. Next, it checks if the given value can be interpreted as a pointer. That is, $v$ should be an address which falls into the range $(a_j, a_j + s_j)$ for some chunk $C_j$.

If a given value $v$ satisfies the above property, an edge is created from chunk $C_i$ to chunk $C_j$. For said edge, the value of $\delta_s$ is the position where $v$ was found in $C_i$. Conversely, $\delta_d$ is the offset in chunk $C_j$ (i.e. $v - a_j$). Note that the graph allows self-loops, since a pointer can point to the same chunk that it resides in. The displacement $\delta_s$ is called pointer offset.

The aforementioned Pointer Probing algorithm takes its name from the fact that chunks are treated as opaque blocks of memory, but $P$ bytes sized blocks are interpreted as pointers and checked against the boundaries of existing chunks. As such, this algorithm may occasionally generate edges that do not actually represent pointers. This can happen if some datum has a value that, when interpreted as an address, coincidentally represents a valid memory address inside some chunk. This issue is resolved by running the
graph construction on several separate executions of the program, and only accepting edges that appear in every execution. Generally, two runs are enough to eliminate false edges, however Light-PREM can be configured to perform more runs to increase the confidence about the validity of the recorded edges. Moreover, additional security measures can be adopted to prevent a faulty prefetch statement from causing an invalid dereference as mentioned in Section 5.3.2.

Relative expressions In the current Light-PREM C-based implementation, a relative expression for an observed memory access at location $\lambda$ is a navigation path inside the memory layout graph that starts from a handle and ends in the chunk which contains $\lambda$. As shown below, any relative expression can be translated using only pointer arithmetic, pointer dereference operations, and name of handle variables. The key insight to relative expressions is that any location of an observed memory access inside one of the considered chunks can be reached using said expressions. For instance, a structure reference `some struct->field` can be expressed as `*(some struct + off)`, where `off` is the constant offset in bytes of `field` from the beginning of the structure.

Relative expressions reduce the complexity of Light-PREM and allow it to stay compiler independent, since the content of memory blocks is considered as opaque binary data.

Relative expression translation Dereference chains are used to create relative expressions. To show how they can be translated to produce the final C-compliant relative expression, let us consider Figure 5.6 where, similarly to Figure 5.5, a sample memory layout is reported, but this time with explicit address and size values for the chunks. Specifically, suppose that the application under analysis allocates two blocks of memory prior to calling the target function, with sizes of 256 bytes and 4 KB respectively.

These chunks get assigned starting addresses of `0xAAAA0000` and `0xBBB0000` respectively. Furthermore, a function parameter named `handle1` is located at `0x1110000` and has value `0xAAAA0001`. Finally some pointers exist that link the three chunks together as depicted in the figure. The figure reports only the most significant 16 bits of each address.

![Figure 5.6: Memory layout in a sample C application](image)

Given the depicted memory layout, the final relative expression includes: (1) a prefetch statement of a
given size; (2) a dereference chain which follows the dashed path highlighted in orange for some location \( \lambda \) at offset \( \delta_\lambda \) in \( C_3 \). In our notation, it becomes:

\[
PREFETCH(handle1^{\delta_1} \rightarrow C_2^{\delta_2+\delta_\lambda} \rightarrow \lambda, \text{size})
\]

The resulting translation with trailing prefetch statement in the targeted language is:

```c
1 \text{void } *C_2 = * (\text{void } **)((\text{void } *)\text{handle1 } +0) -1;
2 \text{void } *C_3 = * (\text{void } **)(C_2 +32) -256;
3 \text{PREFETCH}(C_3 + \text{delta_lambda}, \text{size});
```

Being able to generate relative expressions involves two steps. In the first, detailed in Section 5.3.4 the interconnections between the recorded chunks lead to the construction of an oriented multigraph. In the second step, a reverse graph traversal strategy is employed to decide the optimal path through the vertices to connect a handle with a leaf memory access recorded during the data collection phase. This last step is detailed below.

**Graph traversal strategy**  After the construction of the memory graph, Light-PREM is ready to convert memory accesses (recorded in the very early stages of the process as absolute memory addresses) to relative expressions. To produce the relative expression, Light-PREM simply needs to find a path from a handle to the chunk that the memory access resides in. Finding such a path can be easily achieved by using existing graph traversal algorithms such as Depth First Search (DFS) and Breadth First Search (BFS). However, since the graph represents the memory organization of the application under analysis, additional traversal policy are employed to support the selection of the best path.

To understand the reason, let us consider again the example in Figure 5.6. To convert a memory access inside \( C_3 \), Light-PREM can use the dashed path highlighted in orange, and we have already seen what the correspondent relative expression looks like. However, when multiple paths exist, a simple depth or breath first search does not always yield the best results. In fact, due to the speculative nature of Light-PREM’s analysis, there is no guarantee that a path, and consequently a relative expression, will be valid for all possible inputs to the program. To mitigate these effects, two heuristics are used to improve the performance of Light-PREM. The two heuristics that are used can then be summarized as: (1) requiring all pointer offsets to be non-negative; (2) given two pointer offsets for two edges linking the same chunks, selecting the smaller one.

The first heuristic (non-negative pointer offsets) comes from the fact that typical programs only utilize positive integers in pointer arithmetic. In fact, both array and structure references can be rewritten using non-negative pointer offsets.

