MEMORY OPTIMIZATIONS FOR TIME-PREDICTABLE EMBEDDED SOFTWARE

VIVY SUHENDRA

NATIONAL UNIVERSITY OF SINGAPORE

2009

MEMORY OPTIMIZATIONS FOR TIME-PREDICTABLE EMBEDDED SOFTWARE

VIVY SUHENDRA

(B.Comp.(Hons.), NUS)

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2009

Acknowledgements

My gratitude goes to both of my supervisors, Dr. Abhik and Dr. Tulika, for their firm and attentive guidance throughout my candidature. Their joint supervision has enabled me to see from different perspectives and to adopt different styles, lending breadth and depth to our research work. Their advices have also led me into many valuable experiences in the form of projects, internship, teaching.

I am also fortunate to have interacted with wonderful and fun labmates, from my first years with the Programming Languages Lab to my final years with the Embedded Systems Lab. They have truly been great company at work and at play.

Lastly, I dedicate this thesis to my parents, the very personification of love and the ever most important presence in my life.

Contents

Acknowledgements **i**

Contents **ii**

Abstract **vii**

Related Publications **ix**

List of Tables **x**

List of Figures **xi**

1 Introduction **1**

1.1 Motivation 1

1.1.1 Real-Time Systems 2

1.1.2 Memory Optimization 3

1.2 Thesis Statement 7

1.3 Thesis Organization 8

2 Background **10**

2.1 Cache 10

2.1.1 Cache Mechanism 10

2.1.2 Cache Locking 12

2.1.3 Cache Partitioning 13

2.2 Scratchpad Memory 14

2.3 Worst-Case Execution Time 16

2.4 Integer Linear Programming 17

3 Literature Review **21**

3.1 Cache Analysis 21

3.2 Software-Controlled Caching 23

3.3 Scratchpad Allocation 25

3.4 Integrated Cache / Scratchpad Utilization 29

3.5 Memory Hierarchy Design Exploration 29

3.6 Worst-Case Optimizations in Other Fields 31

4 Worst-Case Execution Time Analysis **32**

4.1 Overview 32

4.1.1 Flow Analysis 33

4.1.2 Micro-Architectural Modeling 34

4.1.3 WCET Calculation 36

4.2 WCET Analysis with Infeasible Path Detection 37

4.2.1 Infeasible Path Information 38

4.2.2 Exploiting Infeasible Path Information in WCET Calculation . . 43

4.2.3 Tightness of Estimation 48

4.3 Chapter Summary 52

5 Predictable Shared Cache Management 53

5.1 Introduction 53

5.2 System Settings 56

5.3 Memory Management Schemes 57

5.3.1 Static Locking, No Partition (SN) 58

5.3.2 Static Locking, Core-based Partition (SC) 59

5.3.3 Dynamic Locking, Task-based Partition (DT) 60

5.3.4 Dynamic Locking, Core-based Partition (DC) 60

5.4 Experimental Evaluation 61

5.5 Chapter Summary 67

6 Scratchpad Allocation for Sequential Applications 68

6.1 Introduction 68

6.2 Optimal Allocation via ILP 70

6.3 Allocation via Customized Search 72

6.3.1 Branch-and-Bound Search 75

6.3.2 Greedy Heuristic 78

6.4 Experimental Evaluation 79

6.5 Chapter Summary 85

7 Scratchpad Allocation for Concurrent Applications 86

7.1 Introduction 87

7.2 Problem Formulation 92

7.2.1 Application Model 92

7.2.2 Response Time 94

7.2.3 Scratchpad Allocation 95

7.3 Method Overview 98

7.3.1 Task Analysis 100

7.3.2 WCRT Analysis 101

7.3.3 Scratchpad Sharing Scheme and Allocation 103

7.3.4 Post-Allocation Analysis 104

7.4 Allocation Methods 106

7.4.1 Profile-based Knapsack (PK) 108

7.4.2 Interference Clustering (IC) 113

7.4.3 Graph Coloring (GC) 115

7.4.4 Critical Path Interference Reduction (CR) 117

7.5 Experimental Evaluation 122

7.6 Extension to Message Sequence Graph 126

7.7 Method Scalability 131

7.8 Chapter Summary 136

8 Integrated Scratchpad Allocation and Task Scheduling 137

8.1 Introduction 137

8.2 Task Mapping and Scheduling 138

8.3 Problem Formulation 141

8.4 Method Illustration 144

8.5 Integer Linear Programming Formulation 147

8.5.1 Task Mapping/Scheduling 148

8.5.2 Pipelined Scheduling 151

8.5.3 Scratchpad Partitioning and Data Allocation 156

8.6 Experimental Evaluation 159

8.7 Chapter Summary 165

9 Conclusion 166

9.1 Thesis Contributions 166

9.2 Future Directions 167

Bibliography 169

Abstract

Real-time constraints place a requirement on systems to accomplish their assigned functionality in a certain timeframe. This requirement is critical for hard real-time applications, such as safety device controllers, where the system behavior in the worst case determines the system feasibility with respect to timing specifications. There is often a need to improve this worst-case performance to realize the system with efficient use of system resources. The rule remains, however, that all impacts of performance enhancement done to the system should not compromise its *timing predictability* — the property that its performance can be bounded and guaranteed to meet its timing constraints under all possible scenarios.

Due to the yet-to-be-resolved gap between the performance of processor and memory technology, memory accesses remain the reigning performance bottleneck of most applications today. Embedded systems generally include fast memory on-chip to speed up execution time. To utilize this resource for optimal performance gain, it is crucial to design a suitable management scheme. Popular approaches targeted at enhancing average-case performance, typically done via profiling, cannot be directly adapted to effectively improve worst-case performance, due to the inherent possibility of worst-case execution path shift. There is thus a need for new approaches specifically targeted at optimizing worst-case performance in a time-predictable manner.

With that premise, this thesis presents and evaluates memory optimization techniques to improve the worst-case performance while preserving timing predictability of real-time embedded software. The first issue we discuss is time-predictable management schemes for *shared caches*. We examine alternatives for combined employment of the popular mechanisms *cache locking* and *cache partitioning*. The comparative evaluation of their performance on applications with various characteristics serves as design guidelines for shared cache management on real-time systems. This study complements existing researches on predictable caching that have been largely focused on private caches.

The remaining of the thesis focuses on the utilization of *scratchpad memory*, which has inherently time-predictable characteristics and is thus particularly suited for real-time systems. We present optimal as well as heuristic-based scratchpad allocation techniques aimed at minimizing the worst-case execution time of sequential applications. The techniques address the phenomenon of worst-case execution path shift and target the global, rather than local, optimum. The discussion that follows extends the concern to scratchpad allocation for concurrent multitasking applications. We design flexible space-sharing and time-multiplexing schemes based on task interaction patterns to optimize overall worst-case application response time while ensuring total predictability.

We then widen the perspective to the interaction among scratchpad allocation and other multiprocessing aspects affecting application response time. One such dominant aspect is task mapping and scheduling, which largely determines task memory requirement. We present a technique for simultaneous global optimization of scratchpad partitioning and allocation coupled with task mapping and scheduling, which achieves better performance than that resulting from separate optimizations on the two fronts.

The results presented in this work confirm our thesis that explicit consideration of timing predictability in memory optimization does safely and effectively improve worst-case application response time on systems with real-time constraints.

Related Publications

V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad Allocation for Concurrent Embedded Software. In *Proc. ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2008.

V. Suhendra and T. Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores. In *Proc. ACM Design Automation Conference (DAC)*, 2008.

V. Suhendra, C. Raghavan, and T. Mitra. Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures. In *Proc. ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2006.

V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient Detection and Exploitation of Infeasible Paths for Software Timing Analysis. In *Proc. ACM Design Automation Conference (DAC)*, 2006.

V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 2005.

T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting Branch Constraints without Explicit Path Enumeration. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2005.

List of Tables

4.1 Benchmark statistics 48

4.2 Comparison of observed WCET, WCET estimation with and without infeasibility information 49

4.3 Efficiency of our WCET calculation method 51

5.1 Design choices for shared cache 57

5.2 Benchmarks comprising the task sets 62

6.1 Benchmark characteristics 80

6.2 Running time of allocation methods for scratchpad = 10% of data memory 84

7.1 Code size and WCET of tasks in the PapaBench application 125

7.2 Code size and WCET of tasks in the DEBIE application 133

8.1 Benchmark characteristics 161

8.2 Best-case and worst-case algorithm runtimes for the benchmarks 165

List of Figures

2.1 Way-based and set-based cache partitioning 13

2.2 Scratchpad memory 14

3.1 Classification of scratchpad allocation techniques 25

4.1 An example program and its control flow graph (CFG) 41

5.1 Different locking and partitioning schemes for the shared L2 cache . . . 59

5.2 Effects of shared caching schemes *SN*, *DT*, *SC*, and *DC* on task sets with various characteristics 64

6.1 Non-constant WCET reduction due to variable allocation 74

6.2 Pruning in the branch-and-bound search tree 77

6.3 Original and reduced WCET after scratchpad allocation by *ILP*, *greedy (Grd)*, and *branch-and-bound (BnB)* for various benchmarks and scratchpad sizes 82

6.4 Original and reduced WCET after *ILP*, *greedy (Grd)*, *branch-and-bound (BnB)*, and *ACET-based (Avg)* scratchpad allocation for the `fresnel` benchmark 83

7.1 Message Sequence Chart model of the adapted UAV control application 87

7.2 A sample MSC extracted from the UAV control application case study . 88

7.3 Naive memory allocation strategies for the model in Figure 7.2 89

7.4 Choices of scratchpad overlay schemes for the model in Figure 7.2: (a) safe, (b) unsafe, and (c) optimal 90

7.5 Workflow of WCRT-optimizing scratchpad allocation 91

7.6 A simple MSC running on multiple PEs with scratchpad memories . . . 92

7.7 Task lifetimes before and after allocation, and the corresponding interference graphs 99

7.8 Motivation for non-increasing task interference after allocation 105

7.9 Four considered allocation schemes with varying sophistication 107

7.10 Welsh-Powell algorithm for graph coloring 116

7.11 Mechanism of slack insertion for interference elimination — (a) task lifetimes without introducing slack, and (b) the corresponding lifetimes after introducing slack 120

7.12 WCRT of the benchmark application after allocation by *Profile-based Knapsack (PK)*, *Interference Clustering (IC)*, *Graph Coloring (GC)*, and *Critical Path Interference Reduction (CR)*, along with algorithm runtime 124

7.13 Message Sequence Graph of the PapaBench application 127

7.14 WCRT of the complete PapaBench application after allocation by *Profile-based Knapsack (PK)*, *Interference Clustering (IC)*, *Graph Coloring (GC)*, and *Critical Path Interference Reduction (CR)*, along with algorithm runtime 129

7.15 Message Sequence Graph of the DEBIE application 132

7.16 WCRT of the DEBIE application after allocation by *Profile-based Knapsack (PK)*, *Interference Clustering (IC)*, *Graph Coloring (GC)*, and *Critical Path Interference Reduction (CR)*, along with algorithm runtime . . 134

8.1 Embedded single-chip multiprocessor with virtually shared scratchpad memory 141

8.2 Task graph of LAME MP3 encoder 144

8.3 Optimal pipelined schedule for the task graph in Figure 8.2 without considering data allocation 144

8.4 Optimal pipelined schedule for the task graph in Figure 8.2 through integrated task scheduling, scratchpad partitioning and data allocation . . 146

8.5 An example task graph 147

8.6 An optimal non-pipelined schedule for the task graph in Figure 8.5 on four processors 151

8.7 An optimal pipelined schedule for the task graph in Figure 8.5 with (a) single-instance execution view, and (b) steady-state execution view . . . 152

8.8 Initiation interval (II) for the different benchmarks with *EQ*, *PF*, and *CF* strategies given varying on-chip scratchpad budgets on a 2-processor configuration 163

8.9 Improvement in initiation interval (II) due to *PF* and *CF* over *EQ* for benchmark `lame` 164

Chapter 1

Introduction

Safety first.

And yet—

quality, quality, quality.

1.1 Motivation

The omnipresence of computers in this era owes much to the existence of *embedded systems* — specific-purpose applications running on customized, typically compact devices varying in size and complexity from cell phones to automated aircraft controllers. Competitive manufacturers are concerned about tuning the execution platform to maximize the system performance, a term that covers many facets at once: speed, accuracy, energy requirement, and other aspects that define the level of customer satisfaction. Certainly, the optimization effort needs only be catered to the single target application. Nevertheless, with the advanced features ready for exploitation given present-day technology, this alone is a non-trivial matter, and often involves thorough modeling, analysis, and/or simulation of the program.

1.1.1 Real-Time Systems

Performance measure in terms of execution speed is closely related to the concept of *real-time (or timing) constraints*. These are expectations of how much time an application may take to respond to a request for action. They form a part of the specifications of real-time systems, whose functioning is considered correct only if tasks are accomplished within the designated deadlines. For instance, cell phone users expect to see the characters they type on the keypad appear on the screen "instantly", which, given human perception limits, may translate to microseconds of system time. On a more serious note, a car anti-lock braking system (ABS) has to react within a short time to prevent the wheel from locking once the symptoms are detected, so that the driver does not lose steering control over the vehicle under heavy braking.

The cell phone example describes a *soft real-time* system, where exceeding the promised response time amounts to poor quality but does not cause system failure; while the ABS is a *hard real-time* system, where missing the deadline means the system has failed to accomplish the mission. For both types of systems, timing constraints are an important part of the system specification, and it is mandatory to verify those properties by bounding the application response time in the *worst* possible scenario. In other words, the execution time of a real-time system should be *predictable* in all situations – it is guaranteed to be within the stipulated deadlines under any circumstances.

An application generally consists of one or more *processes*, logical units each with a designated functionality that together achieve the application objective. A process, in turn, consists of one or more *tasks*, complete implementations of a subset of the objective. It is sometimes the case that the various tasks in the application have differing deadlines to meet in order for the application deadline to be met in the whole. For example, for the ABS to prevent the wheel from locking in time, the detection of symptoms should be conducted with sufficiently tight period, the signals must be relayed within sufficiently

short interval, and the actuation of the anti-lock mechanism must be sufficiently prompt. As tasks may share dependencies and also resources, it is vital to schedule their execution in a way that enables them to meet their respective deadlines. The analysis that verifies whether a real-time system under development satisfies this requirement is the *schedulability analysis*.

Obviously, the schedulability analysis is primarily concerned with the *worst-case response time (WCRT)* of tasks, that is, the maximum end-to-end delay from the point of dispatch until the task is completed. This delay should account for the time needed for all computations, including the time to access system resources such as memory and I/O devices, and possible contention with other tasks in a concurrent environment. If it is not feasible for the application to meet the timing constraints given the required task response times, or if it costs too much system resource for it to be feasible, then some optimizations are in order. Optimizations can be employed at many levels and in many different forms; however, one important rule to observe in the real-time context is that the optimization effort should be *analyzable* in the interest of schedulability analysis, so that a safe timing guarantee can still be produced.

1.1.2 Memory Optimization

The performance gap between memory technology and processor technology affects all computer systems even today. This is also true for embedded systems. The task execution time is typically dominated by the time needed to access the memory, termed *memory access latency*. As such, memory remains the major bottleneck in system performance, and consequently, memory optimization is one of the most important classes of optimization for embedded systems. While this thesis focuses on the aspect of execution speed, another reason for the significance of memory optimization is the fact that conventional memory systems typically make up 25%–45% of the power consump-

tion as well as the chip area in an embedded processor [15]– which are the two other important measures for the quality of a real-time embedded software.

The memory system is organized in a hierarchy, where the lower level is faster, smaller, and resides closer to the processing unit than the level above it. The lowest levels usually reside on the same chip as the processor ("on-chip").

Traditionally, on-chip memories are configured as *caches*. At any time, caches keep a subset of the memory blocks in the program address space that are stored in full in the main memory. A requested memory block is first sought in the caches, starting from the lowest level, by comparing its address to the cache tags. If it is not found in the cache, it will be loaded from the main memory. At the same time, a copy of the block is kept in the cache. This block will then remain accessible from the cache for reuse until its occupied space is needed by later blocks.

The procedure for loading and replacement of cache contents is managed dynamically by hardware, conveniently abstracted from the point of view of the programmer and/or compiler. However, this abstraction introduces complications in accurately determining the execution time. The timing analysis has to model the workings of the cache and predict the latency reduction for memory requests that can be fulfilled from the cache. Factoring in external influences such as bus contention or preemption by other processes, the analysis can get extremely complex. In order to provide reliable timing guarantee, one solution is to bring down the extent of abstraction and impose software control over the cache operation, thus making the access behaviour more predictable at the cost of sub-optimal cache utilization. Popular approaches in this direction are *cache locking* [106, 22], which fixes the subset of memory blocks to be cached, and *cache partitioning* [65, 120], which eliminates contention for cache space among multiple tasks.

Recent years have seen the surge of popularity of *scratchpad memory*, a design alternative for on-chip memory [16]. In contrast to caches, which act as a "copy", the scratch-

pad memory is configured to occupy a distinct subset of the memory address space visible to the processor, with the rest of the space occupied by main memory (Figure 2.2). This partitioning is typically decided by the compiler, and the content is fully under software control. Memory accesses in a scratchpad-based system are thus completely predictable. This feature has been demonstrated to lead to tighter timing estimates [139] and hence especially suited for real-time applications.

Scratchpad memory also consumes less area and energy compared to caches, because it is accessed only for the pre-determined address range, voiding the need for a dedicated comparison unit at each access. Empirical evaluation [15] has shown that scratchpad usage can offer an average of 34% area reduction and up to 82% energy saving compared to cache usage, making it attractive for embedded applications in general. The down side is that the utilization of scratchpad memory requires additional programming effort for memory blocks allocation.

Multiprocessing Impact In a multiprocessing environment, which is often the case in today's computer systems, lower levels of the memory hierarchy are typically shared among the multiple processing cores. A widely encountered memory architecture is a two-level cache system consisting of privately accessible Level-1 (L1) caches close to each core, and a shared Level-2 (L2) cache placed between the L1 cache and the main memory. Inevitably, resource sharing gives rise to issues such as contention and interference among concurrently executing processes, which leads to higher timing unpredictability. Modeling all of these aspects in addition to memory optimization effects in a complete timing analysis framework is a tremendous task that has not been fully resolved to date. Conventional multiprocessor systems have been relying on simulation to measure computation speed [73]. This method is obviously not strict enough to give performance guarantees in hard real-time systems.

Memory optimization is affected by inter-processor interactions as well, as processors may share on-chip memory and communication channels in various ways, introducing additional delays that need to be factored into the timing analysis. In addition, task division among processors affects overall utilization of memory and other system resources, which in turn affects the effectiveness of memory optimization methods. These should certainly be factored into our optimization effort when targeted at such platforms.

Worst-case Performance Optimization Most optimization efforts have been focused on improving the application performance in the most-encountered scenarios (*average case*), as they are typically taken as the measure of service quality. It is also the case in the field of memory optimization, either for cache-based [106, 132, 111, 120] or scratchpad-based [8, 14, 100, 101] systems. For real-time systems, however, it is often more important to improve the *worst-case* performance, on which the feasibility of the system depends.

While the average-case and worst-case performance may be closely related, a memory management decision that is optimal for the average case may not necessarily be optimal for the worst case. The issue lies in the fact that average-case guided optimizations rely on the *profiling* of the application execution, which collects the information along the execution path triggered by the most encountered input set. The main concern in this context is indeed more focused on the problem of discovering such input sets.

However, the path discovered via profiling is only a single path among all possible execution paths. As the "longest" of these paths defines the worst-case performance of the application, a straightforward extension to worst-case optimization is to simply profile this longest path (deduced from path analysis) and perform the same procedure as in the average-case optimization. However, once the optimization is applied, we can expect a reduction in execution time along this path, which may now render it shorter than some other execution path. We say that the worst-case execution path has *shifted*.

Thus, the effort we have spent on the former worst-case path only achieves a *local* optimum in application performance. To aim for the global optimum, the method needs to factor in the shifting of the worst-case path.

Our work tackles the challenge of performing memory optimizations targeted at improving the worst case application performance, in order to meet real-time constraints of embedded software in both uniprocessing and multiprocessing environments. In every optimization effort, the timing predictability of the system is maintained in order to retain safe timing guarantees.

1.2 Thesis Statement

The thesis of this research is that real-time concerns affect the effectiveness of memory hierarchy optimization in embedded real-time systems, and therefore need to be factored in to achieve optimal memory utilization. Conversely, it is important to develop optimization methods that do not compromise the timing predictability of the system, in order to safely meet the system requirements.

In this thesis, we discuss the following connected facets of memory optimization for real-time embedded software.

- How can we accurately bound the effects of memory hierarchy utilization on application response time?
- From the other end of the perspective, how may we guide our optimization effort based on the quantification of its effect on the worst-case performance?
- In situations where it is necessary, what is the point of balance where optimality should be compromised to respect timing predictability without leading to significant performance degradation?

- How can we use the knowledge of application characteristics and platform features in making design decisions for the memory hierarchy management?
- What other system features affect task response times and/or the effectiveness of memory optimizations? How can we model the interaction among them?

1.3 Thesis Organization

We have introduced the motivation and thesis of our research in this chapter. Following this, Chapter 2 will first lay the foundation for discussion by presenting the basics of cache and scratchpad memory, along with an introduction to worst-case execution time (WCET) analysis and integer linear programming. The last concept provides a precise way to formulate optimization problems in our framework.

Chapter 3 further surveys state-of-the-art optimization techniques related to memory hierarchy management and real-time constraints. We look at cache-based techniques, scratchpad-based techniques, as well as the integration of both. Broadening our perspective, we proceed to survey multiprocessor memory management and design space exploration. The chapter concludes with a brief review of worst-case performance enhancement techniques in aspects other than memory optimization, which are still relevant due to their interaction on the execution platform.

As timing analysis is an issue that is inseparable from predictable memory optimizations, Chapter 4 details the key points and techniques for analysing the WCET of tasks. We present an efficient WCET analysis method with enhanced accuracy that, when integrated into our memory allocation framework, enables us to obtain immediate feedback and finetune optimization decisions.

We open our discussion on predictable memory optimizations in Chapter 5 by addressing the problem of utilizing shared caches in a manner that preserves timing predictabil-

ity. This study complements the existing researches on predictable cache management that have been largely focused on private caches.

We then proceed to describe optimization methods targeted at scratchpad memory as a better choice for real-time systems, and dedicate two chapters for the treatment of the issue. We start by presenting scratchpad allocation aimed at minimizing the WCET of a single task or sequential application in Chapter 6. After that, we proceed to discuss scratchpad allocation for concurrent multitasking applications in Chapter 7.

Following these, we extend our view and look at how scratchpad allocation may interact with other multiprocessing aspects that also influence task response times. One such dominant aspect is task mapping/scheduling, which largely determines task memory requirement. Chapter 8 thus studies scratchpad memory partitioning and allocation coupled with task mapping and scheduling on multiprocessor system-on-chips (MPSoCs).

Finally, we conclude our thesis with a summary of contributions and examine possible future directions in Chapter 9.

Chapter 2

Background

In this chapter, we first look into the details of caches and scratchpad memories as the basis for discussion on memory optimization techniques in later chapters. The operating principles and features relevant to real-time requirements are discussed. We then present an intuitive overview on the concept of worst-case execution time and its determination, as a prelude to a more detailed treatment in Chapter 4. Finally, we give an introduction to the concept of integer linear programming, which we utilize significantly in the formulation of the optimization problem.

2.1 Cache

2.1.1 Cache Mechanism

A cache is a fast on-chip memory that stores copies of data from off-chip main memory for faster access [49]. The small physical size of caches allows them to be implemented from the faster, more expensive SRAM (Static Random Access Memory) technology, as compared to DRAM (Dynamic Random Access Memory) used to build the main

memory. In addition, they are positioned close to the processor, so that bit signals need to travel only a short distance. Most of all, a cache is effective because memory access patterns in typical applications exhibit locality of reference. In a loop, for example, data are very likely to be used multiple times. If these data still remain in the cache after the first fetch, the next request to them can be fulfilled from the cache without the need for another fetch, thus saving the access time. This is the *temporal locality* property.

To a lesser extent, caches also benefit from *spatial locality*: nearby data are fetched along with the current requested data, as it is anticipated that they will be required soon in the future. This type of locality is especially applicable to instruction caches, as the sequence by which program code blocks are stored in memory largely corresponds to the sequence by which they are executed.

The unit of transfer between different levels of cache hierarchy is called *block or line*. The size of a cache line commonly ranges from 8 to 512 bytes. The cache is divided into a number of *sets*. In a cache of N sets, a memory block of address Blk can be mapped to only one cache set given by $\lfloor \frac{Blk}{N} \rfloor$. If a cache set ("row") contains S cache lines, then we say the cache has *associativity* S, or the cache has S ways ("columns"). A block mapped to a set can occupy any column in the set. The total size of the cache is thus $N \times S$ multiplied by the cache line size.

Each datum in the cache has a *tag* to identify its address in main memory. Upon a processor request for a datum at a certain address, the cache is searched first. If a valid tag in the cache matches the requested address, the access is a *cache hit* and the corresponding datum is delivered to the processor. Otherwise, it is a *cache miss* and the datum is sought in the next memory level. The access latency of a block of data that is found at level $\mathcal{L}$ in the cache hierarchy thus includes the time taken to search the cache levels up to $\mathcal{L}$ in addition to the time needed to bring the block from level $\mathcal{L}$ all the way to the processor. In the event of a cache miss where the datum is brought in from the

main memory, a copy of the datum is also loaded into the cache, possibly replacing an old block that maps to the same cache set.

2.1.2 Cache Locking

Cache locking is a mechanism that loads selected contents into the cache and prevents them from being replaced during runtime. This mechanism is enabled in several commercial processors, for example IBM PowerPC 440 [53], Intel-960 [54], ARM 940 [10], and Freescale's e300 [57]. If the entire cache is locked, then accesses to memory blocks locked in the cache are always hits (except for the obligatory load or cold misses), whereas accesses to unlocked memory blocks are always misses. That is, knowing the cache contents allows the timing analysis to account for the exact latency taken by each memory access. In practice, designers may provide the options of locking the entire cache or locking a set of individual ways within the cache ("way locking") [57], leaving the remaining unlocked ways available for normal cache operation.

The selected content may remain throughout system run in the *static locking* scheme, or be reloaded at chosen execution points in the *dynamic locking* scheme. Dynamic cache locking views the application or task as consisting of multiple execution *regions*. Regions are typically defined based on natural program division such as loops or procedures, each of which utilizes a distinct set of memory blocks, thus "isolating" the memory reuse. An offline analysis selects memory blocks to be locked corresponding to each region. As the execution moves from one region to another, the cache content is replaced with blocks from the new region. Instructions are inserted at appropriate program points to load and lock the cache. Certainly, the delay incurred by the reloads has to be factored in the execution time calculation.

2.1.3 Cache Partitioning

Cache partitioning is applied to multitasking (or multiprocessing) systems to eliminate inter-task (inter-processor) interference. Each task (processor) is assigned a portion of the cache, and other tasks (processors) are not allowed to replace the content. Cache analysis can then be applied to each cache partition independently to determine the WCET of the task (processor). Cache partitioning is less restrictive than cache locking, as dynamic behavior is still present within the individual partitions.