The second heuristic arises from the fact that the smaller a pointer offset is, the more likely that the offset remains inside the bounds of the datatype the pointer refers to. In the example of a structure pointer,

---

8 the C-like relative expression is simplified for the ease of the reader, however Light-PREM combines the three statements into one statement
adding a small constant to the pointer is more likely to result in a pointer that is still inside the memory of the structure, as opposed to adding large offsets. A similar argument can be applied to arrays. To solidify this concept, consider the memory graph in Figure 5.7.

![Figure 5.7: Heap Memory Graph with a self-loop](image)

In this graph, there exists a self loop. To travel from handle1 to \( C_3 \), it is preferable to take the self loop in \( C_2 \) on the way to \( C_3 \) as opposed to not taking it. To understand why, consider the relative expressions that these two different paths represent:

**Path with self-loop:**

\[ \text{handle1} \xrightarrow{+0} C_2 \xrightarrow{+0} C_2 \xrightarrow{+10} C_3 \]

**Path without self-loop:**

\[ \text{handle1} \xrightarrow{+0} C_2 \xrightarrow{+210} C_3 \]

In the path without the self-loop, the pointer offset 210 is likely to be dependent on the input of the executable. To see why this might be the case, let us consider an example scenario of the memory layout of the chunk represented by node 1. This scenario is depicted below in Figure 5.8.

![Figure 5.8: Sample organization of virtual memory based on graph from Figure 5.7](image)

In this scenario, there exists a variable sized array in the middle of the chunk, where the size of the array is dependent on the input of the executable or some other unknown factor. Thus the offset of 210 will change.
from run to run, whereas the offset of 10 will not. This means it is preferable to take the pointer (shown as an arrow in the diagram) from 0xAAAA0000 to 0xAAAA00C8 to skip over the variable sized array. This simple heuristic works very effectively.

With these two heuristics in mind, Light-PREM actually performs a reverse depth-first search. That is, it inverts the direction of all edges, and performs a depth-first search starting at the chunk that the memory access resided in, stopping when it finally reaches a handle node. In line with the second heuristic, a path is allowed to go through the same node twice. The path exploration is bounded with a maximum exploration depth to break potentially infinite loops.

5.3.5 Evaluation

In this section we present the evaluation performed on Light-PREM. Three main metrics are used to understand the effectiveness of the proposed methodology: (A) coverage in terms of captured cache misses with respect to the manual PREM approach and to a non-PREM execution; (B) impact on temporal predictability of the execution phase of a task refactored with Light-PREM; (C) time needed to perform the Light-PREM code refactoring. Coverage analysis is carried out on a set of 7 EEMBC benchmarks that are reflective of a real-time scenario for their deterministic properties and code simplicity. To stress our implementation and understand the overhead of our technique, 7 additional and more complex benchmarks are used from the MiBench suite. The same set of EEMBC benchmarks is used to provide an idea of the time required for a full run of Light-PREM.

Testbed Setup Since we want the evaluation to be to comparable with what presented in [16], we configured out testbed in a similar way. Specifically, we used an Intel Q9500 CPU in which we disabled the speculative CPU HW prefetcher since it negatively impacts the predictability of any real-time task. The Q9500 is a quad-core CPU and each pair of cores shares a common level 2 (last level) cache. Each cache is 16-associative with a total size of 6 MiB and a line size of 64 bytes. Since we use a PC platform running a COTS Linux operating system, there are many potential sources of timing noise, such as interrupts, kernel threads and other processes, which must be removed for our measurements to be meaningful. For this reason, in order to best emulate a typical uni-processor embedded real-time platform, we divided the 4 cores in two partitions. The system partition, running on the first pair of cores, receives all interrupts for non-critical devices (e.g., the keyboard) and runs all the system activities and non real-time processes (e.g., the shell we use to run the experiments). The real-time partition runs on the second pair of cores. One core in the real-time partition runs our real-time tasks. The other core is turned off. Note that the cores of the system partition can still produce a small amount of unscheduled bus and main memory accesses or raise rare inter-processor interrupts (IPI) that cannot be easily prevented. However, in our experiments, we found these sources of noise to be negligible.
Another source of unpredictability comes from the fact that the cache does not support deterministic allocation. However, our work can be easily integrated with practical solutions described in [199, 6, 202]. Thereby, we currently mitigate this problem by making sure that in our experiments the amount of prefetched memory is far less than the available cache (6 MiB). Self-evictions are also a minor concern since our platform uses an Intel SmartCache which does not perform a direct mapping from physical addresses to cache lines [203]. The cache is trashed before any measurement is taken in order to perform a worst-case oriented analysis.

Evaluating a task using Light-PREM requires 2 main steps: (A) the Light-PREM refactoring as discussed throughout the previous sections, and (B) the actual task run once Light-PREM has completed its refactoring. In this section, we will refer to A and B using the term Analysis and Run respectively.

Coverage Analysis  As previously stated, the main purpose of the PREM model is to enforce a structure in the memory access pattern of the applications so that all the memory accesses for a particular predictable interval are performed at the beginning, leaving the bus available for scheduling I/O flows or memory phases of tasks scheduled on other CPUs [198].

Manually performing such refactoring requires a deep understanding of the code and often requires non-trivial modifications to the code itself that can break correctness and optimizations. Light-PREM is able to perform this refactoring without changing the existing code and automatically inserting prefetch statements. Also, we have discussed how with Light-PREM we can relax the assumption that the code of the predictable interval is self-contained in a single function. However, the increased flexibility can have a cost. In particular, if the code performs calls to dynamically linked libraries or system calls inside the predictable function, Light-PREM will not be able to generate a corresponding prefetch for those memory accesses and more
cache misses could be experienced during the execution phase. Thereby, the effectiveness of Light-PREM can be measured looking at how many cache misses are being issued during the memory phase and how many are experienced during the execution phase. In this sense, the coverage is defined as the number of misses that are avoided in the execution phase thanks to the inserted prefetches. The desired property would be to have a perfect coverage for simple, self-contained predictable intervals and a sharp reduction of misses for complex code with calls to external functions.

We first analyze 7 benchmarks from the EEMBC automotive benchmark suite [204]. In past work [16], these benchmarks were modified to run multiple times, instead of just once, so that timing could be more accurately measured. The number of iterations needed to complete a pass on the input varies from benchmark to benchmark but this number is always below 10,000 iterations. Thus, 10,000 is the number of iterations used to perform the analysis in order to consume the input. Moreover, the code of the considered EEMBC benchmarks is simple enough so that the core of the computation is almost contained in a single function without having any system call in our candidate predictable interval.

Figure 5.9a shows the comparison between Light-PREM and manual PREM in terms of coverage for the mentioned benchmarks. The ratio between misses captured by Light-PREM and by the manual version with respect to the non-PREM case are reported in the first two bars of each cluster. The plot also reports the number of prefetches issued by Light-PREM as a fraction of prefetched inserted in the manual version in the last bar of each cluster. It is important to notice that in the manual PREM case, the code has been modified to (1) make indirect references explicit for the ease of writing prefetch statements and (2) inline all the functions called in the predictable interval. These modifications are not present in the code analyzed by Light-PREM, hence the slightly worse coverage observed.

Nonetheless, the results shown in the figure highlight that Light-PREM is always able to detect more than 92% of memory accesses (except for cacheb, where even the manual refactoring was not able to achieve a good coverage [10]). Two more behaviors can be noted. First, that, looking at the first pair of bars of each cluster, Light-PREM exhibits a coverage which is always comparable to what was achieved with a manual approach, even though it requires almost no changes to the source code in order to attain its goal. Second, that, according to the third bar of each cluster, the amount of prefetches issued by Light-PREM is always comparable to or much less than what results with a manual approach. This is because the generation of prefetch statements is driven by the observed memory accesses rather than reasoning about what portions of code/data can be reached/accessed.

The same evaluation approach has been used to understand the performance of Light-PREM on more complex code bases. Specifically, we ran our analysis on a set of applications from the MiBench suite [197]. In this case, the benchmarks have a much more complex code which brings them closer to consumer applications rather than real-time tasks. Nonetheless, it is interesting to understand the capabilities of Light-PREM to

---

9 The number of cache misses is retrieved from the hardware performance counters relying on the `rdpmc` instruction.

10 This is mainly because this particular benchmark performs function calls using function pointers inside the predictable function.
extract prefetch statements out of highly connected memory layouts.

Figure 5.9 shows the results obtained for: the Jpeg compression benchmark considering an input of 1 and 8 Mega Pixels; the Qsort vector sorting application; the Susan image processing benchmark that can be used to detect corners, edges in the inputted image or to smooth it. As reported in the table, Light-PREM achieves a high coverage for the Jpeg, Qsort benchmark and Susan with edge detection. In the last two tests, performance gets worse because the algorithms for corner detection and image smoothing, as they are, heavily rely on Libc library functions to perform memory and arithmetical operations. Accesses inside such memory areas cannot be captured in the current Light-PREM implementation.

Specifically, in the case of Susan with corner detection, the number of captured memory accesses is low. This results in a small number of produced prefetch statements and consequently to a low coverage (around 70%). Similarly, the smoothing algorithm of the Susan benchmark causes Light-PREM to build long dereference chains to reach some of the accesses in Libc areas, generating a large number of prefetches which still result in a low coverage. The problem could be solved by statically linking the Libc library but this would affect the non-predictable part of the executable. Instead, as a part of our future work, we plan to include in Light-PREM a strategy to statically link only the subset of library functions used in the predictable interval.

Temporal Predictability The aim of Light-PREM is to automatically refactor the targeted code without relying on compile-time knowledge in a way that the resulting application is compliant to the PREM model. It follows by the definition of PREM (PRedictable Execution Model) that predictability is one of the key features that has to be provided. An experimental way, to evaluate how predictable a fragment of code is, consists in running it a large number of times and observe how stable its timing properties are. Specifically, we are interested in understanding the timing properties of the execution phase in a predictable interval.

As shown in the previous section, Light-PREM is able to achieve a good coverage. This means that the number of bus accesses performed during the execution phase is sharply reduced. Thus, we argue that as a result the code executed inside the predictable interval exhibits better predictability. To evaluate this, bus activity is generated on the system partition of the testbed using an interfering task, which consists of a synthetic benchmark that generates high-bandwidth DRAM traffic.

The evaluation has been performed on the full set of benchmarks presented in the previous section. The last two Susan benchmarks are excluded from this analysis since Light-PREM does not achieve a good coverage and thus they are not representative for this experiment. Figure 5.10 reports average time duration, as well as variance, of the execution phase in the considered benchmarks. For each of them, the comparison is with the non-PREM (legacy) case.