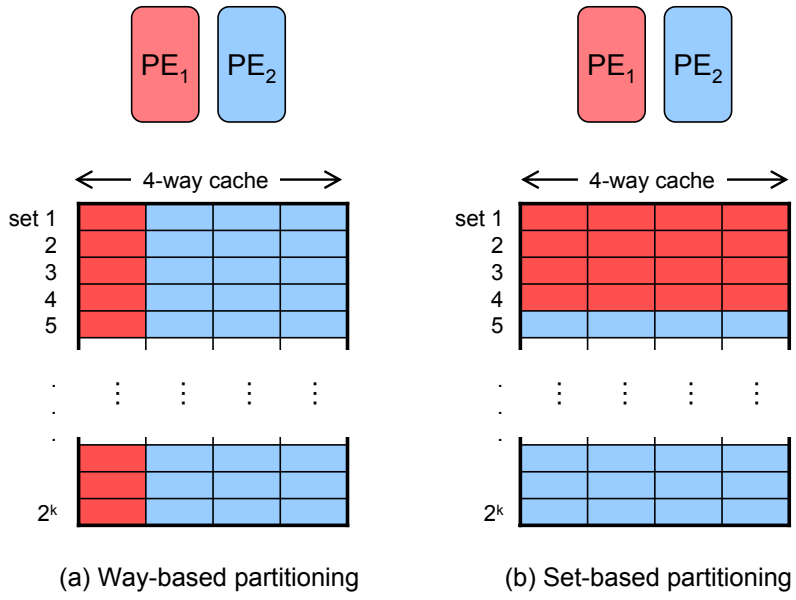


Figure 2.1: Way-based and set-based cache partitioning

There are two schemes in which cache partitioning can be performed. *Way-based* partitioning [27] allocates a number of ways ("columns") to each task (Figure 2.1a). As the number of ways in caches is quite restricted (typically 4 and at most 16), this scheme does not support fine-grained partitioning. In practice, way-based partitioning can be configured so that a task/processor may still read and update cache lines belonging to another task/processor, though it is not allowed to evict them [131]. A more flexible scheme is *set-based* partitioning [66], which allocates a number of sets ("rows") to each task (Figure 2.1b). This partitioning scheme translates the cache index (in hardware) so

that each task addresses a restricted part of the cache. For efficient hardware translation, the number of sets in a partition should be a power of 2.

Molnos et al. [88] compare both partitioning options when applied to compositional multimedia applications, and show experimentally that the greater flexibility of set-based partitioning works well in that particular setting to yield less cache misses compared to way-based partitioning. They also observe that it is technically possible to implement both set- and way-based partitioning in a single configuration, but both implementation overhead will add up, slowing down the cache too much to be practical.

2.2 Scratchpad Memory

Scratchpad memories are small on-chip memories that are mapped into the address space of the processor (Figure 2.2). Whenever the address of a memory access falls within a pre-defined address range, the scratchpad memory is accessed.

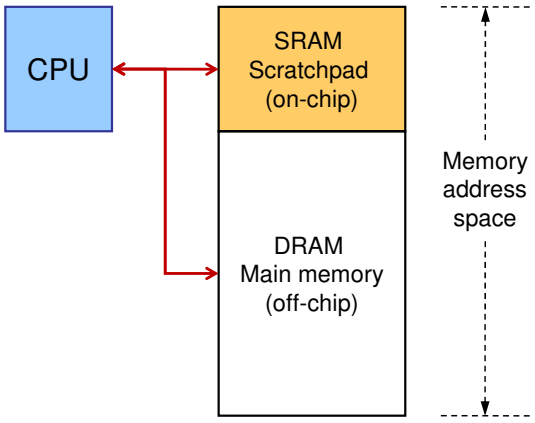


Figure 2.2: Scratchpad memory

Scratchpad memory is available on a wide range of embedded CPUs, including IBM Cell [52], Motorola M-CORE M210 [41], Texas Instruments' TMS-470R1x [126], Intel IXP network processors [54], and others. In general, it can be employed as an alternative

to caches, or on top of caches. Several classes of embedded processors (ARM Cortex-R4 [9], most ARM11 family [10], TigerSHARC ADSP-TS20x [6], Blackfin ADSP-BF53x [5]) have both scratchpad memory and caches built into the chip.

The predictable timing behaviour of scratchpad memory has led to a growth in its utilization for real-time systems. Wehmeyer [139] demonstrates that much tighter WCET estimations can be obtained when employing scratchpad memory instead of caches, leading to better system predictability. Other advantages of scratchpad memory include reduced area and energy consumption compared to caches [16], because it does not need to employ a dedicated comparison unit to check if each access is a hit or miss. However, now the burden of allocating memory objects to scratchpad memory lies with the compiler or programmer.

Scratchpad memory can be used to store program code [8, 34, 58, 109, 135], program data [14, 32, 33, 101, 130], or a combination of both [119, 136, 138]. The granularity of allocation unit is also a compiler decision, in contrast to fixed line sizes in caches. In the case of code scratchpad, it is reasonable to allocate in units of basic blocks, whole loops, or whole functions. Data scratchpad space can be allocated to scalar variables with little or no issue, but finer considerations may be needed for large arrays and heap variables whose sizes are unknown at compile time.

The different access patterns to code and to data give rise to different concerns as well. Allocating program code requires additional care to maintain program flow [119], while allocating program data generally calls for specific considerations depending on the type of the data (global, stack, or heap) and the different nature of their access. The allocation schemes are often coupled with supporting techniques such as data partitioning [40], loop and data transformations [64] or memory-aware compilation [86] to make the access pattern more amenable for allocation.

We shall look at scratchpad allocation strategies in more details when we survey the state of the art in Chapter 3.

2.3 Worst-Case Execution Time

The worst-case execution time (WCET) of a program is the upper bound on the time it takes to execute from start to termination, on the given architectural platform, in the intended environment. This notion is meaningful mainly in the context of (1) providing the guarantee that the program output (computation result, event response, and so on) will be available after a certain amount of time, or (2) ensuring reservation of sufficient system resources for the duration of execution.

For a deterministic program with known and manageable input ranges, a reasonably accurate WCET value can easily be determined by actually running the program with all possible inputs and environment parameters, and observing the longest time taken. Such a case is unfortunately extremely rare in real-life applications, for which some methods for estimation thus need to be developed. The methods will also need to take into account the execution platform and environment, which affect the execution time significantly. The resulting WCET estimation is required to be *safe*, so that it does not underestimate the actual time needed to complete the program. Optionally, it is desired to be *tight*, thus giving a good gauge of the actual running time to be expected in the real execution.

WCET estimation is generally achieved via a static analysis method, that is, by examining the executable code produced at compile time. First, the architectural features determine the time taken to execute each instruction, which is usually the basic building unit of the program code. For instructions that perform memory accesses, the time should include the access latency, accounting for the presence of caches or other mem-

ory optimization schemes. The time taken by a sequence of instructions is not a direct summation of this, but rather should be computed considering the way instructions flow in the datapath and processor pipeline. The analysis at this level is referred to as the *micro-architectural modeling* stage of the WCET analysis.

At the next level, sequences of instructions form basic blocks in the logical flow of the program, related by conditional branching, procedure calls, and so on. The *flow analysis* stage handles the analysis at this level, examining all possible paths that may be followed in an execution of the program and calculating the total time taken from the program entry to any program exit. As the running time of each basic block is already determined in the micro-architectural modeling stage, the final *WCET calculation* stage is able to combine the information and report the maximum execution time over all possible execution paths and behaviors at the micro-architectural level.

The dual of this procedure, which seeks to determine the minimum instead of the maximum execution time, is termed the *best-case execution time (BCET) analysis*. The resulting two metrics together determine the execution time window of the program. This information is important when more than one programs interact within an application, as the total *application response time* may vary with various interaction patterns that depend heavily on the execution time windows of each program.

As the notion of WCET is central to our optimization effort, we shall further discuss the pragmatic as well as technical aspects of WCET analysis in Chapter 4.

2.4 Integer Linear Programming

We now give a quick introduction to the concept of linear programming and integer linear programming, which is central to our problem formulation in the majority of the thesis.

Linear programming is a technique for optimization of a linear objective function, subject to linear equality and linear inequality constraints. In practical uses, it determines the way to achieve the best outcome in a given mathematical model, given requirements represented as linear equations.

The most intuitive form of describing a linear programming problem consists of the following three parts.

- A linear function to be maximized, *e.g.*

$$\text{maximize} \quad (aX + bY)$$

  where X and Y are variables to be determined, while a and b are known constant coefficients.

- Problem constraints in linear form, *e.g.*

$$cX + dY \leq e$$

- Variable bounds, *e.g.*

$$X \geq 0$$

This thesis adopts the common convention of using names starting with capital letters for the variables (to be solved), and names starting with small letters for constants.

A minimization problem is a *dual* of the maximization problem, and can always be written into an equivalent problem in the above form. Certain constraints in alternative forms, for example those involving conditionals, can be *linearized* with the help of auxiliary variables as long as they do not contain multiplication of variables, and thus can still be expressed using linear programming.

Illustration As an illustration, let us model a simple knapsack problem. Suppose a store owner sells three types of beverage products: soft drinks, fruit juice, and milk. The soft drinks come in aluminum cans with the gross weight of 650 g per can and earn him the profit of 30 cents per can. The fruit juice is sold in 1.1 kg bottles priced to yield a profit of 45 cents each, while each carton of milk weighs 1.2 kg and earns 55 cents profit. The store owner drives an open-top truck with 500 kg load capacity to transport the beverages from the warehouse to his store. All three types of beverages are in equal demand, and he makes sure to supply a minimum quantity of 100 each with every trip. Given these requirements, the store owner wants to calculate the quantity of each type of beverage he should take in one trip in order to maximize his profit.

This problem can be expressed as a linear program as follows. Let us represent the soft drink quantity, the juice quantity and the milk quantity using the variables X_s, X_j, and X_m respectively. The objective function is the total profit maximization, that is

$$\text{maximize} \quad (30X_s + 45X_j + 55X_m)$$

The problem constraint is that the total weight of all products to be transported should not exceed the load capacity of the truck. (We assume that the open-top truck does not impose a volumetric limit.)

$$0.65X_s + 1.1X_j + 1.2X_m \leq 500$$

The bounds on the variables are provided by the minimum supply requirement:

$$X_s \geq 100; \quad X_j \geq 100; \quad X_m \geq 100$$

Since the quantities of products should be whole numbers, we require that X_s, X_j, and X_m are integer-type variables.

The optimal solution to the above linear program gives $X_s = 408$, $X_j = 100$, and $X_m = 104$, which achieves the objective value of 224.6. The interpretation in the original context is that the store owner will make the maximum profit of \$224.6 by taking 408 cans of soft drink, 100 bottles of fruit juice, and 104 cartons of milk.

We can see how the memory allocation or partitioning problem is closely related to the knapsack problem, as we can view the memory blocks as "items" to be placed in the fast-access memory, with the "gain" being the expected reduction in latency and the "cost" being the area they occupy in the limited memory space. Certainly, we shall need to extend this basic formulation to handle other concerns in the worst-case performance optimization.

If the unknown variables are all required to be integers as in the above example, then the problem is an integer linear programming (ILP) problem. While linear programming can be solved efficiently in the worst case, ILP problems are generally NP-hard. 0-1 (binary) integer programming is the special case of integer programming where the value of variables are required to be 0 or 1, and is also classified as NP-hard. Solving integer linear programs is a whole field of research by itself, where advanced algorithms have been invented including cutting-plane method, branch and bound, branch and cut, and others. The solution process of the ILP formulations in our problem model is an orthogonal issue and is thus not discussed in detail here; this aspect of our framework is delegated to an external ILP solver, ILOG CPLEX [29].

Chapter 3

Literature Review

This chapter presents an overview of existing research on memory optimization techniques as well as related worst-case performance enhancements targeted at real-time systems.

3.1 Cache Analysis

Caches have been the traditional choice for memory optimization in high-performance computing systems. Cache management is handled by hardware, transparent to the software. This transparency, while desirable to ease the programming effort, leads to unpredictable timing behavior for real-time software. Worst-case execution time (WCET) analysis needs to know whether each memory access is a hit or miss in the cache, so that the appropriate latency corresponding to each case can be accounted for.

A lot of research efforts have been invested in modeling dynamic cache behavior to be factored in WCET calculation. In the context of instruction caches, a particularly popular technique is *abstract interpretation* [2, 127] which introduces the concept of *abstract*

cache states to completely represent possible cache contents at a given program point, enabling subsequent classification of memory accesses into *always hit*, *always miss*, *persistent/first miss*, and *unclassified*. The latency corresponding to each of these situations can then be incorporated in the WCET calculation. Other proposed analysis methods in the literature include data-flow analysis [91], integer linear programming [80] and symbolic execution [84]. In contrast to exact classification of memory accesses, another class of approach focuses on predicting miss ratio for a program fragment, utilizing concepts such as reuse vectors within loop nests [69, 42], conflict misses for a subset of array references [125], and Presburger formulas [26].

The analysis of data caches is further complicated by the possibility of array or pointer aliasing and dynamic allocation. White et al. [142] perform static simulation to categorize array accesses that can be computed at compile time. Xue and Vera [143] utilize abstract call inlining, memory access vectors and parametric reuse analysis to quantify reuse and interferences within and across loop nests, then use statistical sampling techniques to predict the miss ratio from the mathematical formulation.

All these methods work on private caches; we have not known of an analysis method that model the dynamic behavior of a shared cache. The intricate dimensions of the problem lead us to believe that such an analysis will be prohibitively complex to attempt in full accuracy. As we will see later in this chapter, it is then reasonable to curb the dynamic nature of the cache via limited software control.

Cache-Related Preempted Delay Tasks in a multitasking system rarely have simple, constant execution times estimable from their computation needs, due to the various possible interaction scenarios that they may get involved in. Even when the cache behavior can be predicted for a task in isolation, the estimation may turn invalid in the face of preemptions. Cache contents belonging to the preempted task may be evicted during

the run of the preempting task, leading to additional cache misses when the preempted task resumes. This effect is known as *cache-related preempted delay (CRPD)*.

CPRD analysis has been widely researched. A set-based analysis in [72] investigates cache blocks used by the preempted task before and after preemption. Another approach in [129] applies implicit path analysis on the preempting task. Negi et al. [93] perform program path analysis on both the preempted and the preempting tasks to estimate possible states of the entire cache, symbolically represented as a Binary Decision Diagram, at each possible preemption point. This approach is later extended by Staschulat and Ernst [117] for multiple process activations and preemption scenarios.

3.2 Software-Controlled Caching

Due to the complexity of accurate cache analysis described above, most existing analyses handle unpredictable program regions by relying on certain restrictions to eliminate or reduce undeterminism before applying analysis. The two main approaches are *cache locking* and *cache partitioning*, whose main features have been described in Chapter 2.

Cache Locking Static cache locking algorithms that minimize system utilization or inter-processor interference are presented by Puaut and Decotigny in [106]. Campoy et al. [22] also aim at system utilization minimization when employing genetic algorithms for content selection of statically locked caches. A dynamic cache locking algorithm is proposed in [11], targeted at improving the worst-case performance. Further, Puaut in [105] presents a comprehensive study of worst- and average-case performances of static locking caches in multitasking hard real-time systems, as compared to the performance of unlocked caches. The report identifies the application-dependent threshold at which the performance loss in favor of predictability is acceptable.

Cache Partitioning Hardware-based cache partitioning schemes have been presented in the literature; Kirk [66] presents set-based partitioning while Chiou [27] proposes associativity-based partitioning. Sasinowski [111] proposes an optimal cache partitioning scheme that minimizes task utilization via dynamic programming approach. Suh [120] on the other hand proposes a dynamic cache partitioning technique to minimize cache miss rate, while the technique by Kim [65] aims to ensure fairness among multiple tasks. The partitioning technique in [110] has the feature of allowing prioritizing of critical tasks. Meanwhile, Mueller [89] focuses on compiler transformations to accommodate cache partitioning in preemptive real-time systems. The compiler support is needed to transform non-linear control flow to accomodate instruction cache partitioning, and to transform codes executing data references in the case of data cache partitioning. The impact of these transformations on execution time is also discussed.

Combined Approach Puaut in [106] considers static cache locking in a multitasking environment. The proposed technique considers all tasks at once in selecting the contents to lock into the cache, hence partitioning is also formed in the process. Vera [132, 133] combines cache partitioning and dynamic cache locking with static cache analysis to provide a safe estimate of the worst-case system performance in the presence of data cache. The cache is first partitioned among tasks. In each task, program regions that are difficult to analyze are selected for locking. The cache contents to be locked are selected by a greedy heuristic. The remaining program regions are left to use the cache dynamically, and cache analysis determines the worst-case timing for these regions. Only uniform partitioning is investigated in the paper. For the dynamic scheme to be feasible, partitioning needs to be done preceding the content selection. The dynamic policy thus allows less partitioning flexibility, but potentially more improvement from space overlay, compared to the static policy.

3.3 Scratchpad Allocation

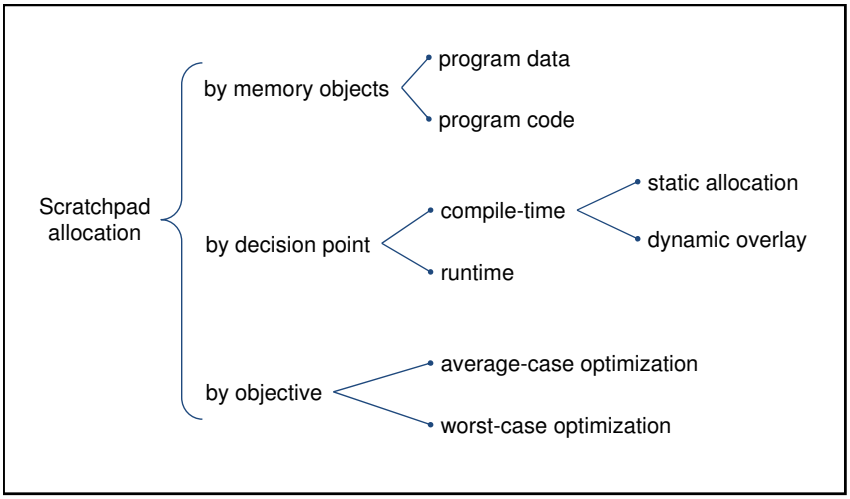


Figure 3.1: Classification of scratchpad allocation techniques

Existing scratchpad allocation schemes in the literature can be majorly classified into *compile-time* and *runtime* techniques (Figure 3.1), differing in the point of time when the allocation decision is made.

Compile-time Allocation *Compile-time* scratchpad allocation techniques perform offline analysis of the application program and select beneficial memory content to be placed in the scratchpad. This approach incurs no computation overhead during the execution of the application itself. The methods in this category can be further classified into *static allocation* and *dynamic overlay*.

Static allocation loads selected memory blocks into the scratchpad during system initialization, and does not change the content until the completion of the application. Techniques for scratchpad content selection include dynamic programming [8] and 0-1 ILP [119, 138]. Panda et al. [101] view the allocation problem as a partitioning of data into the different levels of the memory hierarchy. They present a clustering-based partitioning algorithm that takes into account the lifetimes and potential access conflicts

among program data, as well as possibilities of context switching in a multiprocessing environment. The static allocation scheme is reasonably efficient to implement, even though its effectiveness may be limited to applications with relatively small memory requirement compared to available memory space.

Scratchpad allocation with dynamic overlay, on the other hand, may reload the scratchpad with new contents when the execution reaches designated program points. This approach requires a way to reason about the contents of the scratchpad memory over time. Udayakumaran and Barua [130] introduce the concept of timestamps to mark the program points of interest. A cost model determines the cost of memory transfers at those points, and a greedy compile-time heuristic selects transfers that maximize the overall runtime benefit. Verma et al. [136] perform liveness analysis to determine the live range of each memory object to be placed in the scratchpad. This information is then used to construct an ILP formulation with the objective of maximizing energy savings. Steinke et al. [118] also formulate the problem as an ILP optimization by modeling the cost of copying memory objects at selected program points. Kandemir et al. [61], on the other hand, focus on data reuse factor and use Presburger formula to determine the maximal set of loop iterations that reuse the elements residing in the scratchpad, in order to minimize data transfer between on-chip and off-chip memory. For these methods, even though the reloading of scratchpad contents is executed at runtime, the entire analysis to select memory blocks and reloading points is performed at compile time, thus incurring no runtime delay for computation of the gain functions.

Runtime Allocation In contrast to compile-time allocation described above, *runtime scratchpad allocation* techniques decide on the scratchpad memory content when the application is running. There can be several reasons to opt for this approach. One reason is the situation when it is not possible to perform selection at compile time, because the size of the scratchpad or the program memory requirement is unknown at compile time.

Such a situation may arise when the embedded program is not burned into the system at the time of manufacture, but is rather downloaded during deployment via the network or portable media [95]. Another reason is the situation when the memory requirement of the specific application varies widely across its input. In this case, a beneficial allocation decision can be better determined after analysing the execution trace or history [35, 109]. Egger et al. [35], in particular, make use of the page fault exception mechanism of the Memory Management Unit (MMU) to track page accesses and copy frequently executed code sections into the scratchpad.

Runtime allocation methods inevitably add the cost of performing content selection to the application runtime, even if it may be offset by the gain due to allocation. Nevertheless, most methods are able to alleviate this overhead by pre-computing part of the analysis that does not depend on runtime information. Nguyen et al. [95] first identify potentially beneficial memory blocks at compile time when the scratchpad size is still unknown; then perform the actual selection once the program loader discovers the scratchpad size at startup. In a similar approach, Dominguez et al. [33] allow allocation of heap objects whose sizes are unknown at compile time. Their method performs compile-time analysis to determine potential allocation sites, then reserves fixed-size portions in the scratchpad to be occupied by subsets of these objects once they are created, selected during runtime via a cost-model driven heuristic.

Worst-case Optimization Most of the works discussed above aim to minimize the *average-case* execution time or energy consumption through scratchpad allocation. The growing utilization of scratchpad memory in real-time systems, where the concern is instead the *worst-case* execution time (WCET) of the application, makes it important to consider scratchpad allocation to optimize the worst-case performance (see Figure 3.1). A work we will discuss in this thesis [121] presents optimal and heuristic WCET-centric static allocation methods for program data, while Deverge and Puaut [32] consider dy-

namic allocation for global and stack data. The main concern in WCET-centric allocation is that the worst-case execution path of the program may change as scratchpad allocation changes. These methods account for this change in WCET path by performing iterative analysis along with incremental fine-tuning of the scratchpad allocation.

Scratchpad Allocation for Multiprocessing Systems As we move on to consider applications running on multiprocessor systems, concurrency and sharing among the multiple tasks or processing elements (PEs) become important factors. Early static allocation methods [101] simply partition the scratchpad to the tasks according to gain functions extended from allocation strategy for single-process applications. Verma et al. [134] present a set of scratchpad sharing strategies among the processes for energy consumption minimization. Processes may take up disjoint space in the scratchpad so that no restoration is required upon context switch, or they may share the whole scratchpad whose content will be refreshed when a process is activated. A hybrid between the two is also considered.

Another class of approach focuses on the *mapping of codes/data* to the private scratchpad memories of the PEs [59], or among memory units in a heterogeneous system [14] so as to maximize the benefit from the scratchpad allocation. Others propose *runtime customization of scratchpad sharing* among tasks or PEs to adapt to the changing memory requirement. Ozturk et al. [99] first perform automated compiler analysis to capture memory requirements and reuse characteristics of the tasks at loop granularity, then use this information to dynamically partition the available scratchpad space across tasks at runtime. Kandemir et al. [62] also present a strategy to dynamically reconfigure scratchpad sharing and allocation among PEs to adapt to runtime variations in data storage demand and interprocessor sharing patterns.

3.4 Integrated Cache / Scratchpad Utilization

Verma et al. [135] report in their experiments that a scratchpad memory allocation technique that assumes the absence of cache may not perform well in an architecture that includes a cache on top of scratchpad memory. Several researches have considered both memory types for data allocation, although they do not consider real-time concerns in the approach. Avissar et al. [13, 14] formulate the partitioning of data between scratchpad memory and a data cache as an 0-1 ILP formulation. The strategy is to allocate the most frequently accessed variables in scratchpad memory. Their later work extends this approach to heap memory [33]. Panda et al. [101] focus on the use of scratchpad memory to minimize data cache conflicts instead. For this purpose, scalars and constants are chosen to be mapped into scratchpad memory, so that contiguous blocks of array elements can be loaded into the cache with minimum thrashing. Kandemir et al. [64] apply explicit data transfers between on-chip and off-chip memory in order to accomodate arrays that are larger than the on-chip memory capacity. The scalar variables are kept in off-chip memory to be accessed through data cache. Another work of their group [63] proposes the concept of virtually shared scratchpad memory (VS-SPM) by allowing remote access to each processor's private scratchpad. Sjodin et al. [113] consider the entire address space and allocate variables to appropriate memory types so that the size of the corresponding pointers, and hence the code size, can be reduced. Flexible scratchpad design for multiprocessor System-on-Chips (MPSoCs) is also investigated in [97] and [98].

3.5 Memory Hierarchy Design Exploration

A broader class of research considers exploration of memory hierarchy design space in conjunction with other multiprocessing aspects. Kandemir and Dutt [60] give an

excellent reference to memory system design in the context of chip multiprocessors. Issenin et al. [55] present a multiprocessor data reuse analysis that explores a range of customized memory hierarchy organizations with different size and energy profiles.

Another approach addresses the problem of determining the best cache memory sizes for optimal cost or performance [92]. Lee et al. [74] explore cache design for large high performance multiprocessors, modeling multiprocessing aspects that include cache coherence issues, data prefetching, parallel execution, etc. Further, [124] designs a complete tile-based cache architecture called WaveCache to support the low-complexity/high-performance WaveScalar dataflow instruction set and execution model. WaveCache handles the mapping of instructions onto processing elements as well as replacement of expired instructions during program execution.

From the software perspective, researchers have also investigated ways to maximize the benefit of a shared cache. Kim et al. [65] study fairness in cache sharing among program threads, while Chang and Sohi [24] introduce a cooperative cache management scheme that places data on either the shared cache or one of the private caches based on whether they are globally or locally active. Conversely, cache usage can be taken into account during the assigning and scheduling of tasks on the processing cores. Anderson et al. [7] propose a cache-aware multi-core scheduling scheme for real-time applications, while Squillante and Lazowska [114] consider processor-cache affinity when deciding on task allocation to the multiple cores. Meanwhile, Li and Wolf [78] consider the effects of cache on schedulability during allocation and scheduling decisions. This technique employs cache partitioning and reservation to reduce unpredictability in timing estimation, which is performed based on a distribution model.

3.6 Worst-Case Optimizations in Other Fields

Compiler techniques to reduce the worst-case execution time of a program have started to receive attention over the last few years. Lee et al. [75] develop a code generation method for dual instruction set ARM processors to simultaneously reduce the WCET and the code size. The full ARM instruction set is employed along the WCET path to achieve faster execution, while reduced Thumb instructions are used along non-critical paths to reduce code size. Meanwhile, Yu and Mitra [145] perform WCET-guided selection of application-specific instruction set extensions.

Bodin and Puaut [104] design a customized static branch prediction scheme for reducing a program's WCET. This work employs a greedy heuristic to design the branch prediction scheme — all branches appearing in the current WCET path are predicted based on their outcomes in the WCET path. Zhao et al. [148] use a greedy heuristic for code positioning that places the basic blocks along WCET paths in contiguous positions whenever possible. Their later work [147] attempts WCET reduction by forming superblocks along the WCET path.

Chapter 4

Worst-Case Execution Time Analysis

The *worst-case execution time (WCET) analysis* is inseparable from our effort to improve the worst-case performance of an application. The analysis computes an upper bound on the execution time of a task, on a particular processor, for all possible inputs. This gives a safe timing guarantee as well as a measure for the effectiveness of the optimization technique. It is also important that the analysis can be performed efficiently, enabling iterative feedback process in fine-tuning the optimization decision.

In this chapter, we describe the process of static WCET analysis and the issues involved. We then present an efficient WCET estimation technique that improves on the tightness of existing methods by detecting and exploiting infeasible path information in the program control flow graph without resorting to exhaustive path enumeration.

4.1 Overview

Static WCET analysis of a program typically consists of three phases: *flow analysis* to identify loop bounds and infeasible flows through the program; *micro-architectural*

modeling to determine the effects of pipeline, cache, branch prediction etc. on the execution time; and finally *calculation* to bound the execution time given the results of the two preceding stages. The results of program flow analysis and micro-architectural modeling are typically combined via a *separated* approach. In this approach, the micro-architectural modeling is used to get an estimate of the WCET of each basic block of the program. These basic block WCET estimates are the combined with flow analysis results to produce the WCET estimate of the whole program.