As shown in the figure, the prefetches inserted by Light-PREM have two effects. First they sharply decrease the average execution time. Second they contribute to an improvement for the observed variance

11 We have seen that doing so would boost the coverage of Light-PREM to 97%
Figure 5.10: Exec. phase runtime and variance for Light-PREM and non-PREM.

in the execution time of the considered code fragment. This demonstrates the effectiveness of Light-PREM as a way to increase the predictability in terms of timing behavior, as well as memory access pattern, in line with the advantages of the PREM model and subject to its limitations, as discussed in [16].

**Analysis Runtime** Light-PREM involves performing several computation steps to carry on a complete analysis of the targeted executable. Some of these steps involve running the actual code to collect memory accesses and chunks, while other perform post-processing on acquired data.

As previously mentioned, memory access collection is performed using Valgrind which instruments the profiled code at runtime. This intuitively means that there is an expected overhead when running the observed code in the profiling environment. It is worth to mention that in earlier implementations of our technique, the analysis runtime could easily take hours even for simple benchmarks like a2time. In the latest version, however, the running time has been sharply reduced to the order of tens of minutes. The main improvements involved (1) patching Valgrind to analyze memory accesses for a selected fragment of code only and (2) postponing the actual data processing after the data collection phase has complete to avoid unnecessary slowdown.

Figure 5.11 shows the ratio between the time required to perform the analysis of a given task and the runtime of the predictable interval of the task itself. This gives an idea of how the analysis technique scales as the complexity of the profiled application increases. It can be noted that, although the trend is linear, it can require minutes to analyze predictable intervals with execution time in the order of seconds. However, since the Light-PREM analysis and refactoring need to be done offline only once per task, the incurred cost not only does not represent a limitation to its applicability, but it is also a significant improvement.
Figure 5.11: Trend of time required to perform the Light-PREM analysis (in seconds) compared to the execution time of the benchmark (in milliseconds). Benchmarks ordered by runtime.

considering that the time needed to perform manual refactoring could easily take days.

5.4 Summary

As we have extensively discussed in Chapter 3 and 4, memory resources have a determinant role on the performance and schedulability of real-time applications in multi-core systems. Hence, shifting the attention from CPU-time management to memory management is a fundamental requirement in modern scheduling theory. Nonetheless, comparatively less techniques and tools exist in the literature to accurately characterize and profile how memory resources are used by real-time applications. This chapter represents an attempt in bridging the gap between (i) what is required to perform memory-centric schedulability analysis and (ii) what is available to perform memory usage analysis of application workload.

First, we presented a novel technique for memory access detection and propose the implementation of a proof-of-concept tool, namely MadT, that is capable of recording memory accesses performed by an application in a fully symbolic form. In addition, our tool is capable of accurately tracking dynamic memory allocations and releases, forging when necessary new symbol names that simplify the identification of accesses to the dynamic memory. By using MadT system designers can accurately relate all memory accesses, including those to dynamically allocated blocks, to the source code of the target applications. Compared to other profiling tools, MadT shows similar performances, and in some cases it is significantly faster. Furthermore, MadT does not require modifications to the source code of the application or of the operating system.

By using MadT or commercial memory tracing tools, we have demonstrated that it is possible to: (i)
easily derive a ranked list of frequently accessed memory regions that is beneficial to allocate in cache; and (ii) construct a profile offline that can be used online to perform cache allocation via Colored Lockdown (see Section 3.2). Being able to analyze the task offline and leverage the gathered profile information to perform cache allocation is fundamental to maximize the benefit of data locked in cache. In Section 2.2 we have discussed the importance of cache replacement policies in caches. When explicit allocation via locking is performed, the OS is effectively overriding the cache replacement policy provided by the hardware. In many senses, accurate profiling allows us to implement an ideal replacement policy, based on the knowledge of the data required by an application throughout its execution.

Finally, we have discussed how structured task execution can be exploited to perform multi-resource co-scheduling. In this case, we referred to a two-phase load-execution model, called PREM (see Section 5.3.1). Enforcing the PREM model on a group of real-time tasks allows performing a high-level coarse grained scheduling of CPU time and memory bus. In this way it becomes possible to regulate the contention for accessing such resources, with measurable improvements in terms of predictability. In this context, we have presented Light-PREM: a software technique to automatically generate the prefetch statements that are needed to perform a legacy-to-PREM porting of a given executable. Light-PREM leverages on profiling and memory analysis strategies to produce a PREM-compliant code in an automatic fashion and without almost any knowledge about the semantics of the ported code. We have tested the proposed technique on a real testbed. The presented results effectively show that Light-PREM is able to achieve performances that are comparable (and in some cases superior) to a manual core re-engineering approach.

5.5 Future Work

Our work on application behavior inspection has mostly focused on tracing-based memory analysis for specific applications. We envision that workload automated characterization of real-time tasks can be greatly extended in a number of directions.

A first direction is input and execution variability. Both the proof-of-concept implementations in Section 5.2 and 5.3 only operate under the assumption that an arbitrary input vector can be used offline to derive profiles that are representative of the application behavior online. This is true only for relatively simple control tasks, and may not hold in general for complex application workload. A possible approach to overcome the issue could consist of three main steps: (i) repeat the analysis for a set of input vectors that provide a good coverage of the possible execution paths. In this way it would be possible to associate specific information about the usage of memory resources per each branch of the execution graph; (ii) identify key variables in the state of the task that determine major branches in the execution graph; and (iii) based on the state of the key variables, perform memory prefetch and/or locking using the data derives in the first step. In order to attain this goal, we firmly believe that a hybrid profiling approach would need to be taken, i.e. a combination of memory tracing and compiler-based techniques.
Secondly, as a next-generation real-time systems will increasingly rely on dedicated accelerators, such as GPUs and DSPs, it is important to extend profiling techniques to operate on heterogeneous platforms. On such, platforms, not only it will be important to characterize of CPUs access the memory, but also how other bus masters use memory resources. Unlike PCI-express GPUs, modern SoCs for self-driving car technology implement GPUs that directly access main memory[^12]. Two main challenges, however, need to be tackled to extend profiling to heterogeneous platforms. First, complete technical documents for specialized hardware components are not publicly available as such components constitute intellectual property. Second, the monitoring and debugging capabilities of hardware accelerators greatly vary across vendors. As such, it may not be possible to implement techniques that are generic enough to have a significant impact on industry. A thorough investigation of the problem, however, is still fundamental to understand what hardware features are necessary to perform proposed workload characterization.

Finally, in the case of cache-based architectures, we have explored a technique for profiling data for last-level cache allocation. For simplicity, in this work we have assumed that the last-level cache is the only level of cache in the system, or that higher cache levels have been disabled. This is not true in general. Additionally, consider the following: (i) different cache levels significantly vary in size; and (ii) suppose that cache look-ups for a certain memory address always hit in a lower cache level, say L1. Then, there is not need to allocate said memory in L2 (or LLC), despite the considered memory block is frequently accessed. In light of these considerations, profiling can be extended to accurately drive cache allocation for multiple levels of cache.

[^12]: See Nvidia Tegra K1 [205] and Nvidia Tegra X1 SoCs [206]
CHAPTER 6
CONCLUSION

In this chapter, we first summarize the contribution of this work to the state of the art in techniques to deploy hard real-time systems on modern multi-core platforms. Next, getting inspiration from the many lessons learned, we outline a set of recommendation for future generation of processing platforms in term of desirable features and design principles. Albeit our focus was on multi-core platforms, in the next sections we will discuss how other popular trends in hardware manufacturing could, in the near future, represent viable options for safety-critical systems. Specifically, we will discuss many-core, heterogeneous and re-configurable platforms. Finally, we will conclude with a summary of open challenges that we deem important to address to further progress the field of embedded and real-time systems.

6.1 Summary of Contribution

In our research, we have addressed the problem of deploying safety-critical systems on commercially available multi-core platforms. We have investigate a set of orthogonal techniques for two important trends in modern multi-core manufacturing: (i) cache-based architectures and (ii) scratchpad-based architectures. We have also discussed how application profiling and characterization play a fundamental role in achieving efficient resource management.

In the first approach (cache-based platforms, see Chapter 3), we have valued system engineering time over silicon cost and reasoned on transparent technologies that can be used to create a seamless OS layer of resource partitioning techniques. At a logical level, the multi-core platform exported by the layer can be seen as a collection of single-core equivalent machines, hence the name Single-Core Equivalence. In this way, consolidated software that was previously developed and certified for single-core systems can be migrated to a multi-core platform without expensive re-engineering. Conversely, our approach targeting scratchpad-based systems (see Chapter 4) took a different direction. In fact, we devised a set of techniques that are less constrained with backward compatibility. Achieving real-time determinism in such systems can be done by re-structuring the task model and exploiting hardware resource specialization. Finally, the work described in Chapter 5 investigates aspects of application behavior inspection, i.e. the process of analyzing application-specific usage of memory resources to (i) optimize allocation of cache space, or to (ii) re-structure the application’s memory access pattern. We have shown that profiling can be largely done automatically,
relying on workload tracing techniques.

6.2 Recommendation for Next-generation Computing Platforms

The first round of investigation for this research started back in 2012. Roughly 4 years down the line, we have observed a boom in the multi-core embedded industry. Compared to the very expensive and relatively low-performance systems available few years ago, what is available today is different under a number of aspects: (i) from a performance standpoint, clock frequencies have not increased much, but modern embedded systems have heavily followed the route of specialization. They now include dedicated co-processors and fast on-chip memory resources. A steep increase in the number of cores has also been observed. For instance, the evolution of the Freescale/NXP P4080 8-core platform used in our evaluations, namely Freescale/NXP T4240, features 12 cores (24 threads) with 6 MB of L2 memory and 1.5 MB of L3 cache, configurable as SPM; (ii) cost-wise, the price of more advanced hardware has followed the fall in price that has been observed in the general-purpose market; (iii) the industry, especially in platforms targeting the automotive domain, has demonstrated an overall inclination to test in production a number of architectural innovation. For instance, the Freescale/NXP MPC4777M used in our evaluation has used the idea of dividing memory and I/O bus. At the same time, ARM Cortex-A15 platforms and successive families have introduced support for hardware virtualization. Hardware performance monitoring is also considered a key feature for modern embedded platforms.

Given the rate at which the design and production of embedded systems is progressing, we hereby make a number of recommendation for future generations of multi-core platforms that we deem fundamental to facilitate the development of safety-critical systems.

**Fusion of performance monitoring and control**

One of the issues that we have faced in the development of our techniques is that often the hardware provides extensive support for gathering performance data, but few ways to take actions without introducing excessive overhead. For instance, it is possible to gather information about the number of cache hits in a given interval of execution, but it is relatively hard to instruct the cache to automatically prevent eviction of frequently accessed memory regions. Similarly, it is possible to derive the memory bandwidth consumed by a given core (or application). However, taking any decision on the state of the running entity currently involves going through an expensive interrupt handling path. For this reason, an effort should be made by manufacturers to provide not only fine-grained monitoring capabilities, as it is currently the trend, but also additional logic to take decisions based on simple threshold-based logic in hardware, with minimal intervention of the main CPUs.