The writeup in [107] gives a comprehensive summary of techniques employed in prominent WCET analysis tools and research groups up to the year 2000. Some of these techniques have well persisted until today. In the following, we discuss the development in each stage of the WCET analysis.

4.1.1 Flow Analysis

The program flow analysis derives constraints on the possible execution paths of a program, such as iteration bounds for loops, dependencies among conditional branches, and so on. These constraints can originate, for example, from the programmer's knowledge of the program functionality. They are typically provided to the WCET analysis tool through manual annotations. Certain flow informations can be inferred statically from the program, and it is the goal of the flow analysis to extract such information as much as possible in an automated manner.

Most researches on flow analysis have focused on loop bound analysis, since it is absolutely necessary to determine upper bounds on the number of loop iterations in order to produce a WCET estimate [43]. Existing approaches include syntactical loop pattern detection [46, 128], a combination of data flow analysis and value analysis [31], abstract interpretation and program slicing [38].

Another facet of flow analysis deals with identifying infeasible paths. These are paths that can be traced in the control flow graph of the program, but are not feasible in actual execution due to the semantics of the program and/or possible input data values. Removing infeasible paths from consideration allows the WCET analysis to obtain a tighter estimate. Techniques along this direction have employed, among others, branch-and-bound search [3] and extended simulation [84]. Ermedahl and Gustafsson [37] use dataflow analysis to derive and exploit infeasible paths using program semantics; a nice feature of this work is that it also automatically derives minimum and maximum loop bounds in a program. Healy and Whalley [48] detect and exploit branch constraints within the framework of the path-based technique. The key idea here is to compute the effect of any assignment or a branch on other branch outcomes; this is an efficient way of computing many common infeasible path patterns. Lastly, in the context of software model checking, Henzinger et al. [50] abstract data values via a finite number of propositions and eliminate infeasible paths by invoking an external theorem prover.

Our Technique As execution scenarios are closely related to program flow, flow analysis plays an important role in the interaction between WCET analysis and worst-case performance optimization. For the WCET analysis portion of our framework, we develop a lightweight infeasible path detection technique to keep the analysis fast and efficient while still sufficiently accurate for our purpose. For the same objective, loop bounds are supplied manually. The detailed description of this flow analysis procedure will be given following this section.

4.1.2 Micro-Architectural Modeling

The execution time of an instruction is largely determined by the micro-architecture of the underlying platform. Modern processors employ advanced performance enhancing

features such as pipelining, caches, branch prediction, and so on, which induce variation in the instruction execution time. A counter-intuitive timing behavior called *timing anomaly* [85, 141] is also known to exist. This refers to the phenomenon when earlier completion of instruction execution (due to variation in cache hit/miss result, for example) unexpectedly leads to longer execution time of the task as a whole, because of the way instructions move along the processing pipeline. Micro-architectural modeling attempts to capture the timing effects due to these features in order to provide instruction timing information for WCET calculation of the program.

An early attempt to model pipelines for WCET analysis was performed by Zhang et al. [146] for a two-stage pipeline, where much of the complication lies in determining how much of the next instruction to be executed is in the prefetch queue. Burns et al. [21] use colored petri nets to model the timing of speculative and out-of-order instruction execution for processors with multiple execution units and with multi-staged pipelines. The petri net model is then simulated to evaluate the timing of instruction execution.

Colin and Puaut [28] employ static program analysis and branch target buffer modelling to statically bound the timing penalties due to erroneous branch predictions. Their method collects information on branch target buffer evolution by considering all possible execution paths of a program. This information can then be used to classify control transfer instructions for the purpose of estimating their worst-case branching cost.

Healy et al. [47] introduce static cache simulation for instruction caches. They analyse possible control flows of programs and statically categorize the caching behavior of each access to the instruction memory as *always hit*, *always miss*, *first hit*, or *first miss*. This categorization is then used in the pipeline path analysis to estimate the worst- and best-case execution time of the sequence of instructions. Another development by Mueller [90] generalizes the static cache simulation of direct mapped caches to set-associative caches. Theiling and Ferdinand [127] apply the concept of abstract inter-

pretation to perform both cache analysis and pipeline analysis. Further, Li et al. [77] model the combined effects of caching, branch prediction, and wrong-path instruction prefetching in a unified ILP-based framework for worst-case timing estimation.

Our Technique The micro-architectural modeling in our framework has been largely based on the same platform used in the open-source WCET analysis tool Chronos [76], which provides the capability to model pipelining, branch prediction, and first-level instruction caching. We note, however, that this stage is a relatively independent facet in the interaction between WCET analysis and memory optimization decisions, and simplifying assumptions on the underlying architecture will be made in certain cases in order to speed up the analysis and to more clearly expose the direct effect of our optimization techniques.

4.1.3 WCET Calculation

There exist mainly three different approaches for WCET calculation — *tree-based*, *path-based*, and *implicit path enumeration*.

The tree-based approach estimates the WCET of a program through a bottom-up traversal of its syntax tree and applying different timing rules at the nodes (called "timing schema") [102]. This method is quite simple and efficient. But it is difficult to exploit infeasible paths in this approach as the timing rules are local to a program statement.

Implicit path enumeration techniques [79] represent program flows as linear equations or constraints and attempt to maximize the execution time of the entire program under these constraints. This is done via an integer linear programming (ILP) solver. The result is a quantitative solution from which we can infer which basic blocks in the control flow are executed in the worst-case scenario as well as the execution count for each, but *not*

the exact execution path that leads to that scenario — hence the *implicit* qualifier. Many diverse kinds of flow information have been successfully integrated into ILP [36, 108], increasing the popularity of ILP-based WCET calculation.

Path-based techniques estimate the WCET by computing execution time for the feasible paths in the program and then searching for the one with the longest execution time. Naturally, they can handle various flow information, but they enumerate a huge number of paths. Stappert et al. [116] have sought to reduce the cost of this expensive path enumeration by removing infeasible paths from the flow graph. Their method proceeds by (1) finding the longest program path π, (2) checking for the feasibility of π, and (c) removing π from control flow graph followed by the search for a new longest path if π is infeasible. Stappert et al. also develop the concept of *scope graphs* and *virtual scopes* in order to scale this approach for complex programs.

Our Technique The WCET calculation in our framework adopts the path-based approach, as this technique has the feature of preserving the worst-case path information that can be utilized to perform memory optimization for worst-case performance. The next part of this chapter describe this procedure.

4.2 WCET Analysis with Infeasible Path Detection

In this section, we present a technique for finding the WCET of a program in the presence of infeasible paths without performing exhaustive path enumeration. Our technique traverses the control flow graph to find the longest path, but avoids the high cost of path enumerations by keeping track of more information than just the heaviest path during traversal. We pre-compute possible *conflict relations*, or sources of infeasibility, and maintain the heaviest path for each such source. Thus, even when we find that the

"heaviest" path is infeasible, we do not have to backtrack to find alternative paths. This indeed is the key idea of the approach.

In the following, we define conflict relations, describe how this information is used in the WCET calculation, then evaluate the tightness and efficiency of the estimation.

4.2.1 Infeasible Path Information

First, we describe the inferencing of infeasible path information that is exploited in our WCET calculation method. For efficient analysis, we only detect/exploit infeasible path information within a loop body, that is, we do not detect infeasible paths spanning across loop iterations. Thus, in the following, we consider the control flow graph (CFG) to be a *directed acyclic graph* (DAG), representing the body of a loop. Further, we only keep track of pairwise "conflicts" between branches/assignments, which can be Assignment-Branch (AB) conflicts or Branch-Branch (BB) conflicts.

Definition: Effect Constraints

- The *effect constraint* of an assignment $var := expression$ is $var == expression$.
- The *effect constraint* of a branch-edge e in the CFG for a branch condition c is c if e denotes that the branch is taken.
- The *effect constraint* of a branch-edge e in the CFG for a branch condition c is $\neg c$ if e denotes that the branch is not taken.

Definition: Branch-Branch (BB) and Assignment-Branch (AB) Conflicts

A branch-edge e' has *BB conflict* with a subsequent branch-edge e if and only if

- conjunction of the effect constraints of e' and e is unsatisfiable, and

- there exists at least one path from e' to e in the CFG that does not modify the variables appearing in their effect constraints.

An assignment x has *AB conflict* with a subsequent branch-edge e if and only if

- conjunction of the effect constraints of x and e is unsatisfiable, and
- there exists at least one path from x to e in the CFG that does not modify the variables appearing in their effect constraints.

The meaning of *subsequent* here is in the sense of the topological order of the control flow DAG. The second condition in each of the above definitions also reflects that, from the perspective of an effect constraint, each variable is a *one-time entity created at the point of assignment or modification of its value*. This means that an effect constraint A that uses a variable v *before* modification in fact refers to a *different* entity than an effect constraint B that uses v *after* modification. Conjunction of A and B will thus necessitate a renaming of v and cause the two effect constraints to be incomparable, regardless of whether or not the modification affects the satisfiability of the original conjunction. Certainly, if the modification dominates all paths between A and B, the effect constraint of the modification itself may now be related to B.

Given a program's CFG, we compute the binary relations $BB\_Conflict$ and $AB\_Conflict$. We represent $BB\_Conflict$ between edges e' and e by the tuple (e', e), and $AB\_Conflict$ between assignment x and edge e by the tuple (u, e) where u is the basic block containing assignment x.

Restrictions Since our definition of BB and AB conflicts captures only pairwise conflicts, we cannot detect (and exploit) arbitrary infeasible path information. For example,

$$y = 2; \;\; x = y; \;\; if(x > 3)\ \{...$$

denotes an infeasible path, but it will not be captured in the (restricted) notion of pairwise conflicts. Note that the right-hand-side of an assignment statement can only be a variable or an expression. To avoid the need for expensive data flow analysis, our infeasible path detection technique handles only branch conditions and assignments with constants as their right-hand-side expressions. In other words, the only conditional branches whose edges appear in our *BB_Conflict* and *AB_Conflict* relations are of the form

$$variable \quad relational\_operator \quad constant$$

Similarly, the only assignments which appear in *AB_Conflict* are of the form

$$variable := constant$$

However, this is not a restriction on the programs we can handle; we simply ignore more complicated branches/assignments during path analysis. Moreover, we observe that this simple definition of conflict relations is sufficient for our purposes as we perform the analysis at assembly language level where each individual assignment/branch condition cannot be complex. The featured example above, in particular, can be automatically converted into a series of constant assignments by simple compiler optimizations such as constant propagation.

Example Figure 4.1 shows a program and its corresponding CFG. Here, branch-edge $B1 \rightarrow B2$ and branch-edge $B7 \rightarrow B8$ have BB conflict; branch-edge $B1 \rightarrow B3$ does not have any conflict with either $B7 \rightarrow B8$ or $B7 \rightarrow B9$. Similarly, the assignment $x = 1$ has AB conflict with the edge $B7 \rightarrow B9$ but not with $B7 \rightarrow B8$.

Algorithm 1 describes our technique for deriving infeasible path information in a program. As shown, procedures are treated individually; thus the infeasible path information does not account for procedure call contexts or infeasibility across procedures.

```
int P1 ( int flag, x, y, z )
{
    if ( x > 3 )
        z = z + 1;
    else
        z = flag;
    if ( y == 4 )
        y = y + 1;
    else
        x = 1;
    if ( x < 2 )
        z = z / 2;
    else
        z = z – 1;
    y = y + x – z;
    if ( y > 0 )
        z = x + y;
    else
        z = –1;
    return z;
}
```

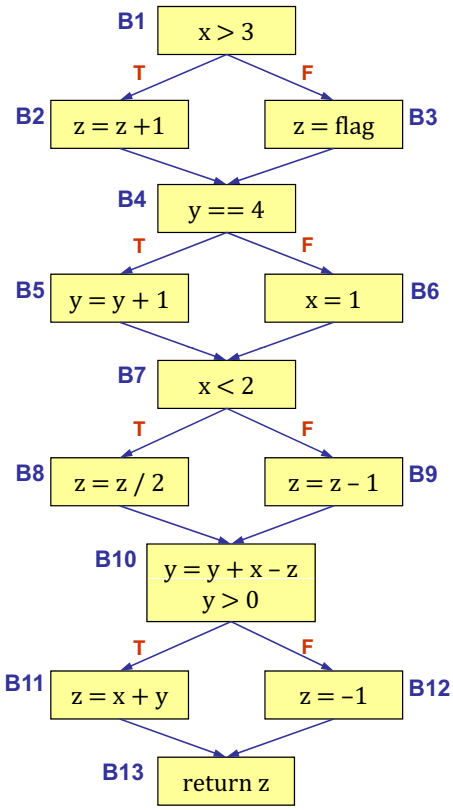


Figure 4.1: An example program and its control flow graph (CFG)

Within a procedure, each loop is in turn considered separately, thus not capturing infeasible paths across loops. However, we take note of assignments within the nested loop or called procedure (lines 10, 12) that may cancel the effect of earlier assignments or branch-edges, so that we do not falsely identify conflicts. The method essentially takes each branch-edge $e = u \rightarrow v$ (line 13) and performs a backward breadth-first traversal of the CFG starting from node u. The traversal along a path terminates when we encounter either an assignment (conflicting or otherwise) to the variable involved in the effect constraint of e (line 18) or a conflicting branch-edge (line 23).

Algorithm Complexity The computation of the $AB\_Conflict$ and $BB\_Conflict$ relations can be accomplished in $O((|V| + |E|) * |E|)$ time for each procedure where $|V|, |E|$ are the number of nodes and the number of edges in the CFG of the procedure.

```
1  AB_Conflict := ∅;
2  BB_Conflict := ∅;
3  foreach procedure P ∈ pgm do
4      foreach loop L ∈ P do
5          Let G be the DAG capturing the control flow in L without the back edge;
6          Detect_Conflicts(G);
7      Let G′ be the DAG capturing the control flow in P;
8      Detect_Conflicts(G′);

   Function Detect_Conflicts(G)
9  foreach loop L ∈ G do
10     Replace L with a dummy node containing all assignments in L;
11 foreach call site CP of procedure P ∈ G do
12     Replace CP with a dummy node containing all non-local assignments in P;
13 foreach branch-edge (e = u → v) ∈ G do
14     Let var be the variable appearing in the effect constraint of e;
15     queue := {u};
16     while queue ≠ ∅ do
17         Dequeue the first node q from queue;
18         if q contains an assignment to var then
19             if last assignment to var in q conflicts with e then
20                 Add (q, e) to AB_Conflict;
21         else
22             foreach predecessor p of q in G do
23                 if (p → q) conflicts with e then
24                     Add (p → q, e) to BB_Conflict;
25                 else
26                     Enqueue p to queue;
```

Algorithm 1: Infeasible path detection in a program pgm

This is because each branch-edge is tested for conflict against all the ancestor nodes and branch-edges in the worst case.

The infeasible path detection technique we have described is now part of the ILP-based path analysis in the open-source WCET estimation tool Chronos v3.0 [76]. Our thesis employs this infeasible path detection algorithm in a more lightweight, path-based WCET calculation framework to be described in the next section.

4.2.2 Exploiting Infeasible Path Information in WCET Calculation

Let us now present our WCET calculation algorithm that exploits the pre-computed conflict relations. The main feature of our technique is that it avoids enumeration of large number of possible execution paths, which is typical in medium to large control-intensive programs.

Algorithm 2 estimates the WCET of a program given the conflict relations. We will illustrate the steps with an example later on. The algorithm calculates the WCET for each individual procedure (lines 1–6) and accounts for this cost at the call sites of the procedure (line 11). We assume that there is no recursive procedure call. Within a procedure, the WCET of each loop is calculated separately and nested loops are analyzed starting from the innermost loop (line 2).

To estimate the WCET of a loop, we find the heaviest *acyclic path* in the loop. An acyclic path is a possible path in a loop iteration, that is, a path in the loop's control flow DAG from source to sink. If the estimated execution time of the heaviest acyclic path is t and the loop bound is lb, then the estimated WCET of the loop is $lb \times t$ (line 4 of Algorithm 2).

Our algorithm traverses the loop's control flow DAG from sink to source (lines 14–26). This traversal constitutes the heart of our method. For a basic block u in the the loop,

foreach procedure $P \in pgm$ in reverse topological order of the call graph **do**
  /* Process innermost loops first */
  foreach loop $L \in P$ in decreasing order of nesting depth **do**
    Let G be the DAG capturing the control flow in L without the back edge;
    $L.cost := L.loopbound \times$ `WCET_Estimate(`G`)`;
  /* Process P, treating loops and procedure calls in P as black boxes */
  Let G' be the DAG capturing the control flow in P;
  $P.cost :=$ `WCET_Estimate(`G'`)`;
Let M be the `main` procedure in pgm; **return** $M.cost$;

Function `WCET_Estimate(`G`)`
foreach loop $L \in G$ **do**
  Replace L with a dummy node of cost $L.cost$, containing all assignments in L;
foreach call site CP of procedure $P \in G$ **do**
  Replace CP with a dummy node of cost $P.cost$, containing all non-local assignments in P;
foreach node $u \in G$ **do** $visited(u)$:= FALSE;
$paths(G.sink) := \{\langle G.sink \rangle\}$; $visited(G.sink)$:= TRUE;
$\langle G.sink \rangle.conflictList := \emptyset$;
/* Traverse from sink to source */
foreach node $u \in (G - G.sink)$ in reverse topological order **do**
  $visited(u)$:= TRUE; $paths(u) := \emptyset$;
  foreach immediate successor v of u **do**
    foreach partial path π in $paths(v)$ **do**
      /* Augment the partial path π with node u if there is no conflict*/
      if $\nexists\, e \in \pi.conflictList$ s.t.
      $(u \rightarrow v, e) \in BB\_Conflict \ \vee \ (u, e) \in AB\_Conflict$ **then**
        $\pi' := \langle u \rangle \circ \pi$; $\pi'.cost := \pi.cost + u.cost$;
        $paths(u) := paths(u) \cup \{\pi'\}$;
        /* Augment $conflictList$ while removing expired elements */
        $newCF := \{u \rightarrow v \mid u \rightarrow v$ appears in $AB\_Conflict$
          or in $BB\_Conflict\}$;
        $cancelCF := \{e \mid e \in \pi.conflictList$ and
          u modifies a variable appearing in e's effect constraint $\}$;
        $expiredCF := \{e \mid e \in \pi.conflictList$ and $\nexists\, x$ s.t. $\neg visited(x)$
          $\wedge ((x, e) \in AB\_Conflict \ \vee \ (x \rightarrow y, e) \in BB\_Conflict)\}$;
        $\pi'.conflictList :=$
          $(\pi.conflictList \ \cup \ newCf) \setminus (cancelCF \ \cup \ expiredCF)$;
  /* Remove partial paths that clearly cannot lead to WCET path */
  foreach partial path $\pi \in paths(u)$ **do**
    if $\exists\, \pi' \in paths(u)$ s.t. $\pi'.conflictList \subseteq \pi.conflictList \ \wedge \ \pi'.cost > \pi.cost$ **then**
      $paths(u) := paths(u) - \{\pi\}$;
Let $\pi \in paths(G.source)$ be the path with maximum cost; **return** $\pi.cost$;

Algorithm 2: Estimating the WCET of a program pgm given infeasible path information

we keep $paths(u)$, a subset of possible execution paths in the subgraph rooted at u that may be part of the overall WCET path. To take into account the infeasible path information, we cannot afford to remember only the "heaviest path so far" at the control flow merge points. This is because the heaviest partial path may have conflicts with earlier branch-edges or assignment instructions resulting in costly backtracking. For each path $\pi \in paths(u)$ we also maintain a $conflictList$, which contains the branch-edges of π that participate in conflict with ancestor nodes and edges of u.

Now let us consider a single backward traversal step from v to u along the edge $u \rightarrow v$ (lines 16–23). We construct $paths(u)$ from partial paths in $paths(v)$ that do not have a conflict with the edge $u \rightarrow v$ or any assignment in u (line 18) by adding node u at the beginning of each of these partial paths (line 19). The $conflictList$ of this extended path contains exactly the edges (a) whose conflicts have not "expired" due to assignments and (b) whose corresponding conflicting branch-edges/assignments have not been visited (lines 20–23).

We notice that a partial path $\pi \in paths(u)$ has no chance of becoming the WCET path if there is another path $\pi' \in paths(u)$ with strictly greater cost and less potential conflicts (that is, its $conflictList$ is subsumed by π's $conflictList$). In that case, π can be removed from the set (lines 24–26). This implies that if the $conflictList$ of a path $\pi \in paths(u)$ becomes empty and π is the heaviest path in $paths(u)$, we assign the singleton set $\{\pi\}$ to $paths(u)$.

Algorithm Complexity In the worst case, the complexity of our algorithm is exponential in $|V|$, the number of nodes in the CFG. This is because the size of $paths(u)$ for some node u may be $O(2^{|V|})$ due to different decisions in the branches following u. In practice, this exponential blow-up is not encountered because (1) branch-edges that do not participate in any conflict are not kept track of, and (2) a branch-edge that conflicts

with other branch-edges/assignments is no longer remembered after we encounter those conflicting branch-edges/assignments during traversal.

Illustration Let us illustrate our WCET calculation by employing it on the control flow DAG of Figure 4.1. The conflicting pairs detected are

$$BB\_Conflict = \{(B1 \rightarrow B2, B7 \rightarrow B8)\}$$

$$AB\_Conflict = \{(B6, B7 \rightarrow B9)\}$$

We traverse the DAG from sink (node $B13$) to source (node $B1$) and maintain a set of paths $paths(u)$ at each visited node u. For each path $\pi \in paths(u)$, we maintain $conflictList$ — a subset of branch-edges drawn from branch decisions made so far. Thus each path in $paths(u)$ is written in the form

$$\langle \text{ Sequence of basic blocks starting with } u \rangle_{conflictList}$$

Starting from node $B13$ in Figure 4.1, our traversal is routine until we reach node $B10$. Here, ϕ denotes an empty set.

$$paths(B13) = \{\langle B13 \rangle_{\phi}\}$$

$$paths(B12) = \{\langle B12, B13 \rangle_{\phi}\}$$

$$paths(B11) = \{\langle B11, B13 \rangle_{\phi}\}$$

At node $B10$, we have two potential paths. However, all branch-edges in these paths, $B10 \rightarrow B11$ and $B10 \rightarrow B12$, do not participate in any conflict relation, hence both paths have empty $conflictList$. Therefore, we only carry the heaviest of the two paths (assuming $B11.cost \geq B12.cost$).

$$paths(B10) = \{\langle B10, B11, B13\rangle_{\phi}\}$$

$$paths(B9) = \{\langle B9, B10, B11, B13\rangle_{\phi}\}$$

$$paths(B8) = \{\langle B8, B10, B11, B13\rangle_{\phi}\}$$

Node $B7$ again has two potential paths, and both of its outgoing edges appear in conflict relations. Until we visit the corresponding conflicting edges or nodes, we cannot determine the feasibility of the partial paths. Consequently, we maintain both paths along with the potentially conflicting edges in the set $conflictList$ associated with each path.

$$\begin{aligned} paths(B7) = \{ \quad & \langle B7, B8, B10, B11, B13\rangle_{\{B7 \rightarrow B8\}}, \\ & \langle B7, B9, B10, B11, B13\rangle_{\{B7 \rightarrow B9\}} \ \} \end{aligned}$$

Moving on to node $B6$, we find that the assignment in node $B6$ conflicts with $B7 \rightarrow B9$ rendering the path

$$\langle B6, B7, B9, B10, B11, B13\rangle$$

infeasible. Thus we only extend one path leading to

$$paths(B6) = \{\langle B6, B7, B8, B10, B11, B13\rangle_{\phi}\}$$

We drop $B7 \rightarrow B8$ from the $conflictList$ as we have encountered an assignment to program variable x in $B6$. The assignment implies that the conflict between $B7 \rightarrow B8$ and $B1 \rightarrow B2$ has "expired" along this partial path.

Indeed, these last two steps show the key source of efficiency in our method. Since we have kept track of both possibilities in which branch at node $B7$ can be resolved, we do not need to backtrack when we find that the branch decision $B7 \rightarrow B9$ can lead to infeasibility. Also, we do not store paths corresponding to the decision of every branch, but only those involved in conflicts. Furthermore, once we have encountered an

assignment to the variable involved in a conflict, we need not keep track of that conflict any further.

Continuing in this way we reach node $B1$; we omit the details for the rest of the traversal. Note that the control flow DAG of Figure 4.1 has four branches and $2^4 = 16$ paths; yet, when we visit any basic block u of the control flow DAG, $paths(u)$ contains at most two paths; that is, exponential blow-up is avoided in this example.

4.2.3 Tightness of Estimation

We evaluate the tightness of our WCET estimation method on several benchmark programs listed in Table 4.1. `adpcm` is the ADPCM coder taken from Mediabench [71]. `display` is an image dithering kernel taken from MPEG-2. `compress` is a data compression program, while `statemate` is a car window controller automatically generated from a statechart specification; both are taken from C-Lab [137]. `susan_thin` from MiBench's automotive application suite is a kernel performing edge thinning [44].

Table 4.1: Benchmark statistics

Benchmark	# Basic Blocks	# Conflicts		# Paths		
		AB	BB	Total	Feasible	Feasible/Total
`adpcm`	32	2	5	1,536	288	18.75%
`display`	37	13	0	96	42	43.75%
`statemate`	334	74	15	6.55×10^{16}	1.09×10^{13}	0.02%
`susan_thin`	93	15	13	146,189,962	33,820	0.02%
`compress`	213	9	3	110	45	40.91%

Table 4.1 shows the basic block count for each benchmark along with the number of AB conflicts and BB conflicts detected by our method. The last three columns of the table give, respectively, the total number of paths, the number of feasible paths, and the ratio of feasible paths out of all paths. The total path counts for the large-scale benchmarks (`statemate` and `susan_thin`) cannot be efficiently obtained via enumeration, and

have thus been estimated as a function of the number of conditional branches present in the respective programs. These two applications are also the benchmarks with huge numbers of infeasible paths, despite the limited conflict detection applied to discover them. We see that less than 1% of the execution paths are actually feasible in both programs. The `statemate` code, having been automatically generated from a statechart, contains a lot of repetitive flag checks that give rise to many infeasible paths. `susan_thin` applies different computations based on a single value that is checked at multiple points; thus the entrance of a computation block renders all partial paths within the other computation blocks infeasible. A general observation is that unoptimized programming practices contribute substantially to infeasible program paths.

We use SimpleScalar toolset [12] for the experiments. The programs are compiled using gcc 2.7.2.3 targeted for SimpleScalar. The binary executables are then run through our prototype analyzer that disassembles the binary to generate the control flow graph (CFG). As our focus is on evaluating the tightening of estimation via infeasible path detection, we assume a simple embedded processor with single-issue in-order pipeline, perfect instruction cache and branch prediction. The execution time corresponding to each basic block is thus easily estimated for this simple architecture. For more complex micro-architectures, we can employ one of the state-of-the-art techniques or tools reviewed earlier in this chapter. We assume that loop bounds required for WCET calculation are provided via manual annotation. All experiments are performed on a 3.0GHz P4 CPU with 1MB cache and 2GB memory.

Table 4.2: Comparison of observed WCET, WCET estimation with and without infeasibility information

Benchmark	Est. WCET (cycles)		Improve-	Obs. WCET	Estimated/Observed	
	with infeas.	w/o infeas.	ment	(cycles)	with infeas.	w/o infeas.
`adpcm`	896,286	907,286	1.21%	717,201	1.25	1.27
`display`	244,187,271	257,556,615	5.19%	229,755,271	1.06	1.12
`statemate`	41,578	44,938	7.48%	31,636	1.31	1.42
`susan_thin`	293,989,241	485,328,185	39.42%	173,769,229	1.69	2.79
`compress`	312,904	383,329	18.37%	25,819	12.12	14.85

Table 4.2 gives the results of our WCET estimation algorithm on the benchmarks, taking into account infeasibility information (*Est. WCET with infeas.*). These are compared with the results of WCET estimation on the same benchmarks when all paths are assumed to be feasible (*Est. WCET w/o infeas.*). The *Improvement* column shows the reduction in the estimated WCET value when infeasibility is considered. As expected, the WCET estimation yields tighter values when infeasibility is taken into account.