**Fine grained cache management**

Cache memories are crucial resources from performance optimization standpoint. The traditional ap-
approach to caches is heavily borrowed from general-purpose computing and provides no guarantees for worst-case performance. Either a complete redesign should be made in the direction of predictability, or substantial cache management capabilities should be provided in hardware. We have seen a number of platforms that have already undertaken the challenge of adapting the cache design to support real-time cache management. In fact, the PowerPC-based line of Freescale/NXP QorIQ SoCs features atomic cache-line locking. Unfortunately, however, we have seen the opposite trend in ARM-based platforms. Originally, with ARM Cortex-A9, support for lockdown by master (see Section 3.1) was introduced. Successive architectures (from ARM Cortex-A15 ahead) have effectively removed support for cache locking and do not even allow to disable L1-data and L2 cache independently \[207\]. As a result, most of the current research on cache management has focused on PowerPC systems or relatively outdated ARM Cortex-A9 SoCs.

Support for virtualized scratchpad memory

There has been an increase in the popularity of scratchpad memories in multi-core architectures targeting the embedded systems market. Scratchpad memories provide the benefits of caches and the ability to be fully managed by OS and applications. One of the limitations that we have encountered during our work, however, is the lack of virtualization. Often, scratchpad-based micro-controllers only provide memory protection units (MPUs) to manage scratchpad memory ranges, meaning that applications can be prevented from corrupting each others in memory, but they still need to be compiled with hardcoded non-overlapping physical addresses. While this may not represent a problem for relocation in scratchpad, as we have shown in Section 4.2.5, the lack of support for memory virtualization does complicate scratchpad management. Consider a relocatable task, where each data (and instruction) access is relative to the value of a known offset or register. As the task executes, it performs relative-to-absolute translation of memory locations and may temporarily store an intermediate absolute value used to calculate the final address in a general purpose CPU register. If the task is preempted and relocated on a different portion of scratchpad memory, however, such an absolute value would be incorrect, leading to a logic misbehavior. Support for virtualization of scratchpad space, would allow to lift a number of constraints for fine-granularity temporal management of on-chip memory.

Cache prefetching engine

Caches have the advantage to speed-up unmanaged non-critical applications in a transparent manner. Hence, for systems where the average performance of non-safety-critical workload is important, the presence of caches can be a more desirable feature than scratchpads. As mentioned above, the presence of cache locking mechanisms enables a set of management schemes. The main problem with prefetch and locking, however, is that they need to be performed synchronously with CPU execution. If the target cache is a shared cache, then the operation can be carried out by any of the general
purpose cores, but it still translates into a CPU time overhead. An alternative could be integrating a dedicated co-processor that can perform cache prefetching and locking in parallel and in background with executions flows on the main CPUs. The co-processor would largely behave as a DMA engine, implementing a scatter&gather scheme and targeting the shared cache or the coherent interconnect for private cache levels as a destination for data transactions. A work in this direction was presented in [208], but a generalization of the approach could be beneficial to broaden the spectrum of feasible management schemes.

**Decentralized I/O buffers**

In Section 4.2.3 we have stressed the importance of preventing I/O transactions to asynchronously initiate transactions on the main memory bus. One of the approaches followed in some modern chips, like the one used for our evaluations, is to introduce a separate bus for all the I/O communication. We have shown that, as long as I/O traffic does not saturate the dedicated bus, a portion of scratchpad memory attached to the I/O bus can be used as a buffer to regulate I/O traffic. The main limitation of this approach is that a misbehaving device could saturate the I/O bus, affecting the rest of the system. A better approach would be providing decentralized buffers where the OS can instruct the I/O devices to retain the data before allowing them to transmit on the main bus. In this way, devices are not throttled (assuming that device buffers are large enough), while the OS remains in full control of the main memory bus.

### 6.3 Many-core Platforms

Recently, there has been an uptrend in the popularity of many-core systems. Many-core platforms feature a large number of computing elements grouped into clusters with local memory and I/O resources. The cores are interconnected via a Network-on-Chip (NoC) that can be reconfigured in terms of request routing and flow priority management.

At a first glance, a many-core system may appear as even more challenging than multicore SoCs to deploy safety-critical systems. This is in general true, but recent many-core architecture designs have gone in the direction of precision-timed platforms. This approach is known as Multi-Purpose Processing Array (MPPA). MPPA many-core platforms, in fact, are designed to ensure composability inside each computing cluster, and then to combine cluster-level computation with NoC prioritized communication. Thanks to the careful design, timing analysis is simplified by the compute-cluster micro-architecture.

An example of MPPA architecture is the Kalray MPPA-256 Bostan\(^1\). The SoC architecture features 16 compute clusters and 2 I/O clusters. Each cluster includes a local scratchpad memory (SMEM) shared by the 16 processing elements in the cluster, or by the 4 cores in each I/O cluster. Only the processors in

---

the I/O clusters can directly address the external DRAM memory. The mentioned MPPA many-core has a number of desirable features: (i) implementation of the same very-long instruction word (VLIW) ISA for all the cores, including those in the I/O clusters; (ii) adoption of a distributed memory organization, with local scratchpads close to the processing elements; (iii) local scratchpad memories with independent memory banks and configurable address mapping, as a form of simplified virtualization; (iv) core-private access buses with round-robin arbitration; (v) presence of a double NoC with configurable routes and flow priorities; and (vi) connection of I/O clusters with PCI-Express, DDR, and Ethernet network controllers. It has been shown that the Kalray VLIW core is timing-compositional, meaning that it does not exhibit timing anomalies. Thanks to these properties, static timing analysis is precise and tight, and can be safely derived by reasoning on local worst-cases only [209].

Nonetheless, a number of challenges need to be addressed for a provably safe deployment of safety-critical systems on many-core architectures. Predictability on these platforms, in fact, depend on the sound interaction of a number of software components deployed on a number of different computing elements. Hence, it is important to (i) model the behavior of each element involved in computation and communication; and (ii) develop an end-to-end analysis capturing the behavior of processing elements, local memories, NoC routers, I/O processors, DRAM controller and I/O devices. It is also necessary to define a clear migration path for the large amount of legacy certified single-core software currently employed by the avionic and automotive industry.

6.4 Heterogeneous Platforms

Heterogeneous multi-core platforms embed a number of different types of processing elements, each specialized to optimize a set of functions, or providing a different trade-off between performance and energy efficiency. As such, each core may feature different ISAs, clock speeds and capabilities in terms of interaction with the rest of the system. Following the rapid development in chip manufacturing, heterogeneous multi-core designs represent a valid response to the increasing demand for performance and energy efficiency. For instance, ARM has recently launched a heterogeneous multi-core SoC, namely big.LITTLE\(^2\) which has been used to develop state-of-the-art smartphones. The “big.LITTLE” architecture features two types of cores: (i) high-performance (“big”) cores; and (ii) power-efficient (“LITTLE”) cores. Since both types of cores share the same ISA, tasks can be migrated from one to the other, who also share a coherent interconnect. Meanwhile, differences in the internal micro-architecture of big and LITTLE cores allow them to establish different trade-offs in terms of clock speeds and power consumption.

Another typical example where heterogeneous platforms are gaining in popularity is image processing and vector graphics. In this context, performance, power efficiency and scalability in multi-core platforms are achieved through the introduction of Graphic Processing Units (GPUs). GPUs are effectively processors

\(^2\)See https://www.arm.com/products/processors/technologies/biglittleprocessing.php
optimized for vector-oriented, with an heavy single-instruction, multiple-data (SIMD) computation model and feature up to thousands of parallel compute cores. This makes GPUs ideally suited for computing tasks that process large data sets and have intensive arithmetic computation requirements. Vector-oriented processing on a dedicated engine, however, is not always beneficial due to timing overheads that arise from the CPU-GPU communication.

Scheduling, intended as multi-resource management, on heterogeneous multi-core platforms adds more challenges on top of traditional multi-core scheduling. On one hand, the processing speed depends on not the core type where task is being scheduled. On the other hand, a different number of parallel cores exists per each different processor type. This means that tasks that are logically independent can progress in parallel on one type of processor but need to be serialized on a second type. Hence, they are structured in phases to be co-scheduled on a number of different resources. When the different power profiles and the different memory access patterns are considered, the co-scheduling problem can easily become intractable. In recent years, however, the interest for heterogeneous multicore scheduling has spiked in the real-time community. This is because such platforms attract an increasing number of industrial players especially in the context of technology for self-driving cars and unmanned vehicles (UVs) in general, that have heavy requirements for image processing. Despite some important results have been achieved on the scheduling of multi-phase tasks [190, 210, 211, 212], the general problem is hardened by the fact that optimal scheduling of multi-phase tasks on multiple resources is NP-hard [210].

From a system perspective, the challenges for the implementation of hard-real time systems on heterogeneous platforms arise from the difficulty to predictably manage a set of diversified processing units. In fact, when real-time tasks require a multitude of hardware resources to operate, it is important to model the timing behavior of each unit (e.g. low-power cores, GPUs, DSPs, DMAs). At the same time, an understanding is necessary about how all the different processing elements access and use shared hardware resources, such as main memory and buses.

### 6.5 Re-configurable Platforms

In Section 3.5, we have discussed the Single Core Equivalence (SCE) [8, 9] technology. Under SCE, access to shared memory resources is strictly regulated using a set of OS-level techniques. For software certification, SCE allows \( m \) cores to be treated as independent processors (i.e. conventional single-core chips), meaning that local workload changes require only local re-validation.

What we have learned from the R&D of SCE is that modern multi-core platforms lack efficient resource control hardware primitives. This causes significant complexity and inefficiency when creating the SCE abstraction. New generation multi-core hardware has started to incorporate programmable logic (PL). We firmly believe that this new type of Multi-Core platforms with re-Programmable Logic (MC+PL) provides an unique opportunity in the history of real-time computing to create novel embedded platforms which
leverage the rapid improvements of computing throughput and provide near-optimum real-time resource management.

We expect that in the near future, MC+PL platforms will attract significant interest from the embedded systems community. These platforms offer the opportunity to further improve the technology behind SCE, re-thinking the role of software and hardware components and embedding resource management primitives at a much deeper level than what was previously done. More specifically, strong resource management can be greatly improved and extended by exploiting hardware/software co-design. For instance, it could be possible to implement in hardware the missing resource control primitives and effectively enhance existing architectures for use in safety-critical applications. Some work in this direction has been proposed in [213, 214, 19].