To evaluate the tightness of the analysis, we perform simulation on each benchmark. Each benchmark program is run with varying input sets whose ranges are deduced from the program specification and randomly generated where applicable. From the simulation results, we extract the maximum number of execution cycles as presented in the column *Obs. WCET* in the table. Finally, the column *Estimated/Observed* shows the ratio of the estimated WCET values, with and without infeasibility detection, to the observed WCET values: the closer the ratio to 1, the tighter the estimation.

Our analysis gives tight estimates in all benchmarks except `compress`; the result for compress can only be improved by considering infeasibility across loop iterations.

Method Scalability Among existing WCET analysis methods that account for infeasible execution paths, the work of Stappert [116] is closest to ours. As described in subsection 4.1.3, Stappert's method iteratively searches for a longest path, tests its feasibility, and removes the path if it is infeasible. We compare the efficiency of our method with that of Stappert's approach. We provide both methods with the same execution time for basic blocks and infeasibility information (see Section 4.2.1); thus both yield the same WCET path for each benchmark.

Table 4.3 displays the result of the comparison. For our method, the column *# Explored Conflicts* gives the maximum length of $conflictList$ maintained during computation, which is small in all cases. The column *# Explored Paths* shows the maximum number

Table 4.3: Efficiency of our WCET calculation method

Benchmark	Our Method			Stappert's Method	
	# Explored Conflicts	# Explored Paths	Runtime	# Explored Paths	Runtime
`adpcm`	2	2	0.20 ms	5	0.42 ms
`display`	2	2	0.21 ms	7	0.61 ms
`statemate`	15	738	853.52 ms	> 2000	$>$ 36 mins
`susan_thin`	3	12	1.06 ms	> 2000	$>$ 24 mins
`compress`	2	4	1.18 ms	27	3.72 ms

of paths maintained by our technique at any point of time. It is encouraging to note that we only need to keep track of at most 738 partial paths at any time for our benchmarks. This figure depends heavily on the number of conflicting pairs and the distance between the pairwise conflicts. Most of the conflicting pairs are localized; thus they expire quickly and need not be kept track of further. In `statemate`, some conflicting pairs have long "conflict windows", that is, the assignment/branch conditions of a conflicting pair appear far apart in the CFG; this makes it necessary to maintain more partial paths at each node. The number of paths maintained in turn affects the runtime of the algorithm. The *Runtime* column shows that our technique requires less than 1 second for any benchmark. Even for programs with long "conflict windows", our algorithm performs far better than maintaining a single heaviest path throughout the CFG traversal and backtracking when this path turns out to be infeasible.

The last two columns of Table 4.3 give the number of paths examined by Stappert's method and the runtime of the method. We observe that the huge number of infeasible paths in `statemate` and `susan_thin` are the heaviest paths as well. So, in these two benchmarks, we have to terminate the run of Stappert's algorithm after examining 2000 paths without finding a feasible path. The overestimation of the last infeasible path examined by Stappert's method, compared to the feasible WCET value obtained by our approach, is as much as 60% for `susan_thin`.

4.3 Chapter Summary

In this chapter, we have discussed the problem of WCET analysis and presented an enhanced WCET estimation algorithm that takes into account limited infeasible path information in an efficient way. The infeasible path detection technique as described here has also been integrated into an open-source WCET analysis tool Chronos v3.0 [76].

Chapter 5

Predictable Shared Cache Management

As caches have been the conventional choice for a large portion of the computing market, we dedicate this chapter to consider utilization of shared caches in a predictable manner. This is achieved through a combination of locking and partitioning mechanisms. We explore possible design choices and evaluate their effects on the worst-case application performance. Our study reveals certain design principles that strongly dictate the performance of a predictable memory hierarchy.

5.1 Introduction

Multi-core architectures are increasingly common in both desktop and embedded markets. Energy and thermal constraints are effectively precluding the design of complex high-performance single-core processors. In this context, multiple simpler processing cores on a single chip is an attractive option. Several manufacturers (e.g., Intel, AMD)

have released dual-cores while Sun Microsystems' Niagara multiprocessor accommodates 8 cores on the same die. Intel has developed an 80-core processor prototype to be released by 2011. In the embedded domain, ARM MPCore is a synthesizable processor configurable to contain between 1 and 4 ARM11 cores. IBM Cell processor in Sony PlayStation 3 contains 9 cores while Xenon in Microsoft's Xbox 360 is a custom PowerPC-based triple-core.

Multi-core architectures can dramatically improve performance by distributing the tasks or threads of an application among the cores. At the same time, the software design and optimization efforts involved to exploit this performance potential are extremely complex. Extensive resource sharing among the tasks in multi-core architectures (due to shared bus, memory hierarchy, and so on) leads to higher timing unpredictability. The challenge in implementing a real-time application on a multi-core architecture, therefore, is to perform effective resource sharing without sacrificing the timing predictability. In this chapter, we focus on exploiting the shared cache memory to achieve higher performance while maintaining real-time performance guarantees.

A popular choice for the on-chip memory structure is a two-level cache hierarchy (*e.g.*, Niagara, Xenon, Power5). In this architecture, Level 1 (L1) caches are attached to, and privately accessible by, each core. All the cores share the access to a large Level 2 (L2) cache. This architecture is implemented for example in Power5 dual-core chip [112], XBox360's Xenon processor [19] and Sun UltraSPARC T1 [123]. The presence of a shared cache offers the flexibility in adjusting the memory allocated per core according to its requirement, as well as the possibility of multiple cores enjoying fast access to shared code/data. This potentially improves the performance, but also requires complex resource management.

The interaction and resulting contention among multiple cores in a shared cache bring out many new challenges. To the best of our knowledge, no static analysis method has

been developed to estimate WCET bounds in the presence of shared caches. In our opinion, it will be extremely difficult, if not impossible, to develop such a method that can accurately capture the contention. Instead, we propose to use the shared cache in a restrictive manner that eases the analysis effort, with possible tradeoff in performance. Towards this end, we exploit two mechanisms: *cache locking* and *cache partitioning*.

Let us briefly review these mechanisms, which we have described in sections 2.1.2 and 2.1.3. *Cache locking* allows the user to load selected contents into the cache and subsequently prevents these contents from being replaced at runtime. Locking enables software control over a traditional cache, so that static analysis techniques can associate deterministic latency to each memory access leading to safe and tight timing bounds. *Cache partitioning* assigns a portion of the cache to each task (or processor), and restricts cache replacement to each individual partition. Cache partitioning enables compositional analysis where the timing effect of each task (or processor) can be estimated separately.

The interplay among processing elements in a multi-core setting through the shared cache provides some unique design choices and opportunities. One simple choice, for example, is global cache locking where contents are selected from all the tasks in all the cores. More sophisticated policies may partition the cache among cores or among tasks, and manage each partition independently. The relative merits of these different design choices are not obvious.

In this chapter, we explore the possible design choices for a predictable and high-performance shared L2 cache on multi-core architectures. We devise different combinations of cache locking and partitioning schemes, then study their impact on the worst-case performance of applications with different characteristics. Based on this study, we recommend appropriate locking/partitioning strategy depending on the nature of the application. Given the increasing popularity of multi-cores, this study can pro-

vide guidelines to real-time application programmers in terms of design decisions for the memory hierarchy.

5.2 System Settings

We consider a homogeneous multi-core architecture consisting of identical cores. The on-chip memory is configured as a two-level cache hierarchy with a *shared L2 cache*, which is the focus of this work. We assume that the cache coherence is implemented in hardware, and that the caches support locking and set-based partitioning (see subsection 2.1.3). This work focuses on *instruction caches*, although our technique is equally applicable to data caches.

Our framework adopts the classic real-time system model where a set of independent tasks $\{t_1, t_2, \ldots, t_N\}$ is executed periodically. Each task t_i is associated with its period p_i, which also defines its deadline, and its worst-case execution time c_i. We choose the *partitioning* [23] strategy for homogeneous multiprocessor scheduling. In the partitioning strategy, once a task is allocated to a processor, it is executed exclusively on that processor. Any uniprocessor scheduling algorithm can then be applied on each processor. The partitioning strategy has the advantage of lower overhead compared to the global strategy, which allows tasks to migrate to different processors at runtime.

López et al. [82] show that the *Earliest Deadline First (EDF)* scheduling policy coupled with *First Fit (FF)* allocation is an optimal partitioning approach with respect to utilization bounds. Our framework applies this policy. FF assigns a task to the first processor that can accept it. A task set is EDF-schedulable on a uniprocessor if $U \leq 1$, where U is the utilization of a task set $\{t_1, t_2, \ldots, t_N\}$ given by

$$U = \sum_{i=1}^{N} \frac{c_i}{p_i}$$

The *system utilization* of a Q-core multiprocessor is

$$U_{system} = \frac{U}{Q}$$

We measure the performance of a task set on a multiprocessor by the system utilization: lower system utilization implies higher schedulability and thus better performance.

5.3 Memory Management Schemes

We separate the treatment of the private L1 caches and the shared L2 cache, in order to observe the shared cache behavior while abstracting out the effects of the L1 caches. We first decide on the memory blocks to be locked into L1. As our focus is on the shared cache, we choose a simple global static locking scheme for L1. The private L1 cache attached to a core is utilized only by the tasks executing on that core; for each, we adopt the cache content selection algorithm for multitasking systems [106]. The chosen blocks for L1 will be excluded during content selection for the L2 cache.

The shared L2 cache opens up the opportunity to combine different locking and partitioning schemes to achieve high cache hit rate, as shown in Table 5.1.

Table 5.1: Design choices for shared cache

	Static Locking	Dynamic Locking
No Partition	***SN***	–
Task-based Partition	*ST*	***DT***
Core-based Partition	***SC***	***DC***

For cache locking, we can choose a *static* scheme or a *dynamic* scheme. Recall that in the *static* cache locking scheme, the cache content is loaded once and stays in the cache throughout the execution of the task; whereas, in *dynamic* cache locking scheme, the cache content can be reloaded at runtime. Dynamic cache locking requires an applica-

tion/task to be divided into multiple *regions*. An offline analysis selects memory blocks to be locked for each region. As the execution moves from one region to another, the cache content is replaced with that of the new region.

For cache partitioning, we have the following choices.

1. *no partition*: A cache block may be occupied by any task, scheduled on any core.
2. *task-based partition*: Each task is assigned a portion of the cache.
3. *core-based partition*: Each core is assigned a portion of the cache, and each task scheduled on that core may occupy the whole portion while it is executing.

From these, the {*dynamic locking, no partition*} combination must be ruled out, because a dynamic locking scheme strictly requires a dedicated partition. Further, both the {*static locking, no partition*} (*SN*) and the {*static locking, task-based partition*} (*ST*) schemes lock the cache contents chosen from all tasks in the application throughout execution, but *SN* offers more flexibility by not enforcing a concrete boundary. *Thus ST is either inferior or at most as good as SN; we eliminate ST from our evaluation*.

Figure 5.1 illustrates the four eligible possibilities, applied on a multi-core with 2 processing elements (PE_1, PE_2) and 4 independent tasks ($t_1, \ldots t_4$). The scheduler assigns t_1, t_2 to PE_1 and t_3, t_4 to PE_2. t_1 and t_4 can each be divided into two regions for dynamic cache locking. We assume a 2-way set-associative shared L2 cache with 8 sets. The rest of this section details each of the four schemes.

5.3.1 Static Locking, No Partition (SN)

This is the simplest scheme where the cache content is kept unchanged throughout application runtime (Figure 5.1a). A cache block can be assigned to any task irrespective

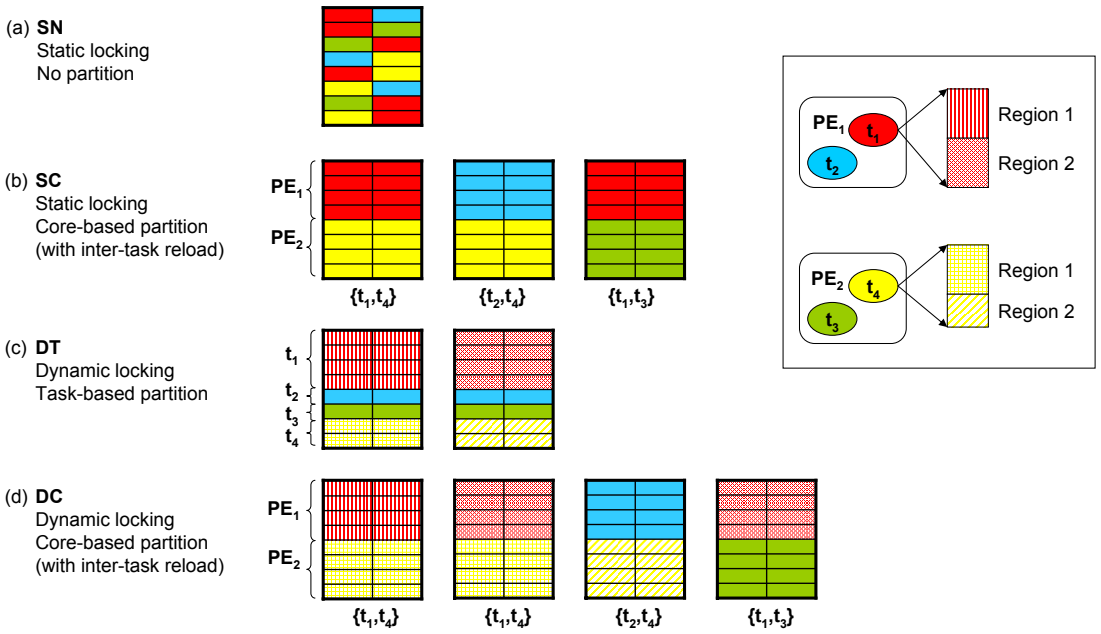


Figure 5.1: Different locking and partitioning schemes for the shared L2 cache

of the processor it is scheduled on. This scheme offers maximum flexibility; however, its performance is restricted if the code size of all the tasks together far exceeds the L2 cache size. For static locking, we apply the cache content selection algorithm presented in [106], which minimizes the system utilization.

5.3.2 Static Locking, Core-based Partition (SC)

In the *SN* scheme, all the tasks occupy cache blocks throughout the execution. On a system with preemptive scheduling, only memory blocks belonging to the "active" tasks are useful at any time. When a task t_i is preempted by t_j on one of the cores, we can replace t_i's memory blocks in the cache with those of t_j's. This comes at the cost of reloading the cache at every preemption. This scheme requires the cache to be partitioned among the cores, as each core has an active task at any point of time and the cores invoke preemptions independently. However, when a task runs on a core, it can occupy the entire partition for that core. In Figure 5.1b, PE_1 gets the first four sets;

PE_2 gets the rest. Initially t_1 occupies PE_1's partition and t_4 occupies PE_2's partition. When t_2 preempts t_1, it loads and locks PE_1's partition with its own content. We adapt a dynamic programming based optimal partitioning algorithm [111] here. This scheme is still categorized as static locking because no reloading is performed within a task.

5.3.3 Dynamic Locking, Task-based Partition (DT)

Dynamic locking allows more memory blocks to fit in the cache via runtime load and lock. The overhead is the cache reload cost every time the execution of a task moves from one region to another. As different tasks have different region formations, the cache is first partitioned among the tasks. Each task then performs dynamic locking within its partition. Figure 5.1c shows the scheme at work for t_1 and t_4. In contrast to *SC*, reloading is performed intra-task. No inter-task reloading is required as the partitioning prevents interference among the tasks, thus preemptions incur no cache reload overhead. However, if the application comprises a large number of tasks, such rigid partitioning might not be effective. *DT* also suffers from the same drawback as *SN*: tasks occupy cache blocks even when they are inactive (preempted). We employ the dynamic locking algorithm in [104] here.

5.3.4 Dynamic Locking, Core-based Partition (DC)

In this most complex scheme, reloading is supported within a task in addition to reloading at preemption (see Figure 5.1d). Initially, the cache is loaded with region 1 of t_1 and region 1 of t_4. As time progresses, t_1's execution (on PE_1) moves to region 2, which then replaces the content of PE_1's portion of the cache. Later on, a preemption brings into the cache the content associated with t_2. However, when t_1 resumes, it again brings in region 2 into the cache.

5.4 Experimental Evaluation

We choose a dual-core multiprocessor for experimental evaluation. Our pool of tasks comprises independent C programs chosen from popular WCET benchmarks listed in Table 5.2. An initial WCET analysis assumes all code blocks are located in the off-chip memory. The resulting WCET values are shown in the table. The column *Potential Regions* shows the number of code regions formed by the dynamic locking algorithm when the single task is allocated cache space half its code size. From this pool, we construct task sets of 4 tasks each. These task sets are chosen to exhibit different characteristics (region formation, code size, and WCET) that affect the effectiveness of the locking and partitioning schemes, as listed under each chart in Figure 5.2. Task periods are set such that the utilization of each core is very close to 1, that is, the task set is schedulable with maximum system utilization.

For both L1 and L2 caches, we choose the line size of 16 bytes (2 instructions) and the associativity of 2. The size of L1 caches is fixed at 128 bytes per core. The small size is chosen so that memory requirements are mostly served from the shared cache, enabling better observation of the multiprocessing effect. Given the average code size of 4 KB per task, we vary the size of the shared L2 cache from 1 KB to 4 KB. We assume that the L1 latency is 1 clock cycle and the L2 latency is 4 cycles. The off-chip latency in real-world systems may vary from about 10 to hundreds of cycles; we choose to fix it at 10 cycles in these experiments to avoid extreme scale difference across benchmark execution times.

In order to better observe the cache effects, we assume that preemption cost is negligible except for the cache reloading. In this way, non-core-based schemes (*SN*, *DT*) do not incur any preemption cost. The only preemption cost incurred by the core-based schemes (*SC*, *DC*) is the latency required to load the cache partition with the selected contents. Note that this cache reload cost is entirely predictable, and is not to be con-

Table 5.2: Benchmarks comprising the task sets

Benchmark	# Potential Regions	Codesize (bytes)	WCET (cycles)	Description
adpcm	4	12,568	3,152,246	Adaptive differential pulse code modulation
cnt	3	1,648	54,228	Matrix sum and count benchmark test
crc	4	2,032	802,866	Cyclic Redundancy Check operation
des	9	6,400	905,692	Data Encryption Standard
desu	6	10,488	889,519	des with selected loops unrolled
edn	9	13,488	983,034	Simple vector multiply
expint	6	1,760	512,824	Loop-intensive calculations
fdct	3	5,424	45,891	Integer implementation of forward DCT
fir	6	3,928	275,722	FIR filter and Gaussian function
idct	2	11,456	30,402	Inverse 2-D DCT, Chen-Wang algorithm
isort-20	3	840	57,212	Insertion sort on a list of 20 elements
isort-20s	17	8,048	53,256	isort-20 with outer loop unrolled
isort-100	2	2,120	1,381,692	Insertion sort on a list of 100 elements
isort-100s	18	9,560	1,389,022	isort-100 with outer loop unrolled
jfdctint	3	5,512	45,962	JPEG integer implementation of forward DCT
lms	5	4,648	6,979,009	LMS adaptive signal enhancement
matmul-10	3	800	376,623	10 x 10 matrix multiplication
matmul-10s	8	4,776	269,308	matmul-10 with outermost loop unrolled
matmul-10u	2	6,976	257,223	matmul-10 with inner loops unrolled
matmul-20	3	800	2,936,153	20 x 20 matrix multiplication
matmul-20s	17	9,496	2,100,158	matmul-20 with outermost loop unrolled
matmul-20u	2	6,720	1,974,153	matmul-20 with innermost loop unrolled
matsum-10	1	512	28,523	10 x 10 matrix summation
matsum-10s	8	2,376	17,698	matsum-10 with outer loop unrolled
matsum-10u	1	3,256	2,447	matsum-10 with all loops unrolled
matsum-100	2	1,016	5,050,275	100 x 100 matrix summation
matsum-100s	8	7,392	5,138,235	matsum-100 with outer loop unrolled
matsum-100u	2	5,736	4,063,275	matsum-100 with inner loop unrolled
minver	6	6,152	53,306	Matrix inversion for 3x3 floating point matrix
qurt	3	1,936	19,787	Root computation of quadratic equations
st	6	3,320	33,213,872	Computation of statistics correlation of 2 arrays
whet	6	4,376	6,570,768	Whetstone benchmark

fused with *cache-related preemption delay* for unlocked caches (see Section 3.1), which refers to the non-deterministic delay due to the burst of cache misses after resuming from preemption. In our predictable caching schemes, this delay is a known constant.

For all task sets, we first select the L1 contents for each core, then apply the four shared caching schemes *SN*, *DT*, *SC*, and *DC*. This gives us the set of locked memory blocks for each task. Note that a task may not be allocated any space in the cache in certain schemes. Based on these locked sets, we calculate the new WCET of each task, taking into account any extra latency involved to reload the cache contents. Finally, we compute the system utilization. Figure 5.2 shows the system utilization computed for each task set given the various schemes and varied cache sizes. These values are normalized against the system utilization without a cache. The scheme that achieves lower system utilization value in these results will be considered more effective in optimizing system performance, leading to higher chance of schedulability.

Cache Partitioning Strategy We have three possible choices for cache partitioning as shown in Table 5.1: *no partition, task-based partition,* and *core-based partition*. The experimental results indicate that the best partitioning choice strongly depends on the locking scheme (static or dynamic) used in conjunction and cache size.

First let us consider static cache locking. As mentioned before, we have eliminated task-based partition (*ST*) from consideration as it is provably inferior compared to no partition (*SN*). This is due to the partitioning granularity of *ST*, which requires the cache partition for a task to contain power of 2 sets. *SN*, in contrast, allows complete flexibility and a task is free to occupy as little as one cache block.

However, when it comes to no partition versus core-based partition (*SN* versus *SC*), the situation is reversed; *SC* consistently performs better than *SN* irrespective of cache size and application characteristics. Recall that in *SC*, the entire partition for a core is

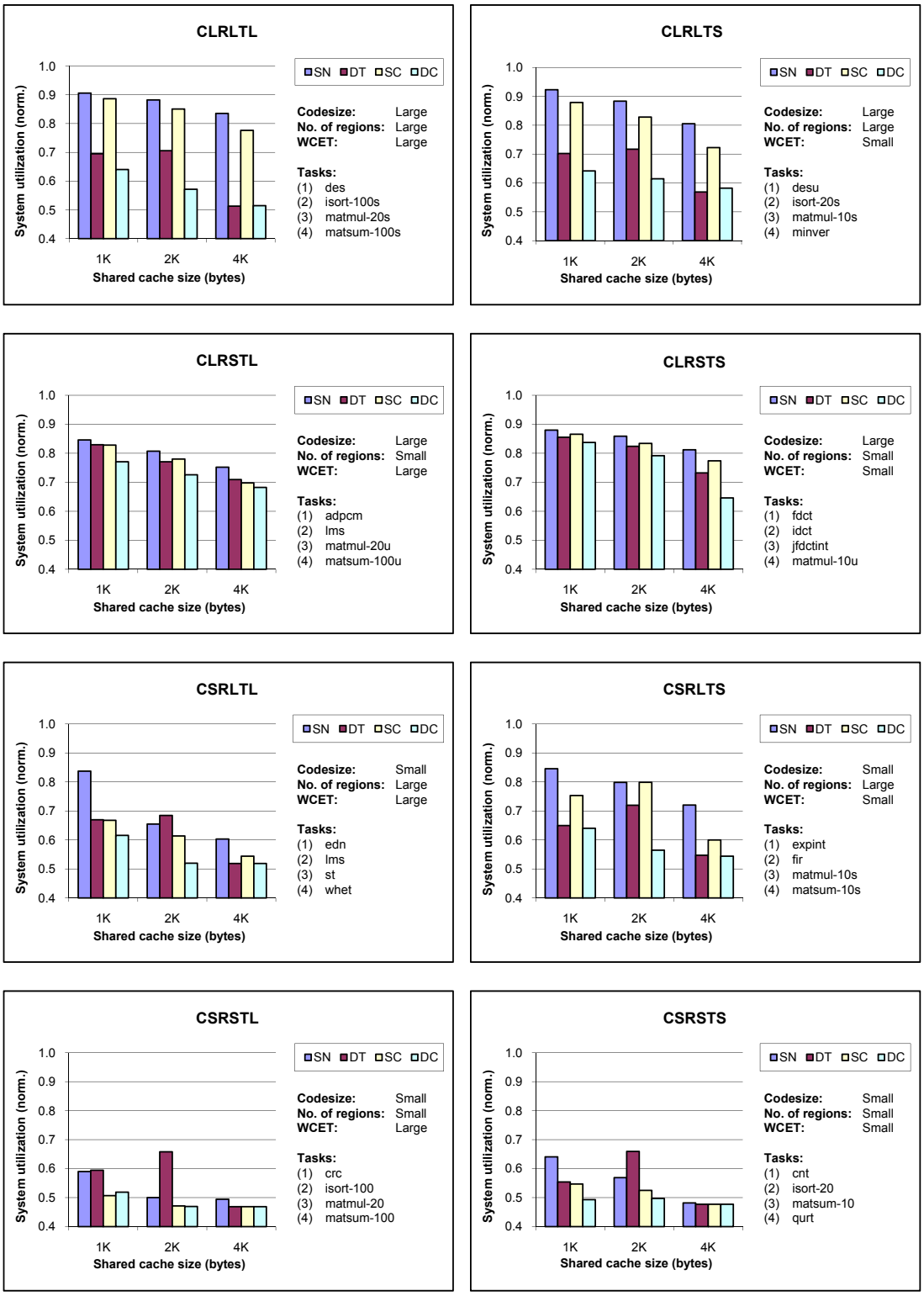


Figure 5.2: Effects of shared caching schemes *SN*, *DT*, *SC*, and *DC* on task sets with various characteristics

occupied only by the "active" tasks at any point of time. For *SN*, in contrast, the "idle" tasks continue to occupy precious real-estate in the shared cache. The downside of *SC* is of course the cache partition reloading and locking cost at every preemption. However, the results indicate that preemption cost does not over-shadow the advantage of better cache utilization by *SC*.

The best partitioning strategy, when used in conjunction with dynamic cache locking scheme, depends heavily on cache size. Here we are comparing *DT* versus *DC*). As dynamic locking allows better cache utilization through intra-task reloading at region boundaries, *DT* becomes a competitive scheme. Its main advantage is zero interference among the tasks, thus avoiding cache reloading at task preemption. *DC*, on the other hand, offers more cache space per task, thus performs better at small cache sizes. As cache size increases, the difference between the two is negligible.

Static versus Dynamic Cache Locking Here the best choice is strongly influenced by application characteristics, cache size, and the cache partitioning strategy.

Let us first consider core-based partition (*SC* versus *DC*). Clearly, *DC* can better utilize the cache if the application has many hot regions and the cache space is limited. The experiments validate this observation with *DC* performing better for task sets with large number of regions (*C*RLT**). Even for those task sets, *DC* gives diminishing returns as cache size increases. If the constituent tasks do not contain profitable regions (*C*RST**), then it is better to use *SC*, which is a much simpler cache management scheme.