In this sense, it could be possible to design and develop a new generation of resource management techniques by exploiting the programmable logic available on new MC+PL platforms. The predicted improvements over the state-of-the-art would fall in three main categories: (i) achieve significant performance improvement in terms of schedulable loads, and in terms of inter-core isolation; (ii) achieve significant safety improvement by designing finer granularity, hardware-based resource management techniques to better monitor application misbehavior and to strengthen platform-wide security – e.g. early detection of security intrusions that can tamper with the resource management mechanisms as well as safety guarantees; and (iii) provide a migration path for the embedded industry that can safely and predictably use MC+PL systems to implement next generation real-time embedded applications.

6.6 Current Limitations and Open Challenges

The majority of the proposed techniques have been implemented in an experimental setting, revealing the feasibility of the proposed design concepts. However, as it often happens, evolving a technology from the stage of a proof-of-concept to maturity for production purposes represents a big leap. Additionally, software-only resource management techniques that are commonly used in real-time operating systems have intrinsic limitations. Such limitations still represent an obstacle for engineers who want to predictably and securely deploy modern multi-core and SoC platforms for real-time embedded applications. Overall, we can identify the following limitations:

- **L1**: A prerequisite for multi-resource management is the ability to monitor and control not only the state of the CPU but also of the memory hierarchy, I/O devices, and more. Currently, modern multi-core platforms lack support for efficiently monitoring memory traffic system-wide. Introduce the missing support for memory traffic monitoring and control is key for efficient resource partitioning.

- **L2**: Software controlled resource management is limited by the granularity of bandwidth accounting and it suffers from the overhead required to perform traffic shaping. As a result, a performance loss
in terms of schedulable resource utilization is commonly experienced. For instance, according to SCE, memory bandwidth management is performed at the granularity of 1 msec; as a consequence, this limitation introduces a penalty in the schedulability analysis (in the form of blocking).

- **L3:** Software controlled resource management is prone to attacks and it can be compromised. As an example, consider Direct Memory Access (DMA) engines available on all commercial platforms. Their design has had minor modifications since their introduction in computer architectures. However, the malicious use of a DMA engine can easily saturate the available interconnect and memory bandwidth causing the starvation of memory requests issued by the cores. Notions like criticality level, Quality of Service (QoS), traffic shaping, and priority are marginally supported or not supported at all.

In light of this discussion on given hat described so far, we hereby summarize some of the research directions that require particular attention in the near future.

**Inter-processor communication:** Many of the presented techniques are designed to enforce core-to-core isolation. However, there is no discussion about how inter-core communication can be supported while still ensuring time predictability among the communicating entities. For this reason, a more comprehensive SCE solution should include deterministic inter-process/inter-core communication primitives that expose backward compatible interfaces. Similarly, the problem of providing inter-core communication capabilities to a system designed around the idea of resource co-scheduling needs to be addressed.

**Extension of IMA to multi-core:** We envision that Integrated Modular Avionics (IMA) can be ported to operate on multi-core systems when SCE is used at an OS-level to export deterministic logical single-core machines. IMA, however, imposes strict constraints about how partitions are arranged in a major cycle, activated and how I/O data are handled. In order to speed-up the adoption of SCE, thus, it is important to develop a mapping between existing components in IMA systems and its multi-core counterpart where SCE is used for performance isolation purposes.

**Provably correct OS component:** A fundamental issue with safety-critical software is that, despite the solidity of the proposed technology, its implementation can be error-prone. Moreover, multi-core platforms embed circuitry that is substantially more complex than single-core systems. Consequently the probability of a transient or permanent software/hardware faults is higher. It follows that being able to establish “trust” in the dependability of multi-core systems is a strong requirement toward their broad adoption. From a software perspective, formal verification is a viable option. Remarkable progress has been made toward the design of verifiable OS’s [180, 215], as well as in the verification of OS-level components [216, 217]. Additionally, significant results have been achieved in machine-aided identification of bugs in system-level code [218]. Nonetheless, significant effort is required in the verification of OS-level components that implement intercore performance isolation mechanisms, such as: bank-aware memory allocators, cache partitioning and allocation logic, memory bandwidth regulation, and time-deterministic inter-core communication logic, to name a few. It is important to underline that, traditionally, many verification attempts have focused on the
logic correctness of the targeted components. However, in order to verify the impact of OS-level components on the temporal behavior of applications, a whole new set of challenges needs to be addressed.

**Safety Nets:** Verification alone only solves part of the problem. In fact, only a subset of the functional and non-functional properties can be verified in a real system. For this reason, future SoCs and OS’s intended for large-scale safety-critical systems will be expected to employ safety nets. Safety nets can be intended as a comprehensive combination of fault-tolerance and recovery techniques to provide strict system-level behavioral guarantees in spite of hardware and software faults. The design of safety nets that can provide sufficient system safety is currently an open challenge.

**Careful performance optimization:** As a final, necessary step into technology maturation, optimization considerations about the proposed implementation need to be undertaken. The current prototypes, in fact, have been produced by integrating novel component into legacy code-bases (e.g. the general-purpose Linux kernel). We believe that significant performance benefit in the form of overhead reduction can be achieved by (i) carefully refining the interaction between novel and legacy components, and (ii) by exploiting advanced hardware support in current and future generation of multi-core platforms.
REFERENCES


“ARM Cortex-A15 - technical reference manual.”