We now turn to task-based partition. Normally, we would compare *ST* against *DT*, but as already discussed, *ST* is never better than *SN*. We thus compare *DT* with *SN* instead. The trend is roughly the same as *SC* versus *DC*: *DT* wins if the tasks have a number of hot regions. However, there is one important difference. As we choose a reload point within a task only if it is profitable to do so, *DC* cannot perform worse than *SC*. We

cannot make this claim for *DT* over *SN*, as *DT* and *SN* use different schemes for cache partitioning. Indeed, *SN* enjoys much more flexibility in cache allocation to tasks. The rigid partitioning of *DT* can completely over-shadow the gain from reloading at region boundaries. Thus *SN* can perform better than *DT* when (1) tasks have very few regions (*CSRSTL* and *CSRSTS*) or (2) some tasks get no space in the shared cache (*CSRLTL*) with *DT* policy.

Guiding Design Principles From the preceding discussion, we can make the following general conclusions.

- The combination {*static locking, core-based partition*} (*SC*) performs better than {*static locking, no partition*} (*SN*), which in turn performs better than {*static locking, task-based partition*} (*ST*) irrespective of cache size and application characteristics. In other words, a design that uses static cache locking should apply core-based partition.

- The combination {*dynamic locking, core-based partition*} *DC* performs better than {*dynamic locking, task-based partition*} *DT* for small shared cache size. They are comparable for large cache size; avoiding cache reloading at preemption via task-based partitioning does not seem to affect performance much. In conclusion, core-based cache partition emerges as overall winner independent of locking strategy.

- Dynamic cache locking is better than static cache locking *only* for tasks with a large number of hot regions and for smaller shared cache size. Moreover, designers should be careful when using task-based partitioning in conjunction with dynamic cache locking. If some tasks do not get any cache allocation, the overall system utilization may be severely affected.

5.5 Chapter Summary

In this chapter, we have explored predictable caching schemes for shared memory multi-cores in the context of preemptive hard real-time systems. In particular, we have developed and evaluated various design choices for the shared L2 cache by exploiting static/dynamic locking and task/core-based partitioning. We have studied system utilization for the different choices with respect to the characteristics of the task set and cache size. Our study reveals some interesting guiding principles for real-time system designers with respect to the memory hierarchy.

Chapter 6

Scratchpad Allocation for Sequential Applications

In this chapter, we discuss scratchpad allocation for data memory that aims to minimize the WCET of sequential applications. We first develop a solution based on integer linear programming (ILP) that constructs the optimal allocation assuming that all program paths are feasible. Next, we exploit infeasible information in the allocation via optimal branch-and-bound search to achieve better accuracy. However, the branch-and-bound search is too time-consuming in practice. Therefore, we design fast heuristic searches that achieve near-optimal allocations for all our benchmarks.

6.1 Introduction

Significant research effort has been invested in developing efficient allocation techniques for scratchpad memory [14, 100, 101]. However, all these techniques aim to reduce the average-case execution time (ACET) by utilizing extensive data memory access profiles. An optimal allocation for ACET may not necessarily be the optimal allocation for

WCET. The main difficulty in developing optimal scratchpad memory allocation technique for WCET is the following. An ACET-guided allocation method uses the access frequencies of variables obtained through profiling. In WCET-guided allocation, we are interested in the access frequencies of the variables along the worst-case path. As we allocate variables along the worst-case path into the scratchpad memory, a new path may become the worst-case path. This leads to a different access frequency profile of the variables corresponding to the new worst-case path. As a result, locally optimizing the current worst-case path may not lead to the globally optimal solution. The elegant techniques used in ACET-guided optimal allocations, such as 0-1 knapsack, are not applicable to WCET-guided allocation.

With this motivation, we propose customized optimal and near-optimal allocation techniques that are guided by WCET. We consider data objects for allocation as they are more difficult to handle as far as timing predictability of real-time tasks is concerned. Our allocation technique can also be applied to code objects with minimal modification.

Our first optimal allocation technique is based on a simple ILP formulation. This solution does not take infeasible path information into account. Therefore, it can potentially allocate objects from a heavy (w.r.t. execution time) but infeasible path, leading to misdirected optimization and consequently sub-optimal performance gain. To overcome this drawback, we propose another optimal allocation technique based on branch-and-bound search that does exploit infeasible path information. But branch-and-bound search may be inefficient for large number of variables and large scratchpad sizes. So we also design a greedy heuristic algorithm that is efficient, considers infeasibility information, and produces near-optimal solutions. To be effective, these allocation techniques require repeated search for the worst-case path and the access frequencies of the variables along that path. This is enabled by the efficient WCET analysis we described in Chapter 4.

6.2 Optimal Allocation via ILP

In this section, we address the problem of allocating data variables to scratchpad memory so as to reduce the WCET of a program. This first setting does not take into account any infeasible path information, that is, all paths in the control flow graph are considered feasible. In the next section, we consider optimal and near-optimal allocation of data variables to scratchpad memory by considering infeasible path information.

Assumptions Our WCET-guided allocation method is static, that is, the allocation of variables is fixed at compile time. We consider both scalar variables and arrays. An array can be allocated only if the entire array fits into the scratchpad. We consider for allocation the global variables and the stack variables (parameters, local variables, and return variables) corresponding to non-recursive functions. We do not consider stack variables corresponding to recursive functions because multiple instances of these variables may exist during program execution. For non-recursive functions, our method treats the stack variables just like global variables, that is, these stack variables are allocated for the entire execution of the program. This restriction can be relaxed in a manner similar to [14] by taking into account functions with disjoint lifetimes.

ILP Formulation We now present our allocation method based on integer linear programming (ILP). Recall that names starting with capital letters are used for ILP variables and names starting with small letters represent constants. First we develop a scheme for allocating data variables appearing in a single program loop; later we extend the technique to general programs.

Let us consider the directed acyclic graph (DAG) capturing the control flow in the loop body, that is, the control flow graph of the loop body without the loop back-edge. We assume that the DAG has a unique source node and a unique sink node. If there is no

unique sink node, then we add a dummy sink node. Each path from the source to the sink in the DAG is an *acyclic path* — a possible path in a loop iteration.

For each data variable v in the program we define a $0-1$ decision variable S_v which indicates whether v is selected for scratchpad allocation. Thus

$$S_v \geq 0; \quad S_v \leq 1$$

$$\sum_{v \in allvars} S_v * area_v \ \leq \ scratchpad\_size$$

where $allvars$ is the set of program variables, $area_v$ is a constant denoting the scratchpad area to be occupied by v if it is allocated and *scratchpad_size* is a constant denoting the total size of the scratchpad memory available.

We consider the DAG representing the loop body's control flow and define a variable W_i for each basic block i in the DAG. Variable W_i denotes the cost of the worst-case path in the DAG rooted at basic block i under the allocation captured by the S_v variables. For each outgoing edge $i \rightarrow j$ from basic block i in the DAG, we have the following constraint.

$$W_i \ \geq \ W_j + (cost_i - \sum_{v \in vars(i)} S_v * gain_v * n_{v,i})$$

where $cost_i$ is a constant denoting the execution time (in terms of cycles) of basic block i without any allocation. The term $vars(i)$ denotes the set of program variables appearing in basic block i, while $gain_v$ is a constant denoting the gain (in number of cycles) of a single access of v by allocating v to scratchpad memory, and $n_{v,i}$ is the number of occurrences of v in basic block i. For the sink node of the DAG, which has no outgoing edge, we define W_{sink} as follows.

$$W_{sink} \ = \ cost_{sink} - \sum_{v \in vars(sink)} S_v * gain_v * n_{v,sink}$$

Clearly, the variable W_{src} (for the source node of the DAG) captures the worst-case acyclic path under the allocation given by S_v variables. Thus, we define the objective function as

$$\text{minimize } W_{src} * lb \tag{6.1}$$

where *lb* is a known constant denoting the maximum number of loop iterations.

The ILP solver finds the assignment of S_v variables (*i.e.*, the scratchpad allocation) which minimizes the worst-case execution time of the loop.

Extension to Full Programs In the preceding, we determine the optimal scratchpad allocation for a single program loop. To extend our formulation to whole programs, we first need to generate the constraints for each innermost program loop as mentioned above. Next, we transform the program's control flow graph by converting each innermost loop to a "basic block", where the cost of each innermost loop is given by the objective function of Equation 6.1. We can now construct the constraints for loops in the next level of loop nesting. We go on in this fashion until we have reached the topmost level of loop nesting; this gives us all the ILP constraints. The new objective function to be minimized is now W_{entry}, where $entry$ is the only entry node in the program's control flow graph (in a C program, the entry block of the main function).

6.3 Allocation via Customized Search

In this section, we present search algorithms for generating optimal and near-optimal allocations by taking into account infeasible path information. We have incorporated infeasible path detection into our WCET analysis technique, which we describe in Chapter 4. Here we show how the problem of variable allocation to minimize WCET can be formulated, where the WCET analysis takes into account infeasible path information.

Given the size of the scratchpad memory *scratchpad_size*, we define an allocation as a set

$$V \in 2^{allvars} \text{ s.t. } \sum_{v \in V} area_v \leq scratchpad\_size$$

where the set $2^{allvars}$ denotes the power-set of $allvars$.

Let $WCET_V$ be the WCET after allocating the set of variables V into the scratchpad memory. We want to choose the "optimal" allocation, that is, the allocation $V \in 2^{allvars}$ that produces the minimum $WCET_V$. As before, $allvars$ denotes the set of all variables accessed in the program, and $area_v$ denotes the size of variable v.

Finding the optimal variable allocation for WCET reduction requires the contribution of each variable towards the WCET. However, we cannot define the contribution of a variable towards the WCET as a constant. This is because of the following reasons.

- First, the contribution of a variable towards the WCET is dependent on the current WCET path. However, allocation of that variable may result in a new WCET path. Therefore, the reduction in WCET due to allocation of a variable is, in general, *not* equal to the contribution of the variable towards the current WCET.
- Secondly, the reduction in WCET due to allocation of two or more variables is not accumulative. That is, if the WCET reduction by allocating variables v_1 and v_2 are X_1 and X_2, respectively, then the WCET reduction by allocating variables v_1 and v_2 together can be less than $(X_1 + X_2)$.

To illustrate these two points, consider two paths π_1 and π_2 having execution times $tm(\pi_1)$ and $tm(\pi_2)$ assuming no variable allocation to scratchpad memory (Figure 6.1). Suppose $WCET = tm(\pi_1) = tm(\pi_2) + \epsilon$. Consider two variables v_1 and v_2 where v_1 is accessed in π_1 but not in π_2, while v_2 is accessed in π_2 but not in π_1. Let $ct(v_1, \pi_1) = ct(v_2, \pi_2) > \epsilon$ where $cost(v_k, \pi_k)$ denotes the contribution of variable v_k towards the execution time of path π_k.

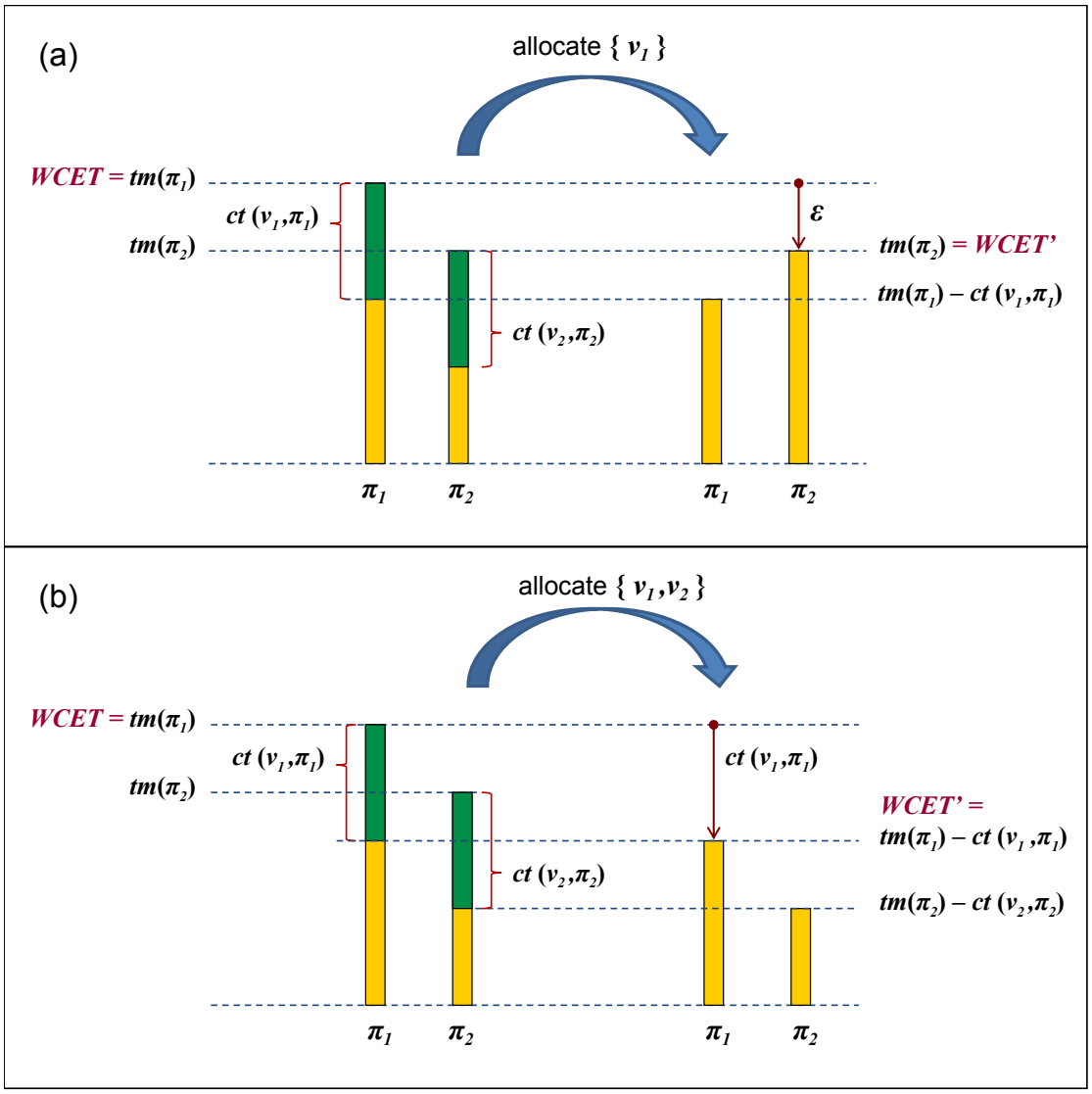


Figure 6.1: Non-constant WCET reduction due to variable allocation

Allocating v_1 in the scratchpad memory reduces $tm(\pi_1)$ by $ct(v_1, \pi_1)$, as illustrated in Figure 6.1a. But π_2 now becomes the WCET path; thus, the reduced worst-case execution time is $tm(\pi_2)$. The reduction in WCET is ϵ instead of $ct(v_1, \pi_1)$. Furthermore, allocating v_1 and v_2 together reduces both $tm(\pi_1)$ and $tm(\pi_2)$, as illustrated in Figure 6.1b. As $ct(v_1, \pi_1) = ct(v_2, \pi_2)$, the reduced worst-case execution time is $tm(\pi_1) - ct(v_1, \pi_1)$. Here, the WCET reduction does not involve the contribution of v_2 at all.

As we cannot define the contribution of a variable towards the WCET as a constant, the optimal allocation problem for WCET reduction cannot be formulated as a knapsack

problem. Furthermore, as the contributions of consecutively allocated variables do not accumulate, the "optimal substructure" property required for dynamic programming is absent. This rules out an optimal dynamic programming solution. Following this, we use a branch-and-bound search algorithm to obtain the optimal solution.

6.3.1 Branch-and-Bound Search

The general paradigm of branch-and-bound deals with optimization problems over a search space that can be presented as the leaves of a search tree. The search is guaranteed to find the optimal solution, but its complexity in the worst case is as high as that of exhaustive search. In our case, the search space consists of the set of all possible allocations $V \in 2^{allvars}$.

Each level k in the branch-and-bound search tree corresponds to the decision of including or excluding a variable $v_k \in allvars$ into the solution set V. Thus, each node m at level k corresponds to a partial allocation $allocation(m)$ with the decision about the variables v_1 up to v_k, *i.e.*, $allocation(m) \subseteq \{v_1, \ldots, v_k\} \subseteq allvars$. Whenever we reach a leaf node of the search tree, we have a complete allocation. We then calculate the reduced WCET corresponding to this allocation. The reduction in WCET is the difference between the original WCET (without any allocation) and the reduced WCET. During the traversal of the search tree, the maximum WCET reduction achieved so far at any leaf node is kept as a bound B. At any non-leaf node m in the search tree, a **heuristic function** computes an upper bound, $UB(m)$, on the maximum possible WCET reduction at any leaf node in the subtree rooted at m. If $UB(m) < B$, then the search space corresponding to the subtree rooted at m can be pruned. Clearly, the choice of the heuristic function UB is crucial in deciding the amount of search space pruning achieved.

We define the heuristic function for a node m at level k of the search tree as follows. First, we compute the WCET reduction, $reduction(m)$, corresponding to the partial allocation $allocation(m)$ at node m. The upper bound, $UB(m)$, is the sum of $reduction(m)$ and the maximum potential reduction in WCET due to the allocation of the variables not yet considered, *i.e.*, $\{v_{k+1}, \ldots, v_{|allvars|}\}$. An estimation of the latter is formulated as a simple knapsack problem, which is solved using dynamic programming. The inputs to the knapsack problem are as follows.

1. Variables: $v_{k+1}, \ldots, v_{|allvars|}$
2. Size of each variable: $area_{v_{k+1}}, \ldots, area_{v_{|allvars|}}$
3. Size limit defined as the remaining space in the scratchpad:
   $scratchpad\_size \; - \; \sum_{v \in allocation(m)} area_v$
4. Bound on the maximum WCET reduction due to allocation of each variable:
   $bound_{v_{k+1}}, \ldots, bound_{v_{|allvars|}}$
   For a variable v, $bound_v$ is defined as the maximum contribution of v towards the execution time of *any* path. Clearly, the WCET reduction achieved by allocating v to scratchpad memory should be bounded by $bound_v$. These bounds can be estimated once and for all through a single traversal of the control flow graph.

The 0-1 knapsack problem allocates some of the variables $v_{k+1}, \ldots, v_{|allvars|}$ to the remaining scratchpad memory space so as to maximize the potential reduction in WCET. Note that the knapsack problem formulation simply computes the heuristic function (not the actual allocation gain) for the purpose of pruning the branch-and-bound search space.

Figure 6.2 illustrates the branch-and-bound search process. Suppose that at one point of the search we have constructed a complete allocation at the leaf node m, which achieves the maximum reduction in WCET among all complete allocations encountered so far.

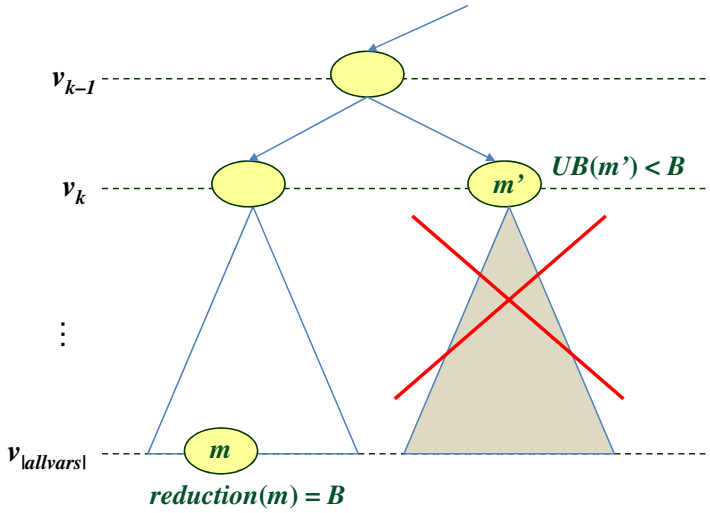


Figure 6.2: Pruning in the branch-and-bound search tree

We remember $reduction(m)$ as the bound B — representing the maximum WCET reduction achieved so far. Suppose later in the search we reach node m' at level k. Using the heuristic function described above, we calculate $UB(m')$, an upper bound on the reduction in WCET achieved by extending the partial allocation at m' to a complete allocation. At this point supposed we find that $UB(m') < B$ which means that any complete allocation we may construct by continuing from m' will never outperform the allocation we have constructed at m. Clearly, in this situation the subtree rooted at m' need not be explored further and can be pruned from the search tree.

In order to achieve effective pruning of the unexplored nodes, the variables are sorted so that a variable that can potentially reduce the WCET more is explored earlier in the search tree. In other words, we measure the potential WCET reduction of a variable v using its maximum contribution over all execution paths, namely $bound_v$. The ordering is simply a decreasing order of $bound_v$ for $v \in allvars$.

The branch-and-bound formulation as described above yields an optimal solution for global WCET optimization. Unfortunately, its complexity is exponential with respect to the number of data variables to be allocated. As such, it is not practical to run the branch-

and-bound search to generate scratchpad allocation from among a large number of data variables unless the scratchpad size is relatively small, in which case the scratchpad capacity constraint may prune out a large portion of the search tree.

6.3.2 Greedy Heuristic

Since the branch-and-bound search is inefficient in running time, we also develop and use a fast heuristic search based on greedy approaches. This search algorithm, in general, may yield a sub-optimal allocation. Nevertheless, our experiments have found that the WCET reduction from the resulting allocation is close to the WCET reduction from the optimal allocation found by branch-and-bound search.

```
1  allocation := ∅;
2  capacity := scratchpad_size;
3  changed := TRUE;
4  Perform WCET analysis to obtain worst-case path π;
5  while capacity > 0 AND changed = TRUE do
6      changed := FALSE;
7      V := { v | v is an unallocated variable accessed in π, area_v ≤ capacity };
8      if V ≠ ∅ then
9          Find v ∈ V with maximum contribution towards the execution time of π;
10         allocation := allocation ∪ {v};
11         capacity := capacity − area_v;
12         changed := TRUE;
13         Perform WCET analysis to compute the new worst-case path π;
14 Return allocation;
```

Algorithm 3: Greedy heuristic for scratchpad allocation to reduce WCET of a program

In our heuristic search, we first find the heaviest path taking into account infeasible path information; let this path be π. The heuristic then allocates one program variable v appearing in π. Naturally, the variable selected is the one with maximum contribution to the execution time of π. Following this, we run the WCET analysis once again to find the heaviest path after allocating v. More variables can then be allocated for this path.

Of course, we stop whenever the scratchpad is filled. The scratchpad is considered as filled (for π) when none of the variables accessed in π can be added into the existing allocation without exceeding the scratchpad size. The skeleton of the greedy heuristic appears in Algorithm 3.

Further Heuristic Consideration We observe that the sub-optimality of the greedy heuristic arises from over-optimization of the first few heaviest paths, so that the scratchpad space is exhausted by the time we get to another relatively heavy path. We thus attempt a more complicated heuristic which balances the allocation among the competing paths by allowing backtracking in allocation. We allow backtracking when the scratchpad is filled, by removing some variables from the existing allocation to make space for variables in the current heaviest path. To guard against unbounded backtracking, we require that the new worst-case path after the replacement is not the same as any of the previously encountered WCET paths. However, this complicated heuristic does not always produce a better reduction in WCET than the greedy heuristic. It is better than the greedy heuristic when we have multiple competing paths with only a few overlapping variables, and the scratchpad size is very small (hence it gets filled up quickly). In our experiments, we found that such a situation occurs rarely. Moreover, allowing for backtracking adds an overhead to the running time of the optimization. Thus, we consider the greedy heuristic to perform better than the complicated heuristic.

6.4 Experimental Evaluation

Setup We choose six data/control intensive kernels as benchmarks. The characteristics of these benchmarks are given in Table 6.1. `lingua` performs language-independent text processing [30]. `statemate` and `compress` are benchmarks taken from [137]; `statemate` is a car window lift controller generated automatically from a statechart

specification, while `compress` is a data compression program. The `susan` benchmark is taken from MiBench's automotive application suite [44]; it is a kernel performing edge thinning in an image. Finally, `des` performs Data Encryption Standard, and `fresnel` computes Fresnel integrals. Both are taken from [103].

Table 6.1: Benchmark characteristics

Benchmark	Data Memory (bytes)			WCET (cycles)	
	Scalars	Arrays	Total	with infeasibility	w/o infeasibility
`lingua`	141	340	481	823,305	825,227
`statemate`	163	64	227	41,578	44,938
`susan`	96	36,136	36,232	293,989,241	485,328,185
`compress`	157	263,849	264,006	319,075	390,937
`des`	208	1,153	1,361	643,270	643,894
`fresnel`	536	0	536	256,172	256,172

Most of our benchmarks are compute-intensive kernels processing one or more arrays. This is evident from Table 6.1 that shows the total data memory size and its division between scalar and array variables. Also, most of our benchmarks have limited numbers of possible paths through any loop iteration; the only exception in this regard is `statemate`. The `statemate` benchmark is a control-intensive application with very little data manipulation. This benchmark has a large number of possible paths (6.55×10^{16}) for one loop iteration. However, a rough estimate shows that a large number of these paths are infeasible. The number of feasible paths for any loop iteration is 1.09×10^{13}, that is, less than 0.016% of the total number of possible paths. Table 6.1 shows the estimated WCET of all benchmarks both with and without infeasible path information. This estimation assumes that data variables have *not yet* been allocated to scratchpad. The heaviest path in `fresnel` is feasible, so estimation with and without infeasible paths produce the same WCET.

We use SimpleScalar tool set [12] for the experiments. The programs are compiled using gcc 2.7.2.3 targeted for SimpleScalar. As our focus is to evaluate the latency reduction due to allocation of data variables to scratchpad memory, we assume a simple embedded

processor with single-issue in-order pipeline and perfect branch prediction. Instructions are accessed from off-chip memory through a perfect instruction cache with 1 clock cycle latency. There is no data cache; a subset of data variables can be allocated to on-chip scratchpad memory. We assume single-cycle scratchpad access latency as can be expected from current technology. Considering the presence of large-scale benchmarks, the main memory access latency is fixed at 10 cycles (real values may range from 10 to hundreds).

We apply the WCET analysis (as described in Chapter 4) on each compiled program to calculate its WCET. We then run all the three different scratchpad allocation techniques (ILP, branch-and-bound, and greedy heuristic) for each benchmark to obtain the corresponding allocation and the reduced WCET. All the experiments have been performed on a 3.0GHz P4 CPU with 1MB cache and 2GB memory.

Discussion Figure 6.3 shows the original and reduced WCET due to scratchpad allocation by the ILP, branch-and-bound and greedy heuristic methods. The original WCET assumes that all variables are allocated in off-chip memory, that is, there is no scratchpad memory. This is quite common for current real-time systems. The reduced WCET is the estimation returned by the different techniques after scratchpad allocation. In Figure 6.3, the reduced WCET is indicated by the yellow bars; the difference between the original WCET and reduced WCET is indicated by the green bars stacked on top of the yellow bars. Thus the total height of each bar (green + yellow) indicates the original WCET.

We choose three different scratchpad memory sizes for each benchmark corresponding to 5%, 10%, and 20% of the total data memory size (shown in Table 6.1). As expected, the WCET reduces by $5 - 80\%$ due to allocation for the different benchmarks. As we increase the scratchpad size from 5% to 10%, there is no reduction in WCET for `compress`, `lingua` and `susan`. This is because these benchmarks have some large

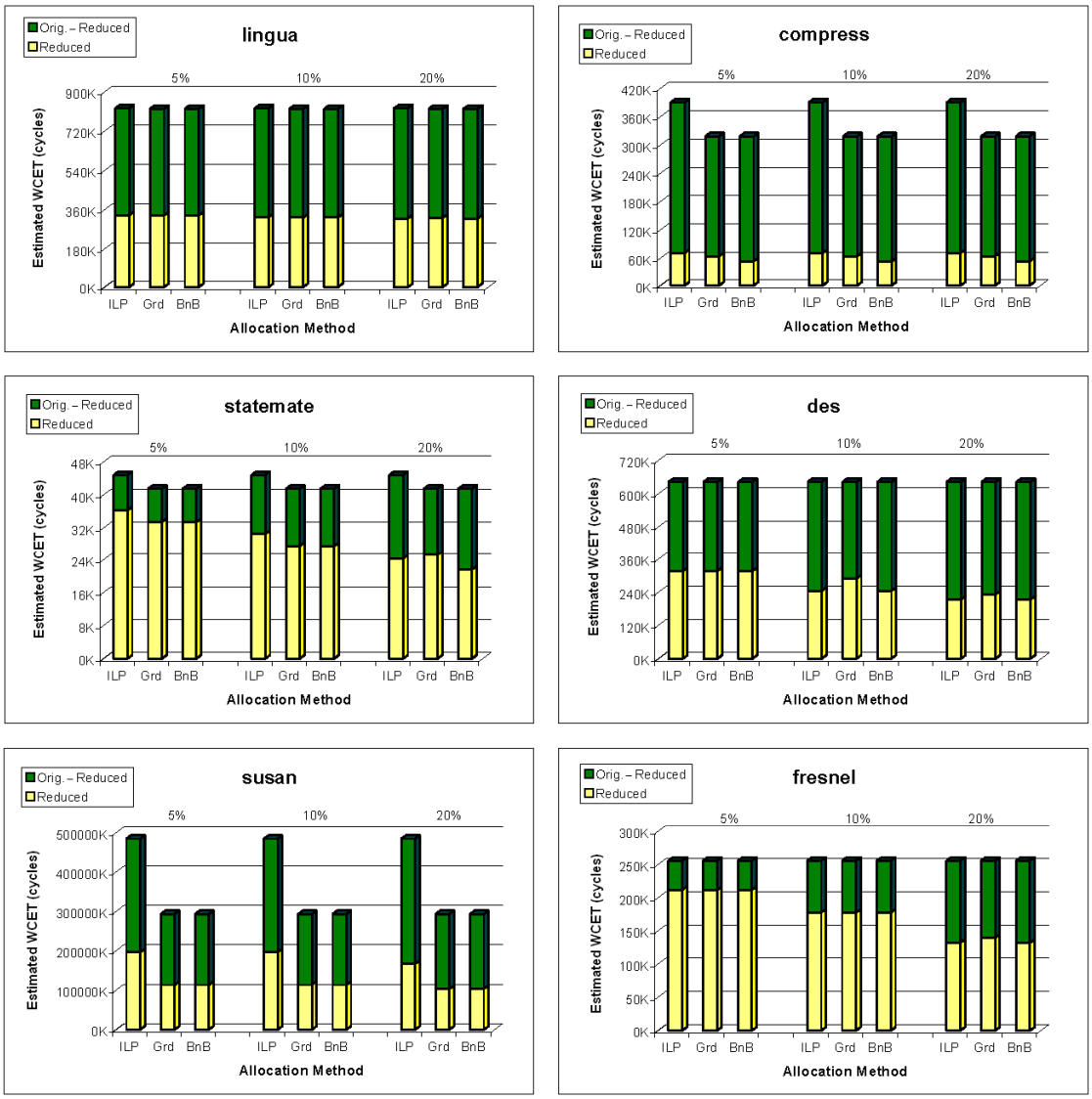


Figure 6.3: Original and reduced WCET after scratchpad allocation by *ILP*, *greedy (Grd)*, and *branch-and-bound (BnB)* for various benchmarks and scratchpad sizes

arrays and increasing the scratchpad size still cannot accommodate these arrays. In general, allocating *only* 10% of the data memory to scratchpad has been able to achieve quite a significant reduction in WCET for all benchmarks.

Notice that the reduced WCET obtained via ILP is typically higher than the reduced WCET with branch-and-bound and/or greedy heuristics. This is because the ILP-based method cannot take into account the detailed infeasibility information as exploited by our efficient WCET calculation method. This result shows that it is important to take the infeasibility information into account when analyzing and optimizing for WCET. In most cases there is very little or no difference between greedy heuristic and branch-and-bound implying that greedy heuristic achieves near-optimal solutions. For the benchmark `statemate` with scratchpad size equal to 20% of the data memory, the reduced WCET with ILP is slightly better than the reduced WCET with greedy approach. Even though the greedy approach takes infeasibility information into account, it is still sub-optimal; in this particular case, it performs worse than ILP does.

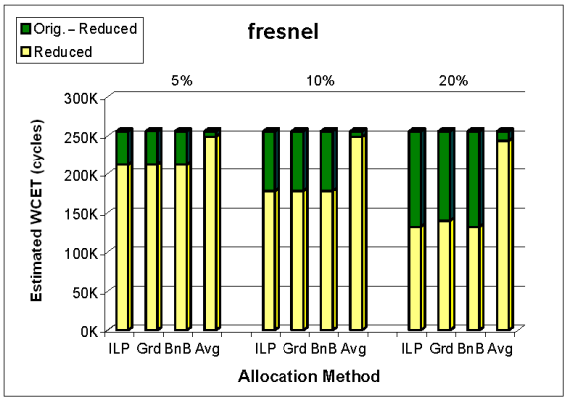


Figure 6.4: Original and reduced WCET after *ILP*, *greedy (Grd)*, *branch-and-bound (BnB)*, and *ACET-based (Avg)* scratchpad allocation for the `fresnel` benchmark

Finally, as mentioned earlier, existing scratchpad allocation techniques use the average-case profile information. In particular, Wehmeyer and Marwedel [139] investigate the effect of scratchpad allocation on WCET where average-case profile is used for determining the allocation. However, the optimal allocation for average-case may not be

optimal in reducing the worst-case execution time. This is shown experimentally in Figure 6.4. Here, we have collected average-case data access frequencies by running the benchmarks with representative set of inputs. We then formulate a 0-1 knapsack problem to find the allocation that optimizes the average-case execution time (ACET), and compute the reduction in WCET using this allocation. This appears as *'Avg'* in the figure. We plot it against the reduction in WCET using our WCET-guided allocation techniques — ILP, branch-and-bound, and greedy methods.

In the `fresnel` benchmark, our WCET-guided allocation methods, via branch-and-bound and greedy heuristics, produce up to 46% reduction in WCET as compared to the allocation produced by the ACET-based technique. Other benchmarks show similar trends but less pronounced WCET reduction. For example, in the `lingua` benchmark our WCET-guided greedy allocation strategy produces up to 22% WCET reduction as compared to the allocation produced by the ACET-based technique.

On the other hand, when we observe the effect of our WCET-guided allocation methods on ACET in this set of benchmarks, we find that our ILP and branch-and-bound methods achieve similar reduction in ACET as compared to ACET-guided allocation methods.

Table 6.2: Running time of allocation methods for scratchpad = 10% of data memory

Benchmark	Runtime (ms)			
	ILP		Greedy	Branch-and-bound
	Formulation	Solution		
`lingua`	3	28	16	78
`statemate`	4	33	12,080	36,616
`susan`	3	15	18	23,960
`compress`	3	18	15	346,740
`des`	3	18	5	19
`fresnel`	3	16	1	6

Table 6.2 shows the running times of our allocation techniques when the scratchpad memory size is equal to 10% of the data memory size. It is interesting to note that even though the reduced WCET with ILP method is typically greater than the reduced WCET

with greedy heuristic, the running time of the greedy method is comparable to or even less than the running time of the ILP method for all benchmarks except `statemate`. For `statemate`, the running time of the greedy method is substantially more than that of the ILP method. This particular benchmark has a large number of program paths, hence it is more time consuming to estimate the WCET by taking infeasibility information into account. ILP-based allocation does not take infeasibility information into account and is therefore much faster.

6.5 Chapter Summary

In this chapter, we have discussed scratchpad memory allocation for data variables with the explicit goal of reducing the WCET of a sequential program. We have presented both optimal and heuristic allocation techniques. The major difference between our work and existing works is that we specifically target WCET reduction instead of using the WCET path (which changes as we fix the allocation) or the ACET path as profiles.

Chapter 7

Scratchpad Allocation for Concurrent Applications

The majority of current generation embedded applications are inherently concurrent in nature: they are built from multiple components that run independently with occasional interactions with each other to accomplish their functionality. This is true for some real-time applications as well, for instance those in automotive and avionics domain. The combination of concurrency and real-time constraints introduces significant challenges to the memory allocation problem.

In this chapter, we address the problem of scratchpad memory allocation for concurrent embedded software with real-time constraints running on uniprocessor or multiprocessor platforms. Our objective is to reduce the *worst-case response time (WCRT)* of the entire application.

7.1 Introduction

In the previous chapter, we have looked at the problem of scratchpad memory allocation for sequential applications. This approach is not directly applicable to concurrent applications with multiple interacting processes. Let us illustrate the issues involved with an example.

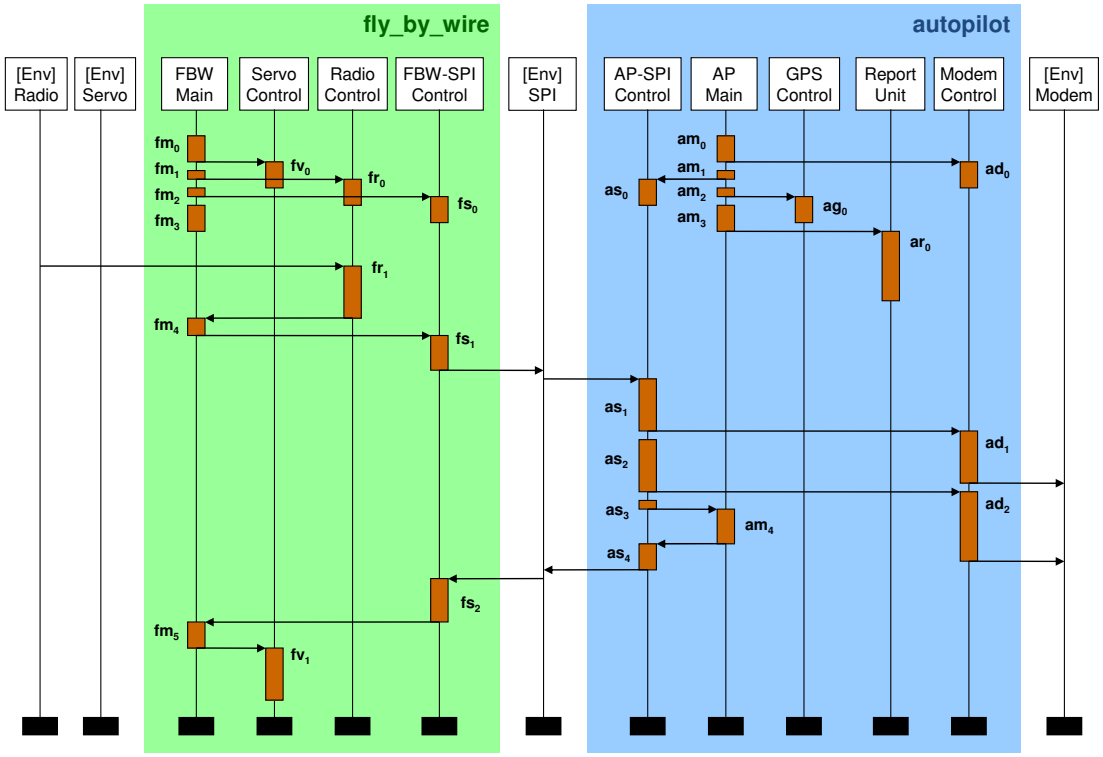


Figure 7.1: Message Sequence Chart model of the adapted UAV control application

Figure 7.1 shows a Message Sequence Chart (MSC) model [4, 56] depicting the interaction among the processes in an embedded application. We use an MSC model as it provides a visual but formal mechanism to capture inter-process interactions. Visually, an MSC consists of a number of interacting *processes*, each shown as a vertical line. Time flows from top to bottom along each process. A process in turn consists of one or more *tasks* represented as blocks along the vertical line. Message communications between the processes are shown as horizontal or downward sloping arrows.

Semantically, an MSC denotes a labeled partial order of tasks. This partial order is the transitive closure of (1) the total order of the tasks in each process, and (2) the ordering imposed by message communications — a message is received after it is sent.

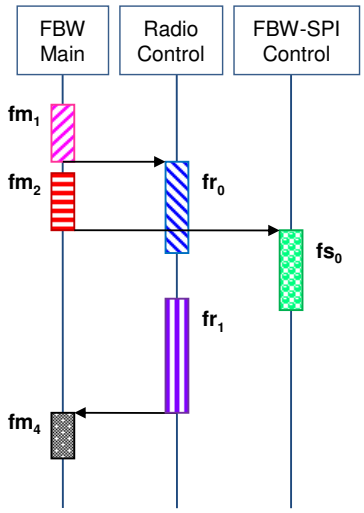


Figure 7.2: A sample MSC extracted from the UAV control application case study

Consider the MSC example in Figure 7.2, which has been extracted from the earlier application. A naive memory allocation strategy can be to share the scratchpad memory among all the tasks of all the processes throughout the lifetime of the application. This is illustrated in Figure 7.3a, where the distribution of scratchpad space over tasks is depicted in the horizontal direction, while the content reloading over time is depicted in order from top to bottom. Allocation algorithms proposed in the literature for sequential applications can be easily adapted to support this strategy. However, this strategy is clearly sub-optimal, as a task executes for only a fraction of the application's lifetime yet occupies its share of the memory space for the entire lifetime of the application. Instead, two tasks with disjoint lifetimes (*e.g.*, tasks fm_1 and fm_2) should be able to use the same memory space through time multiplexing. This is known as *dynamic scratchpad allocation* or *scratchpad overlay*, where the scratchpad memory content can be replaced and reloaded at runtime.

At the other extreme of this approach, we can also let each task occupy the whole scratchpad while it is executing (Figure 7.3b). When a task is preempted, its corresponding memory content in the scratchpad is replaced by that of the preempting task. Certainly, the time taken to load and reload contents into the scratchpad memory following each preemption and resuming of task execution also adds to the total latency experienced by the system. This too has to be bounded to provide timing guarantee. In a system with a large number of tasks vying for CPU time, the chain of preemptions can get arbitrarily long and difficult to analyse.

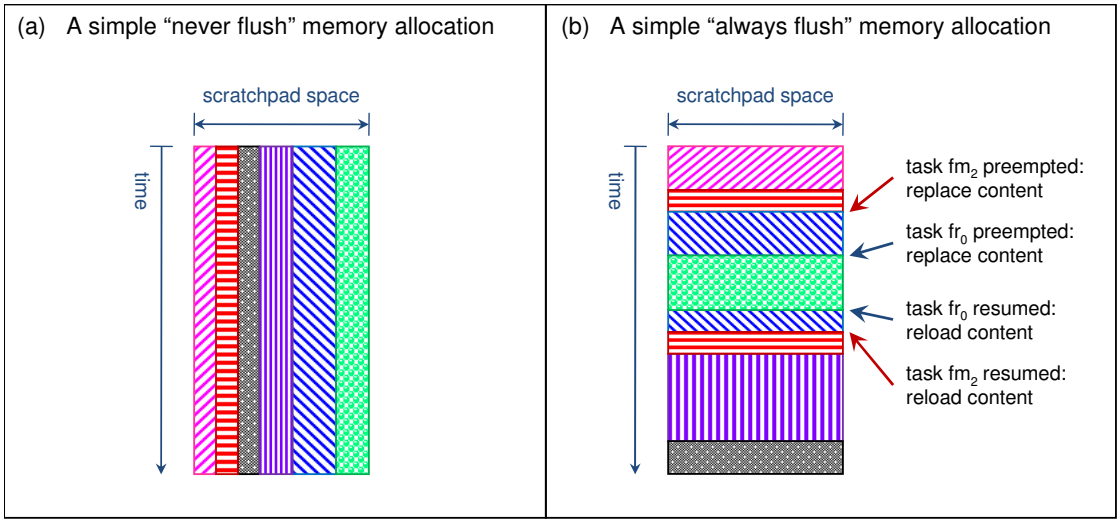


Figure 7.3: Naive memory allocation strategies for the model in Figure 7.2

The key to our proposed technique is finding a balance between these two extremes. Clearly, we would like to employ scratchpad overlay as much as possible for optimal gain in application response time. However, as timing predictability is the main motivation behind the choice of scratchpad memory over caches, it should be maintained even in the presence of scratchpad overlay. This implies that in a concurrent system (*e.g.*, as shown in Figure 7.1), *two tasks should be mapped to the same memory space only if we can guarantee that they have disjoint lifetimes*. Otherwise, the task with higher priority may preempt the other, leading to scratchpad reloading delay at every preemption point.

We can trivially identify certain tasks with disjoint lifetimes based on the partial order of an MSC; for example, as task fm_1 "happens before" task fm_2, clearly fm_1 and fm_2 have disjoint lifetimes (Figure 7.2). However, there may exist many pairs of tasks that are incomparable as per MSC partial order but still have disjoint lifetimes. For instance, the timing analysis may be able to determine that the execution time of fr_0, given any input, is always long enough to ensure that the succeeding task fr_1 can never be started before fm_2 finishes executing. In this case, we will be able to employ a better scratchpad overlay scheme as illustrated in Figure 7.4a, where fm_1, fm_2, fm_4 and fr_1 are mapped to the same scratchpad space which they can utilize during their respective lifetimes without disrupting one another. On the other hand, we should not arrive at a decision such as the one in Figure 7.4b, which lets fr_0 and fs_0 share the same scratchpad space without any guarantee that fs_0 will not preempt fr_0 in the actual execution. This situation will lead to unexpected reloading delays that will invalidate the WCRT estimation.

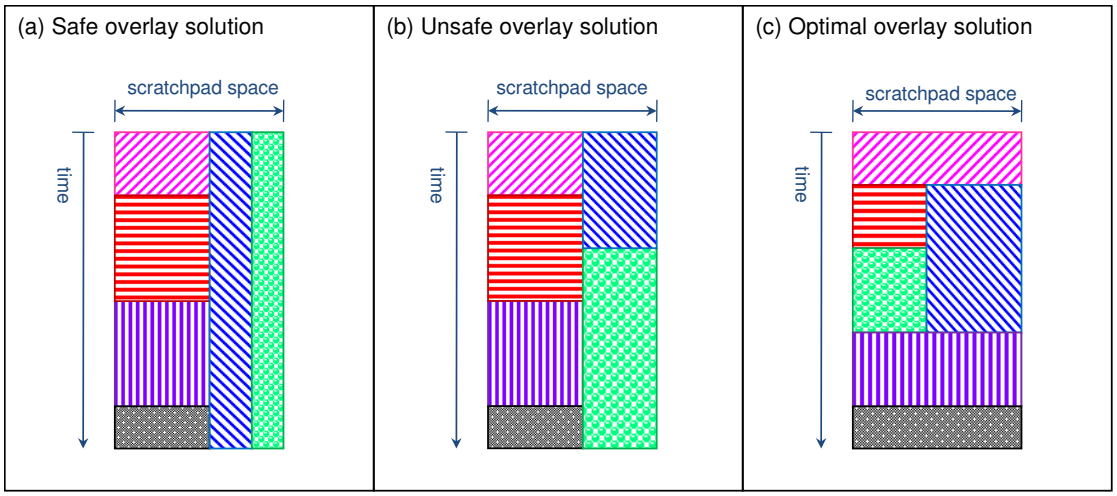


Figure 7.4: Choices of scratchpad overlay schemes for the model in Figure 7.2: (a) safe, (b) unsafe, and (c) optimal

We have found out through experiments that the optimal WCRT reduction for this particular example is achieved by the memory allocation scheme in Figure 7.4c, which

requires a deeper analysis of process interaction to arrive at. Moreover, as scratchpad allocation reduces execution times of the individual tasks, the lifetimes of the tasks and thus their interaction pattern may change. Therefore, an effective scratchpad allocation scheme attempting to minimize the WCRT of the application should consider process interferences as well as the impact of allocation on process interferences.

We propose an *iterative allocation algorithm* (Figure 7.5) consisting of two critical steps: (1) analyze the MSC along with existing allocation to estimate the lifetimes of tasks and hence the non-interfering tasks, and (2) exploit this interference information to tune scratchpad reloading points and content so as to best improve the WCRT. The iterative nature of our algorithm enables us to handle the mutual dependence between allocation and process interaction. In addition, we ensure monotonic reduction of WCRT in every iteration, so that our allocation algorithm is guaranteed to terminate.

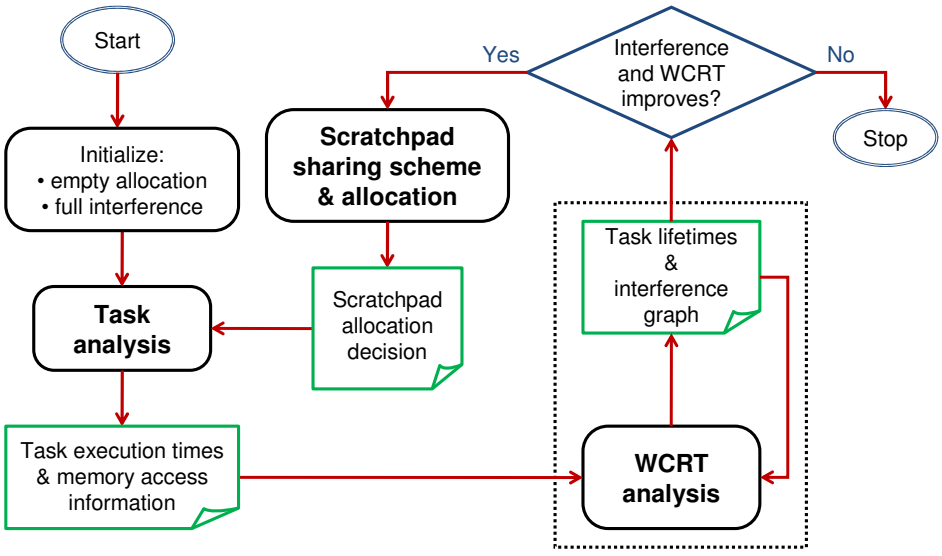


Figure 7.5: Workflow of WCRT-optimizing scratchpad allocation

This work complements the research on *cache-related preempted delay (CRPD) analysis* (see Section 3.1). CPRD analysis provides timing guarantee for concurrent software by analyzing interferences in cache memory due to process interactions. An important point of this analysis is identifying memory blocks of a process that are at risk of getting

replaced by a preempting process, and the additional miss latencies. Our work, on the other hand, eliminates interference in memory through scratchpad allocation.

7.2 Problem Formulation

7.2.1 Application Model

The input to our problem is in the form of Message Sequence Chart (MSC) [4, 56] that captures process interactions corresponding to a concurrent embedded application. We assume a *preemptive, multitasking* execution model. The application is periodic in nature. The MSC represents interactions within one such invocation where all processes involved should adhere to a common period and deadline. The underlying hardware platform contains one or more processing elements (PEs), each associated with a private scratchpad memory. Figure 7.6 shows a simple example that illustrates this setting.

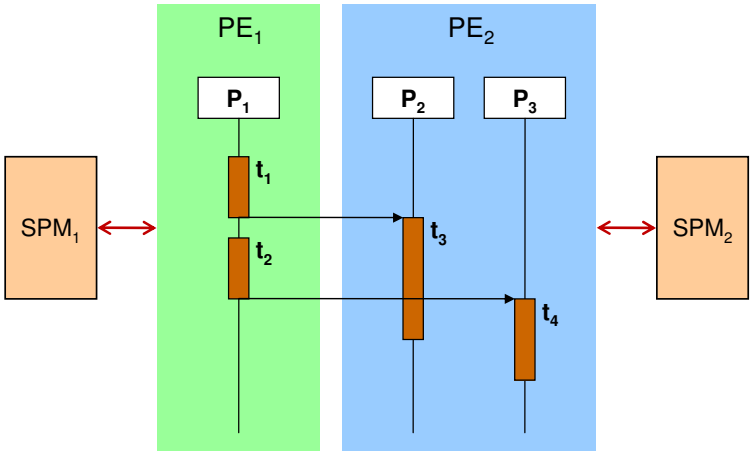


Figure 7.6: A simple MSC running on multiple PEs with scratchpad memories

A vertical line in the MSC represents the lifeline of a process, that is, the time period during which the process is alive. A process may consist of more than one tasks. A process typically corresponds to a specific functionality, and it is thus natural to assign

all the tasks in a process to one PE. The order in which the tasks appear on the process lifeline in the MSC reflects their order of execution on the PE. In Figure 7.6, tasks t_1 and t_2 belong to the same process P_1 scheduled on PE_1, and t_2 executes after t_1 completes execution on the same PE. Dependencies across processes are represented as horizontal arrows from the end of the *predecessor* task to the start of the *successor* task. In our example, the communication delay between processes are zero. Including non-zero communication delay in the analysis is straightforward: the start of the successor task is simply pushed back by the amount of the delay.

Execution Priority Each process in the application is assigned a unique static priority. The priority of a task is equal to the priority of the process it belongs to. If more than one processes are assigned to the same PE, then a task executing on that PE may get preempted by a task from another process with higher priority if their lifetimes overlap. The assignment of static priorities to processes and the mapping of processes to PEs are inputs to our framework.

Note that static priority assignment alone does not guarantee a fixed execution schedule at runtime. The preemptions and execution time variations depending on input lead to varying completion times of a task. This, in turn, gives rise to different execution schedules. In Figure 7.6, supposing process P_3 has higher priority than P_2 on PE_2, then task t_3 will be preempted when task t_4 becomes ready, *if* the execution times of the tasks in that particular invocation are such that t_3 has not completed execution when task t_2 completes on PE_1. We see that this situation is determined not only by the tasks involved in the preemption, but other tasks in the system with dependency relationship with these tasks as well.

The analysis and discussion in the rest of this paper will be at the task level instead of the process level, as we make allocation decision for each task individually. Formally,

let $t_1, \ldots, t_N$ denote the tasks belonging to all the processes in the application. Each task t_i $(1 \leq i \leq N)$ is associated with:

1. a period $pd(t_i)$,
2. a static priority $pr(t_i)$ in the range $[1, R]$ with 1 being the highest priority, and
3. mapping to a PE $PE(t_i)$ in the range $[1, Q]$, where Q is the number of PEs in the system.

As mentioned earlier, all the tasks belonging to a process have the same priority and are mapped to the same PE.

7.2.2 Response Time

We use the term *task lifetime* to mean the interval from the time a task is started and the time it completes execution, specified in absolute time values. The absolute time that a task can be started depends on the completion times of its predecessors, according to the dependency specified in the MSC model. The length of a task's lifetime is its *response time*, which consists of the time it takes to perform computation (without interruption) and the delay it experiences due to preemptions by higher priority tasks.

In general, the computation time of a task may vary due to (1) the variation in input data that triggers different execution paths within the program, and (2) the variation in memory access latencies (whether the accessed code/data object is in scratchpad or main memory). The uninterrupted computation time required by each individual task can be determined via static analysis [76] of the program corresponding to a task, and represented as a range between its best-case execution time (BCET) and worst-case execution time (WCET) [87]. However, this estimation is intertwined with the second

component, memory access latencies, as we will elaborate when we discuss scratchpad allocation later in this section.

Obviously, the longer the execution time of a task, the longer its response time. However, the impact of a task's response time on *other tasks' response times (and thus overall application response time)* is not straightforward. In the example shown in Figure 7.6, a longer t_2 execution time will cause t_4 to start later and not preempt t_3 on the same PE (supposing t_4 has higher priority than t_3). In this scenario, the response time of t_3 becomes shorter, and this possibly leads to an earlier completion time of the overall application. As our ultimate aim is to optimize for the *worst-case response time (WCRT) of the whole application*, we need to take into account the interplay among these components.

7.2.3 Scratchpad Allocation

The term *scratchpad allocation* in this context consists of two components: (1) the distribution of scratchpad space among the application tasks, and (2) the selection of memory blocks of each task to fit into the allocated space. When a task can access part of its memory requirement from the scratchpad memory instead of the significantly slower main memory, its execution time can reduce dramatically. Depending on the portion of memory address space allocated to the scratchpad memory, the time taken by a single execution path within the task itself may vary greatly. Therefore, in the presence of scratchpad memory, the execution path and execution time of a task in the best or worst case should be determined by taking into account the allocation decision.

Here, we consider allocating program codes of the tasks into the scratchpad. The method applies similarly to data allocation. To make better use of the limited scratchpad space, we allow scratchpad overlay, that is, the same scratchpad memory space can be allocated to two or more tasks as long as they have disjoint lifetimes. As apparent from Figure 7.6,

each PE accesses its own scratchpad, and the space will be shared among tasks executing on the corresponding PE only; we do not consider accesses to remote scratchpad which will be effective only if the PEs have access to fast interconnection network [63].

Formally, let $\mathcal{S}$ be a particular scratchpad allocation for the application. As described earlier, $\mathcal{S}$ consists of two components:

1. the amount of scratchpad space $space(t_i)$ allocated to each task $t_i, 1 \leq i \leq N$, and

2. the allocation of $space(t_i)$ among the code blocks of t_i.

By virtue of scratchpad overlay, the sum of the scratchpad space allocated to all tasks, $\sum_i^N space(t_i)$, is not necessarily less than or equal to the total available scratchpad space. This will be clear in the description that follows.

Let $Mem(t_i)$ denote the set of all code blocks of t_i available for allocation. Given $space(t_i)$ assigned in $\mathcal{S}$, the allocation $Alloc(t_i, \mathcal{S}) \subseteq Mem(t_i)$ is the set of most profitable code blocks from t_i to fit the capacity. The BCET and WCET of t_i as a result of allocation $\mathcal{S}$ are denoted as $bcet(t_i, \mathcal{S})$ and $wcet(t_i, \mathcal{S})$ respectively. Given an allocation $\mathcal{S}$ and the corresponding BCET, WCET of the tasks, we can estimate the lifetime of each task t_i, defined as the interval between the lower bound on its start time, $EarliestSt(t_i, \mathcal{S})$, and the upper bound on its completion time, $LatestFin(t_i, \mathcal{S})$. This estimation should take into account the dependencies among the tasks specified in the model (total order among the tasks within a process, and the ordering imposed by message communication) as well as preemptions.

The WCRT of the whole application is now given by

$$WCRT = max_{1 \leq i \leq N}\ LatestFin(t_i, \mathcal{S}) - min_{1 \leq i \leq N}\ EarliestSt(t_i, \mathcal{S}) \qquad (7.1)$$

that is, the duration from the earliest start time of any task in the application until the latest completion time of any task in the application.

Allocation Constraints Our goal is to construct the scratchpad allocation $\mathcal{S}$ that minimizes the WCRT of the application (Equation 7.1). To achieve this objective, $\mathcal{S}$ should employ optimal inter-task overlay while respecting the constraint that only tasks that do not preempt each other can occupy the same scratchpad space over time. This constraint is formalized as follows.

Suppose the WCRT analysis identifies a set of tasks $G_1^{s,e} \subseteq \{t_1, t_2, \ldots, t_N\}$ having disjoint lifetimes within the time interval $[s, e]$. The tasks in $G_1^{s,e}$ are not necessarily related via dependency; the situation may also arise if the times of dispatch are well separated so that one task invariably completes before another is ready. An example of such a set is $\{fm_1, fm_2, fr_1, fm_4\}$ in Figure 7.2, with s being the time when fm_1 becomes ready, and e being the time when fm_4 completes. Formally:

$$\begin{aligned} \forall\, t_i \in G_x^{s,e} \qquad & s \leq EarliestSt(t_i, \mathcal{S}) < LatestFin(t_i, \mathcal{S}) \leq e \\ \forall\, t_i, t_j \in G_x^{s,e} \qquad & EarliestSt(t_j, \mathcal{S}) > LatestFin(t_i, \mathcal{S}) \\ & \vee\ EarliestSt(t_i, \mathcal{S}) > LatestFin(t_j, \mathcal{S}) \end{aligned} \tag{7.2}$$

If the above conditions are satisfied, then we allow scratchpad overlay by mapping all tasks in $G_1^{s,e}$ to the same scratchpad space throughout the duration of $[s, e]$. Each task t_i in $G_1^{s,e}$ can make use of the whole portion allocated to $G_1^{s,e}$ during its lifetime.

$$\forall\, t_i \in G_x^{s,e}\ \ space(t_i) = space(G_x^{s,e})$$

Other tasks that are active within the same interval $[s, e]$ but may interfere with any task in $G_1^{s,e}$ have to share the scratchpad space with $G_1^{s,e}$ during this interval. We may identify

further groupings among those tasks ($G_2^{s,e}$, $G_3^{s,e}$, and so on), each of which satisfies the same set of constraints given by Equation 7.2. In this perspective, the total scratchpad space cap is distributed among the sets of tasks $G_x^{s,e},\ x : 1, 2, \ldots$ (instead of among the individual tasks) for the duration of $[s, e]$, because each set utilizes scratchpad overlay that allows them to occupy the same space during different times.

$$\sum_x space(G_x^{s,e}) \leq cap$$

Clearly, the challenge lies in defining the grouping of tasks $G_x^{s,e}$ as well as the interval division $[s, e]$ in order to maximize overlay opportunities over the entire application. Our earlier motivating example in Figure 7.4c illustrates a situation where the application execution has been "divided" over four intervals, some of which have only one task active within its duration. In the elaboration of our allocation scheme in the rest of this chapter, we will show how such beneficial situations can be induced through careful manipulations.

7.3 Method Overview

Our proposed method for scratchpad allocation is an iterative scheme (Figure 7.5). Analysis is performed on each task to determine the bounds on its execution time, given the initially empty scratchpad allocation. The memory access information of the task is also produced as a by-product of this analysis, to be used in the later steps. The WCRT analysis then takes in the execution time values and computes the lifetimes of all tasks. An inter-task *interference graph* is then constructed. Figure 7.7a shows task lifetimes computed by the WCRT analysis for our MSC example in Figure 7.2, along with the constructed interference graph. An edge between two nodes in the graph implies overlapping lifetimes of the two tasks represented by the nodes.

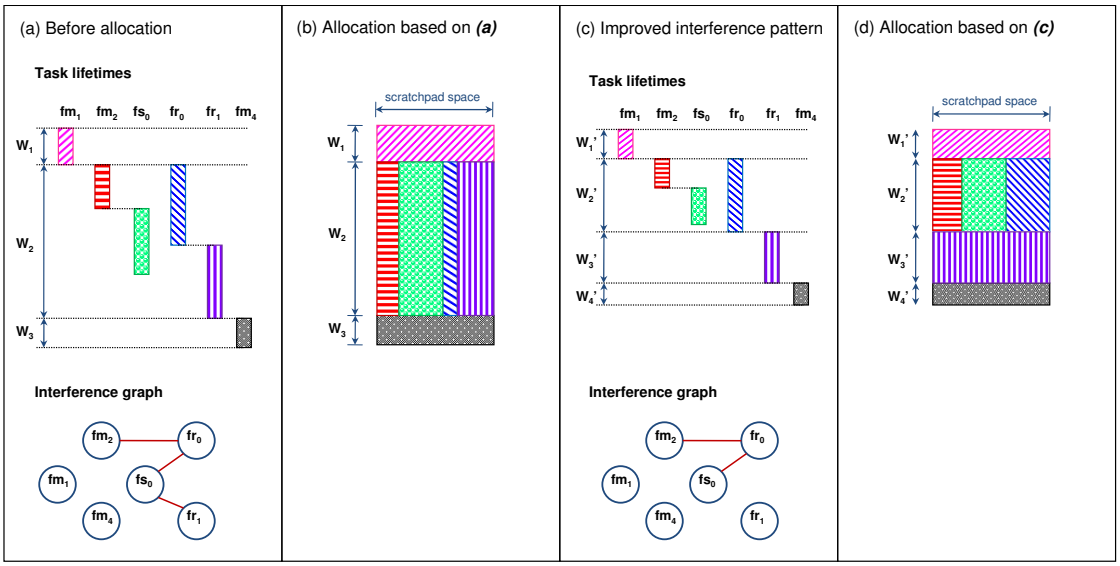


Figure 7.7: Task lifetimes before and after allocation, and the corresponding interference graphs

Based on the analysis result, we can decide on a suitable scratchpad sharing scheme and select actual scratchpad contents for each task, making use of the memory access information from the earlier task analysis. One possible scheme is illustrated in Figure 7.7b, which shows the space sharing among tasks as well as the dynamic overlay over time. With the change in allocation, the execution time of each task is recomputed, and the WCRT analysis is performed once again to update task lifetimes (see Figure 7.5). We ensure at this point that inter-task interference does not worsen. The reason and technique for this will be elaborated in the discussion that follows.

If task interference has been reduced at this point without worsening the application WCRT, we end up with more tasks that are disjoint in their lifetimes (Figure 7.7c). These tasks can now enjoy more scratchpad space through dynamic overlay. If this is the case, we proceed to the next iteration, in which the scratchpad sharing scheme is re-evaluated and the allocation is enhanced (Figure 7.7d); otherwise, the iteration terminates.

Given a finite number of tasks in the application, N, the number of possible task interference pairs is bounded by the number of ways to choose 2 out of N tasks, that

is, $N(N-1)$. As the number of interference relations is finite, by maintaining that inter-task interference does not increase from one iteration to the next, the termination is guaranteed.

We elaborate each of these steps in the following.

7.3.1 Task Analysis

The Task Analysis step determines both the best-case execution time (BCET) and worst-case execution time (WCET) of each task, given a certain scratchpad allocation. These two values bound the range of uninterrupted execution time required to complete the task given all possible inputs.

The timing analysis of the task proceeds as follows. We extract the control flow graph (CFG) of each task. For a given scratchpad allocation, the execution time of each basic block in the CFG can be determined, accounting for the time saving due to faster access time for blocks allocated in the scratchpad. Our static path-based timing analysis method (described in Chapter 4) then traverses the CFG to find the path with the longest (respectively, shortest) execution time. The program may contain loops, which make the CFG cyclic. We require that the bounds on the iteration counts are specified as input to the analysis, so that the execution time can be bounded.

In addition to the BCET and WCET values, the above path-based method also reveals the best-case and worst-case execution *paths* for the given scratchpad allocation. We may then extract the execution frequencies of basic blocks along these paths, which give us the current *gain* of allocating each of the blocks. The *cost* of allocating the block is the area it occupies in the memory space. These two values form the *memory access information* of the task, which will serve as input to the scratchpad content selection in the next iteration of the our technique. In this particular setting, we choose to use

the memory access information corresponding to the worst-case execution path. This information will be used in refining the scratchpad allocation, which will in turn affect the execution time of each basic block, and ultimately affect the best- and worst-case execution paths. Following each such change, the memory access information is also updated. We see here that the allocation step and the task analysis form a feedback loop, which justifies the need for an iterative solution.

7.3.2 WCRT Analysis

As established in the earlier discussion, the response time of a single task has two components: its uninterrupted execution time, and the delay due to preemptions by higher priority tasks. The preemption delay is itself a function of the execution time of the task, as the lifetime of a task affects the way it interacts with other tasks.

The WCRT Analysis step takes the task execution times estimated in the Task Analysis step and the dependencies specified in the MSC model as input. Based on these, it determines the lifetime of each task t_i, which ranges from the time when t_i may start execution until the time when t_i may complete execution, represented by the four values $EarliestSt(t_i, \mathcal{S})$, $LatestSt(t_i, \mathcal{S})$, $EarliestFin(t_i, \mathcal{S})$, and $LatestFin(t_i, \mathcal{S})$. The WCRT of the application given scratchpad allocation $\mathcal{S}$, as formulated in Equation 7.1, is determined via a fixed-point computation that updates these values for each task t_i as more interaction information becomes available throughout the iteration. The change in each iteration is brought on by the variation in the execution time of the individual tasks as the allocation is refined, as well as possible preemption scenarios that arise from the different task lifetimes.

The WCRT analysis method is modified from Yen and Wolf's method [144] and proceeds as follows. A higher priority task can only preempt t_i on the same PE if it is possible to start execution *during* the execution of t_i, that is, after t_i starts and before

t_i finishes. Recall that we denote the priority of task t_i as $pr(t_i)$, with a smaller value translating to a higher priority. The following equation defines the set of such tasks, denoted as $intf(t_i)$.

$$intf(t_i) = \{\ t_j \mid pr(t_j) < pr(t_i)\ \wedge \tag{7.3}$$
$$EarliestSt(t_i, \mathcal{S}) < EarliestSt(t_j, \mathcal{S}) < LatestSt(t_j, \mathcal{S}) < LatestFin(t_i, \mathcal{S})\ \}$$

The WCRT of a single task t_i can then be computed via a fixed-point iteration that finds the root to the equation

$$x = g(x) = wcet(t_i, \mathcal{S}) + \sum_{t_j \in intf(t_i)} wcet(t_j, \mathcal{S}) \times \left\lceil \frac{x}{pd(t_j)} \right\rceil \tag{7.4}$$

The term $wcet(t_i, \mathcal{S})$ refers to the WCET value of t_i given allocation $\mathcal{S}$ determined in the Task Analysis step. The second term in the above equation gives the total preemption delay endured by t_i in the worst scenario. The maximum duration of the delay due to a preempting task $t_j \in intf(t_i)$ is the WCET of t_j, and the maximum number of times t_i may get preempted by t_j is the upper bound of the number of times t_j is activated during t_i's lifetime. The best-case response time (BCRT) of t_i can be computed similarly by substituting the term $wcet(t_i, \mathcal{S})$ with $bcet(t_i, \mathcal{S})$ (that is, the task WCET with the task BCET) and the ceiling operator with the floor operator in the equation.

The fixed-point computation starts by assuming that all higher priority tasks on the same PE can preempt t_i unless they have dependency. The start and finish times of all tasks are computed based on this assumption. After this step, it may become apparent that the lifetimes of certain tasks are well separated, and thus they cannot preempt one another. Given this refined information, the estimation of each task lifetime becomes tighter with each iteration. The iteration stops when there is no further refinement.

The result of the computation gives the WCRT and BCRT of the single task t_i, denoted as $wcrt(t_i, \mathcal{S})$ and $bcrt(t_i, \mathcal{S})$ respectively. The start time and completion time of task t_i are related to these values as follows.

$$EarliestFin(t_i, \mathcal{S}) = EarliestSt(t_i, \mathcal{S}) + bcrt(t_i, \mathcal{S})$$

$$LatestFin(t_i, \mathcal{S}) = LatestSt(t_i, \mathcal{S}) + wcrt(t_i, \mathcal{S})$$

Further, the partial ordering of tasks in the MSC imposes the constraint that task t_i can start execution only after all its predecessors have completed execution, that is

$$EarliestSt(t_i, \mathcal{S}) \geq EarliestFin(t_j, \mathcal{S})$$

$$LatestSt(t_i, \mathcal{S}) \geq LatestFin(t_j, \mathcal{S})$$

for all tasks t_j preceding t_i in the partial order of the MSC.

Observing these constraints, the WCRT analysis computes the lifetimes of all tasks in the application, and determines the application WCRT based on Equation 7.1. Tasks with overlapping lifetimes are said to be interfering, with the higher-priority task possibly preempting the lower-priority task. This interference pattern is captured in a *task interference graph* (Figure 7.7a) for the purpose of scratchpad allocation in the next stage.

7.3.3 Scratchpad Sharing Scheme and Allocation

Given the current interference pattern captured in the interference graph, we decide on an inter-task scratchpad sharing scheme that incurs no unpredictable reloading delay at preemption. We consider four scratchpad sharing schemes with varying sophistication. The simplest scheme performs scratchpad space distribution and allocation without any

regard for the interference pattern (*Profile-based Knapsack*). The second scheme groups tasks based on their lifetime overlap, thus isolating the interference within mutually exclusive time windows and enabling time-multiplexing among the groups (*Interference Clustering*). The third scheme improves this by mapping the allocation problem to a graph coloring problem (*Graph Coloring*). Our final proposal eliminates interferences that compromise tasks on the critical path of the application by inserting strategically placed slacks, in order to improve the situation before applying the allocation scheme (*Critical Path Interference Reduction*). These techniques will be discussed in full details in the next section.

Figure 7.7b and 7.7d visualize possible allocation schemes based on task lifetimes in Figure 7.7a and 7.7c, respectively. They clearly show how task interference pattern influences the allocation decision. An important feature of the allocation is that each task occupies the space assigned to it for the whole duration of its execution, without being preempted by any other task. This ensures that reloading of the scratchpad occurs exactly once for each task activation, and the incurred delay can be tightly bounded.

In each scheme, aside from the scratchpad sharing, the memory content to be allocated in the scratchpad for each task is also selected for optimal WCRT. After the allocation is decided, each task goes through the Task Analysis step once again to determine the updated BCET, WCET, and worst-case memory access information.

7.3.4 Post-Allocation Analysis

Given updated task execution times after allocation, the WCRT analysis is performed once again to compute updated task lifetimes. There is an important constraint to be observed in the WCRT analysis when the allocation decision has been made. The new WCET and BCET values have been computed based on the current scratchpad allocation, which is in turn decided based on the task interference pattern resulting from the

previous analysis. In particular, scratchpad overlays have been decided among tasks determined to be interference-free. Therefore, these values are only valid for the same interference pattern, or for patterns with less interference.

To understand this issue, suppose the interference graph in Figure 7.8a leads to the allocation decision in Figure 7.8b. The reduction in WCET due to the allocation in turn reduces task response times and changes task lifetimes to the one shown in Figure 7.8c. However, this computation of lifetimes is incorrect, because it has assumed the BCET and WCET values of fs_0 given that it can occupy the assigned scratchpad space throughout its execution. If fm_4 is allowed to start earlier, right after its predecessor fr_1 as shown in Figure 7.8c, it may in fact preempt fs_0, flushing the scratchpad content of fs_0 and causing additional delay for reload when fs_0 resumes. Indeed, we see that the interference graph deduced by the WCRT analysis in Figure 7.8d has an added edge from fm_4 to fs_0.

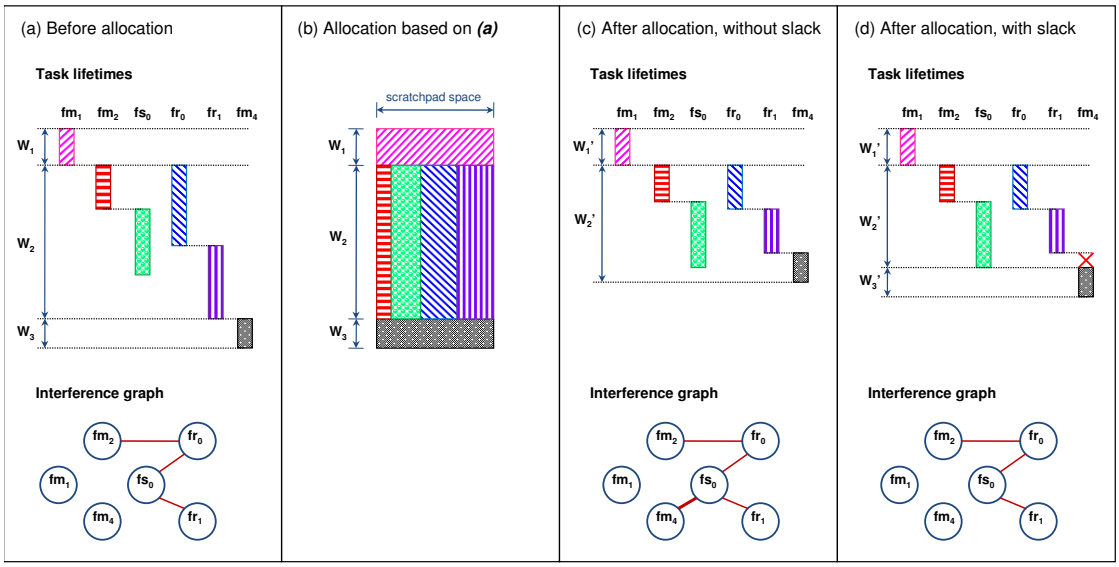


Figure 7.8: Motivation for non-increasing task interference after allocation

To avoid this unsafe assumption, we need to maintain that tasks known *not to interfere* when a certain allocation decision is made *will not become interfering* in the updated lifetimes. This is accomplished by introducing a *slack* that forces the later task to "wait

out" the conflicting time window. The adapted WCRT analysis consults existing interference graph and adjusts the start time of fm_4 such that

$$EarliestSt(fm_4, \mathcal{S}) \geq LatestFin(fs_0, \mathcal{S})$$

The start times of tasks that are dependent on fm_4 are adjusted accordingly. Figure 7.8d shows the adjusted schedule, which maintains the same interference graph as Figure 7.8a by forcing fm_4 to start after fs_0 has completed, thus inserting a slack between the fm_4 and its immediate predecessor fr_1.

With a more sophisticated sharing/allocation scheme and schedule adjustment as we will introduce next, we can sometimes remove existing task interferences without adding interference elsewhere (for example, in a situation depicted in Figure 7.7). When this happens, we iterate over the allocation and analysis steps to enhance current decision, until no more improvement can be made (Figure 7.5). As task interferences are enforced to be non-increasing, the iteration is guaranteed to terminate.

7.4 Allocation Methods

This section describes the scratchpad allocation routine, which is the heart of this chapter. As only one task will be running on the PE at any given time, we can, in theory, utilize the whole scratchpad space for the single executing task (Figure 7.3b). The concern arises when a task is preempted, as flushing the scratchpad content will cause additional reloading delay when the task resumes. In that case, it may be beneficial to reserve a portion of the scratchpad for each of the tasks (*space-sharing*), thus avoiding the need to flush and reload the scratchpad memory at each preemption/resume. On the other hand, two tasks guaranteed to never interfere with each other can share the same space via overlay (*time-sharing*). In Figure 7.7b, for example, tasks fm_2, fs_0, fr_0, and fr_1 in

the interval W_2 are space-sharing tasks, while task fm_1 has a time-sharing relationship with fm_4 and with each of the tasks in the interval W_2.

The various schemes are illustrated in Figure 7.9. The left side of each picture shows task lifetimes as determined by the WCRT analysis, and the right side sketches the state of the scratchpad memory due to the different allocation schemes. In the following, we present the detailed allocation methods, with each solution building on the routines established by the simpler method preceding it.

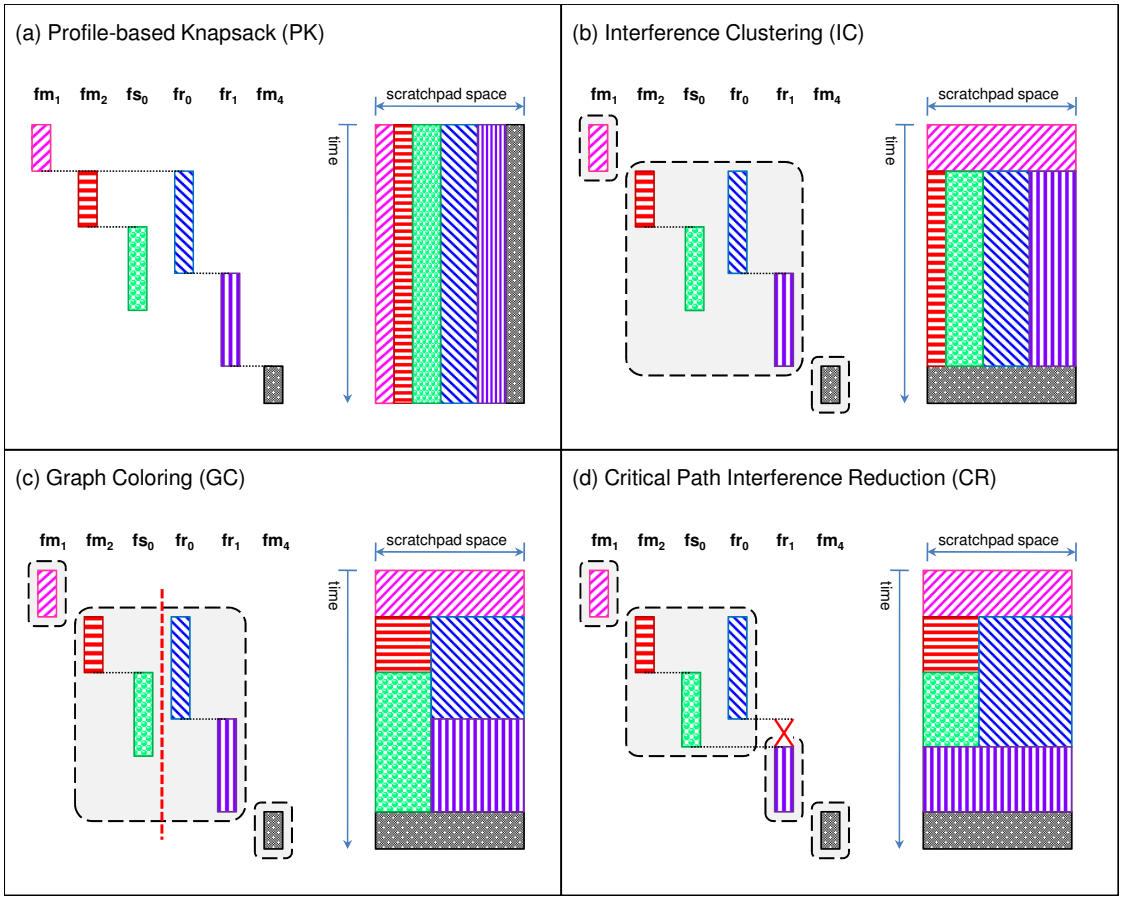


Figure 7.9: Four considered allocation schemes with varying sophistication

7.4.1 Profile-based Knapsack (PK)

As the baseline method, we consider a straightforward static allocation strategy where all tasks executing on the same PE will share the PE's scratchpad space throughout application lifetime. It does not take into account the possible interferences among tasks running on the PE. We refer to this scheme as *Profile-based Knapsack (PK)*, illustrated in Figure 7.9a. The main focus here is the scratchpad content selection routine, which uses the information on sizes (cost) and access frequencies (gain) of code blocks along the worst-case execution path of the tasks. This content selection routine, elaborated below, will also serve as a building block for the other allocation schemes.

As each PE makes use of its own scratchpad space, we can perform allocation for each PE independently. Partitioning and static scratchpad allocation for a PE q can be simultaneously optimized via a 0-1 integer linear programming (ILP) formulation.

Objective Function Our ultimate objective is to minimize the WCRT of the whole application. However, there exists no closed-form linear representation for the definition of the application WCRT given a particular allocation decision (Equation 7.1), whose evaluation requires a fixed-point computation as we have seen in the previous section. Hence, the objective function to find the optimal allocation $\mathcal{S}$ is formulated to *approximate* this definition, as follows.

$$\sum_{t_i : PE(t_i)=q} (F(t_i) + d) \times wcet(t_i, \mathcal{S})$$

Recall that $wcet(t_i, \mathcal{S})$ denotes the WCET of task t_i given scratchpad allocation $\mathcal{S}$. This variable is weighted by the value $F(t_i) + d$, which measures the *potential contribution of the task to the application WCRT*.

The first term, $F(t_i)$, represents the contribution of t_i to the *current* application WCRT, which includes the delay t_i possibly introduces when it preempts any other task. The value of $F(t_i)$ is estimated based on the current WCRT path of the application (critical path) π_0 using the following rules:

$$F(t_i) = \begin{cases} \frac{lifetime(t_i, \mathcal{S}_0)}{\mathcal{W}_0} & \text{if } t_i \in \pi_0; \\ \frac{wcrt(t_i, \mathcal{S}_0)}{\mathcal{W}_0} & \text{if } t_i \text{ may preempt a task in } \pi_0; \\ 0 & \text{otherwise.} \end{cases}$$

The term $\mathcal{S}_0$ denotes the current scratchpad allocation (empty for the first allocation attempt), while the term $\mathcal{W}_0$ denotes the current application WCRT.

If t_i is part of the current WCRT path, then t_i's contribution to the current application WCRT is the ratio of the length of its lifetime to the overall WCRT $\mathcal{W}_0$. Here, the task lifetime $lifetime(t_i, \mathcal{S}_0)$ includes the whole duration when t_i is active, from the earliest start time to the latest completion time:

$$lifetime(t_i, \mathcal{S}_0) = LatestFin(t_i, \mathcal{S}_0) - EarliestSt(t_i, \mathcal{S}_0)$$

If t_i is not on the critical path itself but may preempt a task on that path, then t_i may introduce a preemption delay up to the maximum length of its own response time, $wcrt(t_i, \mathcal{S}_0)$. Recall that the WCRT of an individual task t_i is given by

$$wcrt(t_i, \mathcal{S}_0) = LatestFin(t_i, \mathcal{S}_0) - LatestSt(t_i, \mathcal{S}_0)$$

The second term, d, is a measure of the *density* of the critical path of the application, defined as the ratio of the number of tasks in the critical path to the total number of tasks in the application, N:

$$d = \frac{|\pi_0|}{N}$$

The value of d ($0 < d \leq 1$) is evaluated following each round of WCRT analysis. In our context here, it may be taken to indicate the reverse likelihood of the critical path shifting. If few, long tasks dominate the runtime of the application, then it is likely that these tasks will remain in the critical path. In this case, d evaluates to a small value, and the allocation objective will put more weight on $F(t_i)$. Hence, tasks that concretely participate in the current WCRT path will be prioritized over tasks that are unlikely to contribute to the WCRT. If, on the contrary, the critical path is formed by many tasks, then d will have a large value, which serves as a base weight for tasks that do not participate in the current critical path. This induces a more balanced runtime reduction across tasks that have similar possibilities of forming the application WCRT path.

We now examine the variable component of the objective function. The WCET of t_i in the presence of scratchpad allocation $\mathcal{S}$ is defined as

$$wcet(t_i, \mathcal{S}) = \ c(t_i) - saving(t_i, \mathcal{S}) + trans(t_i, \mathcal{S})$$

$c(t_i)$ is the uninterrupted worst-case running time of t_i without any allocation (that is, all code blocks are fetched from the main memory), which is evaluated once for each task during the initial Task Analysis step. From this value, we discount the time savings due to allocation, $saving(t_i, \mathcal{S})$, but account for additional instructions needed for transition between scratchpad memory and main memory, $trans(t_i, \mathcal{S})$. We elaborate these terms in the following.

Time Savings Let lat_S denote the latency to fetch one byte of instruction from the scratchpad, and let lat_M denote the latency to fetch one byte of instruction from the main memory. Naturally $lat_M > lat_S$, and the difference between the two gives the time saving for each access to a byte allocated in the scratchpad. The total time saving

for t_i due to allocation $\mathcal{S}$ is defined as

$$saving(t_i, \mathcal{S}) = \sum_{b \in Alloc(t_i, \mathcal{S})} freq_b \times area_b \times (lat_M - lat_S)$$

$area_b$ is the memory space (in bytes) occupied by a memory block b, and $freq_b$ is the execution frequency of b in the worst-case execution path; both information are obtained during the Task Analysis step. As defined in the problem formulation, $Alloc(t_i, \mathcal{S})$ refers to the selected set of code blocks of t_i in scratchpad allocation $\mathcal{S}$. In the ILP formulation, this term is expanded using binary decision variables X_b for each block $b \in Mem(t_i)$, whose value is 1 if $b \in Alloc(t_i, \mathcal{S})$, or 0 otherwise. The above definition translates to

$$saving(t_i, \mathcal{S}) = \sum_{b=0}^{|Mem(t_i)|} X_b \times freq_b \times area_b \times (lat_M - lat_S)$$

Transition Cost In this work, we handle code allocation with the granularity of a basic block. In this context, an additional constraint is needed to maintain correct control flow [119]. If two sequential basic blocks are allocated in different memory areas (that is, one in scratchpad and the other in main memory), then a jump instruction should be inserted at the end of the earlier block to maintain the correct fall-through destination. The insertion adds to the occupied area as well as increases the execution time of the basic block. This is reflected in the ILP formulation as follows.

Let Y_b be a binary variable whose value is 1 if b is allocated in the scratchpad ($X_b = 1$) but the block following it is not ($X_{b+1} = 0$) and 0 otherwise. When $Y_b = 1$, it means that an additional jump instruction should be inserted at the end of block b in the scratchpad to jump to the start of block $(b + 1)$ in the main memory. Similarly, let Z_b be the binary variable whose value is 1 if $X_b = 0$ and $X_{b+1} = 1$, to represent the insertion of a jump instruction at the end of block b in the main memory to jump to the start of block $(b + 1)$ in the scratchpad. Otherwise, Z_b has value 0. The definition of Y_b and Z_b can be

linearized in terms of X_b and X_{b+1} as follows:

$$Y_b <= X_b \ ; \ Y_b <= 1 - X_{b+1} \ ; \ Y_b >= X_b - X_{b+1}$$

$$Z_b <= X_{b+1} \ ; \ Z_b <= 1 - X_b \ ; \ Z_b >= X_{b+1} - X_b$$

Let the size of the jump instruction be jz bytes, and the time to execute the instruction be jt. The total time needed to fetch and execute these additional jump instructions is

$$trans(t_i, \mathcal{S}) = \sum_{b=0}^{|Mem_i|} freq_b \times (Y_b \times (jt + jz \times lat_S) + Z_b \times (jt + jz \times lat_M))$$

Finally, all blocks selected for allocation should fit into the scratchpad space of the host PE. Any additional jump instruction inserted into blocks allocated in the scratchpad should also be accounted for. Given the scratchpad size of cap_q attached to PE q, the capacity constraint is expressed as

$$\sum_{b=0}^{|Mem(t_i)|} (X_b \times area_b + Y_b \times jz) \ \leq \ space(t_i) \tag{7.5}$$

for each task t_i, and

$$\sum_{t_i : PE(t_i)=q} space(t_i) \ \leq \ cap_q \tag{7.6}$$

The ILP formulation is solved for X_b, Y_b and Z_b. The solution values for X_b indicate the actual selection of memory blocks for optimal allocation given current worst-case execution profile.

Complexity The complexity of this approach is determined by the number of decision variables, which in this case corresponds to the allocation decision (yes or no) for each

basic block in all tasks, that is

$$\sum_{i=1}^{N} |\{b \mid b \in t_i\}| \tag{7.7}$$

Clearly, this complexity depends on the granularity of scratchpad allocation chosen for the technique.

7.4.2 Interference Clustering (IC)

In this second method, we use task lifetimes determined by the WCRT analysis to form *interference clusters*. Tasks whose lifetimes overlap at some point are grouped into the same cluster. They will share the scratchpad for the entire duration of the common interval, from the earliest start time to the latest finish time among all tasks in the cluster.

The clustering is performed as detailed in Algorithm 4. We start with an empty cluster set (line 1). For each task, we try to find an existing cluster whose duration overlaps the lifetime of t (line 3). If there is no such cluster, we start a new cluster with t as the only member (lines 18–21). Otherwise, we add t into the existing cluster C and update C's duration (lines 4–9). If the duration of the cluster changes because of the new inclusion, we check against all other existing clusters whether any of them now overlaps with C, in which case they will be merged (lines 13–16).

After the clustering is decided, the same partitioning/allocation routine used in *PK* is employed *within each cluster*. The ILP capacity constraint in Equation 7.6 is therefore modified to

$$\sum_{t_i \in C} space(t_i) \;\leq\; cap_q$$

for each cluster C of tasks executing on PE q. Two distinct clusters on q are guaranteed to have disjoint execution time intervals, thus the allocated memory blocks of tasks in a later cluster can completely replace the scratchpad content belonging to the previously executing cluster when the corresponding execution interval is entered.

```
1  Clusters := ∅;
2  foreach task t in the application do
3      if ∃ C ∈ Clusters s.t.
         [C.Begin, C.End] ∩ [EarliestSt(t, S), LatestFin(t, S)] ≠ ∅ then
           /* Update C to include t */
4          C.Tasks := C.Tasks ∪ {t};
5          changed := FALSE;
6          if EarliestSt(t, S) < C.Begin then
7              C.Begin := EarliestSt(t, S); changed := TRUE;
8          if LatestFin(t, S) > C.End then
9              C.End := LatestFin(t, S); changed := TRUE;
           /* Check if inclusion of t affects the overall clustering */
10         if changed then
11             foreach cluster C' ∈ Clusters \ {C} do
12                 if [C.Begin, C.End] ∩ [C'.Begin, C'.End] ≠ ∅ then
                       /* Merge the two clusters */
13                     C.Tasks := C.Tasks ∪ C'.Tasks;
14                     C.Begin := min (C.Begin, C'.Begin);
15                     C.End := max (C.End, C'.End);
16                     Clusters := Clusters \ {C'};
17     else
           /* Create a new cluster C' for t */
18         C'.Tasks := {t};
19         C'.Begin := EarliestSt(t, S);
20         C'.End := LatestFin(t, S);
21         Clusters := Clusters ∪ {C'};
```

Algorithm 4: The *Interference Clustering (IC)* algorithm

The left part of Figure 7.9b shows the clustering decision for the given task schedule. Three clusters have been formed, and fm_1 as well as fm_4 have been identified as having no interference from any other task. Each of them is placed in a singleton cluster and enjoys the whole scratchpad space during its lifetime.

Complexity Given that the number of clusters is at most the number of tasks N, and that we check against all existing clusters when deciding on the placement of a task, the complexity of the clustering step is N^2 in the worst case. However, the complexity of the whole *IC* method is still dominated by the content selection routine, which equals the total number of basic blocks to be allocated (Equation 7.7), as in the *PK* method.

7.4.3 Graph Coloring (GC)

The *Interference Clustering (IC)* method is prone to produce large clusters due to transitivity. In Figure 7.9b, even though fm_2 and fs_0 do not interfere with each other, their independent interferences with fr_0 end up placing them in the same cluster. Because of this, simply clustering the tasks will likely result in inefficient decisions. The third method attempts to enhance the allocation within the clusters formed by the *IC* method by considering task-to-task interference relations captured in the interference graph. If we apply *graph coloring* to this graph, the resulting assignment of colors will give us groups of tasks that do not interfere with each other within the cluster. Tasks assigned to the same color have disjoint lifetimes, thus can reuse the same scratchpad space via further overlay.

The problem of graph coloring using the minimum number of colors is known to be NP-Complete. We employ the Welsh-Powell algorithm [140], a heuristic method that assigns the first available color to a node, without restricting the number of colors to use. Given the interference graph, the algorithm can be outlined as follows.

1. Initialize all nodes to uncolored.

2. Traverse the nodes in *decreasing order of degree*, assigning color 1 to a node if it is uncolored and no adjacent node has been assigned color 1.

3. Repeat step (2) with colors 2, 3, etc. until no node is uncolored.

The above algorithm is illustrated in Figure 7.10, featuring the tasks that have been assigned to the same cluster by *IC* in our running example.

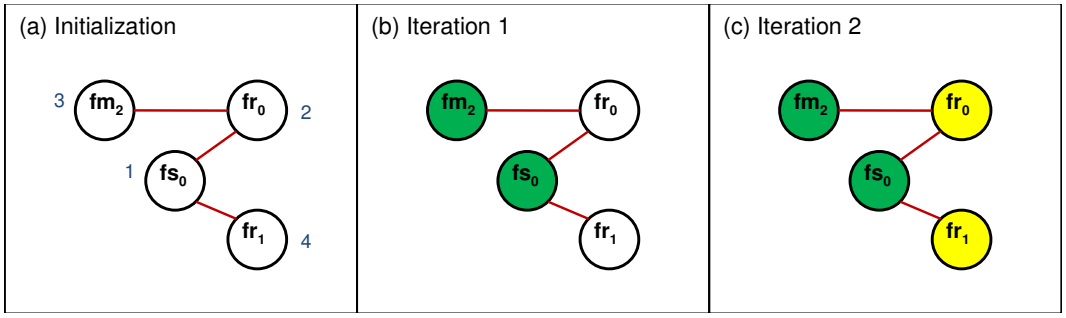


Figure 7.10: Welsh-Powell algorithm for graph coloring

The numbering in Figure 7.10a shows the order of traversal based on the degree of the nodes. The first iteration (Figure 7.10b) assigns the first color to all uncolored nodes with no neighbour of the same color, that is, fs_0 followed by fm_2. The next iteration (Figure 7.10c) assigns the second color to fr_0 and fr_1, and completes the coloring with a total of two colors used. This result dictates that the scratchpad space will be partitioned into two: one portion to be occupied by fm_2 and fs_0 during their respective lifetimes, and another portion to be occupied by fr_0 and fr_1 during their respective lifetimes.

Even though the graph coloring algorithm is a heuristic, in practice we observe that it does not pose serious sub-optimality to the solution, due to the fact that coloring options for actual task interference graphs are typically limited by the dependencies.

After we obtain the color assignment, we formulate the scratchpad partitioning/allocation with the refined constraint (replacing Equation 7.5) that each task t_i with assigned color

$color(t_i)$ can occupy at most the space allocated for $color(t_i)$, which we denote as $space(color(t_i))$.

$$\sum_{b=0}^{|Mem(t_i)|} (X_b \times area_b + Y_b \times jz) \leq space(color(t_i))$$

In place of Equation 7.6, we now have the constraint that the scratchpad space given to all M_q colors used for PE q adds up to the total scratchpad capacity cap_q of the PE:

$$\sum_{m=1}^{M_q} space(color(t_i)) \leq cap_q$$

Figure 7.9c shows the further partitioning within the second cluster previously formed by the *IC* scheme. fm_2 and fs_0 have been assigned the same color, and are allocated the same partition of the scratchpad to occupy at different time intervals. The similar decision applies to fr_0 and fr_1. The partition will be reloaded with the relevant task content when execution transfers from one task to another.

Complexity As each iteration assigns a color to at least one task, it requires at most N iterations to complete the coloring of all N tasks. In each iteration, each of the N tasks is considered for the current color by checking the color of all its neighbours (at most $N - 1$). The complexity of the graph coloring algorithm is thus N^3. Again, we observe from the allocation routine formulation that the complexity of this scheme remains capped at the allocation granularity as given by Equation 7.7.

7.4.4 Critical Path Interference Reduction (CR)

While the above three schemes try to make the best out of the given interference pattern, the final method that we propose turns the focus to reducing the interference instead.

This is motivated by the observation that allocation decisions are often compromised by heavy interference. When the analysis recognizes a potential preemption of one task by another, both tasks will have to do space-sharing; in addition, the lifetime interval of the preempted task t must make allowance for the time spent waiting for the preempting task u to complete. If t is on the critical path of the application, it may be beneficial for the application WCRT to let t complete first before allowing u to start. By removing the interference between t and u, we will be able to apply scratchpad overlay on the both of them as well. The gain from allocation will potentially reduce the execution time of both, leading to further reduction in WCRT. However, as the delay on the start of task u will inevitably be propagated to its successors, we need to take care that the overall delay will not overshadow the time saving on the WCRT path.

Our final proposed method proceeds as shown in Algorithm 5. We first work on the schedule produced by the WCRT analysis to improve the interference pattern. We consider all interferences in which tasks on the critical path are preempted or have to wait for tasks with higher priority (line 2). Each candidate interference is evaluated to determine the effect on overall WCRT if it is chosen to be eliminated. Interference among a task t and its preempting task u is eliminated by forcing u to wait until the completion of t, as illustrated in Figure 7.11b. The amount of slack introduced is thus equal to the remaining execution time of t after the original preemption (line 4), $LatestFin(t, \mathcal{S}) - LatestFin(u, \mathcal{S})$.

We then examine all tasks that execute after u in the current schedule to see if the propagated slack pushes their lifetime beyond the current WCRT (line 5). If a task v is dependent on u (either as a direct successor or in the transitive closure of u's succeeding tasks), its start and completion time will also be pushed back by up to the same amount of slack (line 8). If a task v is not dependent on u but is not interfering with u in the current schedule, then our iterative scratchpad algorithm will require us to maintain the non-interference among them. In that case, we check if the delayed lifetime of u now

repeat
  /* Identify task interferences on current WCRT path π_0 */
  $CrIf := \{ (t,u) \mid t \in \pi_0 \ \wedge \ u \in intf(t) \}$;
  /* $intf(t)$: set of higher priority tasks who may preempt t (Equation 7.3) */
  /* Evaluate effect of eliminating each interference */
  foreach interference pair $(t,u) \in CrIf$ **do**
    /* Estimate cost: propagated slack */
    $slack := LatestFin(t,\mathcal{S}) - LatestFin(u,\mathcal{S})$;
    $maxEnd := \mathcal{W}_0$; /* current WCRT */
    foreach task v that starts after u in current schedule **do**
      if v depends on u **then**
        $maxEnd :=$ max $(maxEnd, LatestFin(v,\mathcal{S}) + slack)$;
      else if
      $v \notin intf(u) \ \wedge \ EarliestSt(v,\mathcal{S}) < LatestFin(u,\mathcal{S}) + slack$ **then**
        $vSlack := LatestFin(u,\mathcal{S}) + slack - EarliestSt(v,\mathcal{S})$;
        $maxEnd :=$ max $(maxEnd, LatestFin(v,\mathcal{S}) + vSlack)$;
    /* Estimate gain: removed preemption and potential gain via overlay */
    $preemptLen := LatestFin(u,\mathcal{S}) - EarliestSt(u,\mathcal{S})$;
    Let $bestFit(x,y)$ be the set of most-accessed unallocated blocks of x to fit $space(y)$;
    $tGain := \sum_{b \in bestFit(t,u)} freq_b \times area_b \times (lat_M - lat_S)$;
    $uGain := \sum_{b \in bestFit(u,t)} freq_b \times area_b \times (lat_M - lat_S)$;
    /* Estimate projected WCRT if this interference is eliminated */
    $est\mathcal{W}(t,u) := maxEnd - preemptLen - tGain - uGain$;
    if $est\mathcal{W}(t,u) > \mathcal{W}_0$ **then**
      Remove (t,u) from $CrIf$;
  /* Choose most beneficial interference to eliminate */
  if $CrIf \neq \emptyset$ **then**
    Find $(t_m,u_m) \in CrIf$ s.t.
    $est\mathcal{W}(t_m,u_m) \leq est\mathcal{W}(t,u) \ \ \forall (t,u) \in CrIf$;
    Set constraint $EarliestSt(u_m,\mathcal{S}) \geq LatestFin(t_m,\mathcal{S})$;
    Run WCRT analysis to propagate lifetime shift;
until $CrIf = \emptyset$;

Algorithm 5: The *Critical Path Interference Reduction (CR)* algorithm

disrupts the lifetime of v. If necessary, the start and completion time of v will also be pushed back to maintain non-interference (lines 9–11).

We move on to examine the potential gain from eliminating the candidate interference. The first source of gain is the removed preemption delay along the critical path (Figure 7.11a) which amounts to the length of the preempting task u's lifetime (line 12). The second source of gain is the potential time saving due to allocation, as we can now allow t and u to occupy more scratchpad space via overlay. This value is estimated by assuming t can place a selection of its unallocated code blocks into the scratchpad space currently occupied by u, and vice versa (lines 13–15). The selected blocks for this estimation are those with highest access frequency along the task's current WCET path, among all blocks that are currently not allocated in the scratchpad. This strategy is chosen for efficiency, and differs from the actual ILP-based content selection in that it uses sub-optimal first-fit "packing" of these most-accessed blocks, and it does not account for the transfer code needed between the scratchpad and main memory.

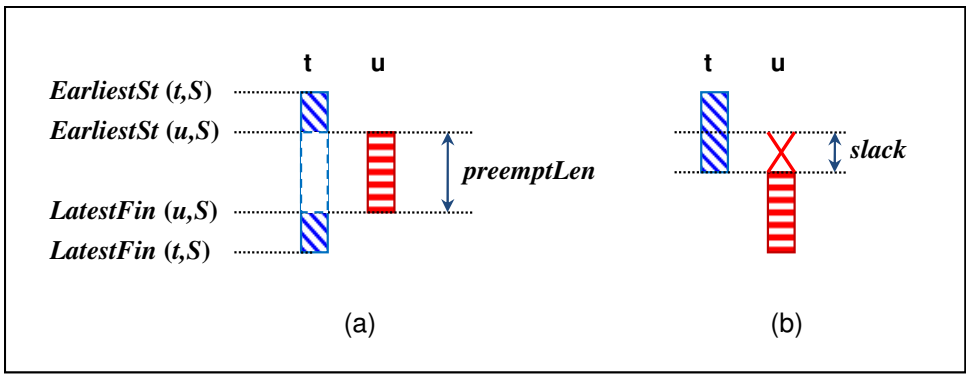


Figure 7.11: Mechanism of slack insertion for interference elimination — (a) task lifetimes without introducing slack, and (b) the corresponding lifetimes after introducing slack

We can now estimate the application WCRT after the elimination of this candidate interference. It is the latest completion time over all tasks affected by the inserted slack, discounting the original preemption delay and the projected gain from allocation (line

16). If this estimated WCRT is larger than the current WCRT, then we conclude that it is not beneficial to remove this interference, and remove this candidate from consideration (lines 17–18).

After evaluating the gain function for all candidate interferences along the critical path, we select one of them to remove. A sensible choice is the candidate with the best projected WCRT after removal (line 20). We eliminate this interference by forcing a delayed start time for the preempting task (line 21), then propagate the shift to all tasks by re-running the WCRT analysis (line 22). Certainly, new interferences are not allowed to arise in this step.

From the new schedule, we again consider preemptions on the critical path, which may or may not have shifted. The elimination and re-analysis are iterated until no more interferences can be eliminated from the current critical path. As the number of tasks N is finite, the number of potential interference relations is also finite, that is, at most $N(N-1)$ as established earlier. Therefore, it is guaranteed that the algorithm will terminate.

After the interference elimination step, we proceed to perform scratchpad partitioning/allocation routine as used by the *Graph Coloring (GC)* scheme on this improved interference graph.

In Figure 7.9d, the interference between fs_0 and fr_1 has been eliminated by letting fr_1 wait out the lifetime of fs_0 instead of starting immediately after the completion of its predecessor fr_0. This improvement frees fr_1 from all interference. It can now occupy the whole scratchpad memory throughout its lifetime.

Complexity Since each iteration chooses one interference to eliminate, the maximum number of iteration equals the maximum number of possible interferences, that

is, $N(N-1)$. Each iteration involves (1) estimating the gain of eliminating each existing interference, and (2) re-running the WCRT analysis after elimination of the chosen candidate.

Step (1) has a maximum of $N(N-1)$ interference candidates to evaluate. Each evaluation employs the basic block selection routine (lines 13–15) whose complexity equals the maximum number of basic blocks in a task (Equation 7.7). Step (2) involves checking dependencies and existing interferences between pairs of tasks with complexity N^2, a term that is dominated by the complexity of the first step. The whole iterative process thus has a complexity of

$$N^4 \times \sum_{i=1}^{N} |\{b \mid b \in t_i\}|$$

In practice, the number of interfering tasks on the critical path is rarely close to the maximum. In addition, it is typically the case that only a handful of major interferences give enough gain to offset the slack cost. There is thus rarely a need to go through a lot of iterations before achieving substantial reduction and reaching a local optimum. This behaviour significantly curbs the complexity of the scheme, as we shall observe in the experiments.

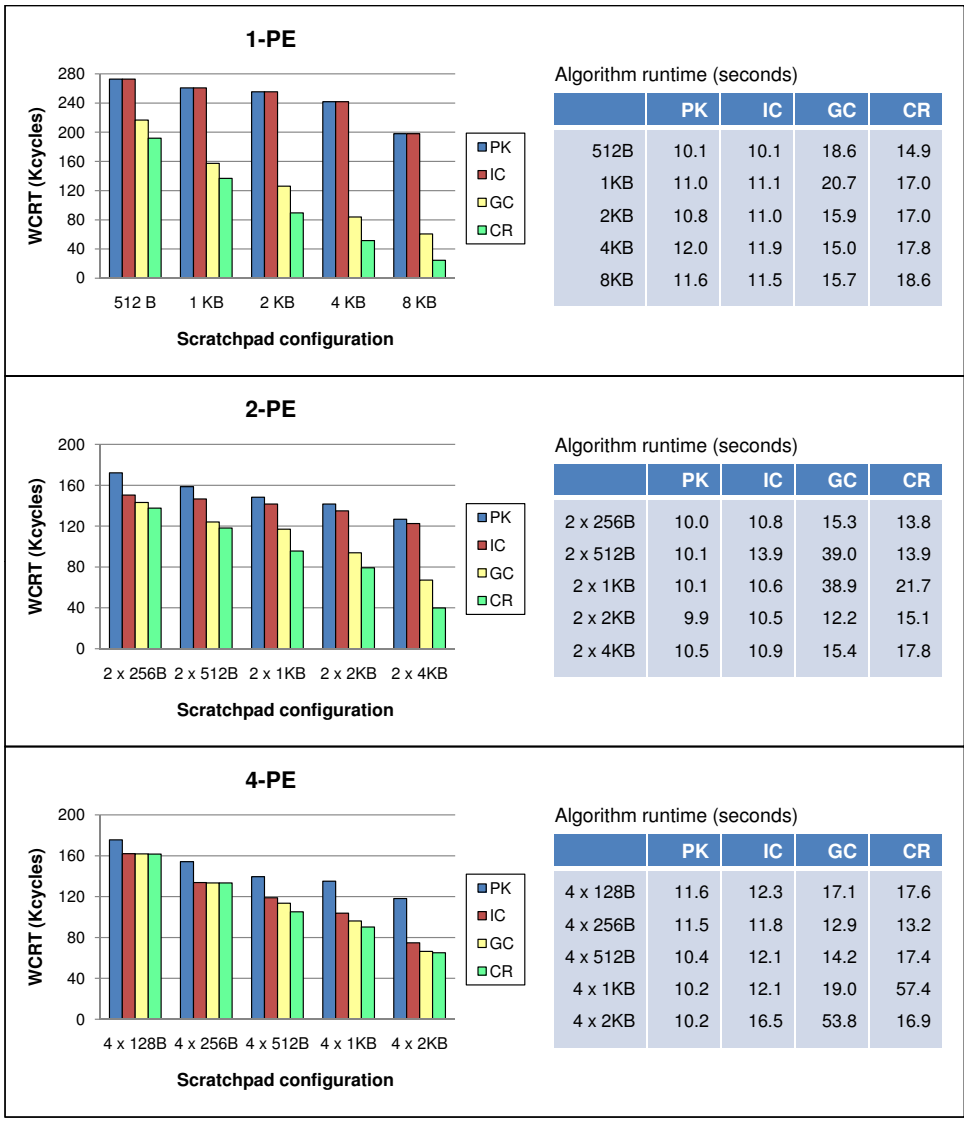
7.5 Experimental Evaluation

Setup We use two real-life embedded applications to evaluate the scratchpad allocation schemes that we have presented. Our first case study is the Unmanned Aerial Vehicle (UAV) control application from PapaBench [94], a derivation from the real-time embedded UAV control software Paparazzi. We adapt the C source code of this application into a distributed implementation. The extracted application tasks are then compiled on the SimpleScalar architectural platform [12] for further analysis.

The controller consists of two main functional units, `fly_by_wire` and `autopilot`, which are inter-connected by an SPI serial link. The `fly_by_wire` unit is responsible for managing radio-command orders and servo-commands, while `autopilot` runs the navigation and stabilization tasks of the aircraft. The two components can operate in one of two modes. In the *manual* mode, `fly_by_wire` obtains navigation instructions via the radio, passes them through `autopilot` computation, then communicates the necessary actions to the servos. In the *automated* mode, `autopilot` reads information from the GPS unit to compute necessary adjustments, which are then passed to `fly_by_wire` to be actuated by the servos.

Figure 7.1 shows the active processes in one of the scenarios under the manual mode. This is the MSC we use in the experiments. The original implementation as shown uses 2 PEs with a total of 5KB scratchpad memory space. In our experiments, we vary the number of PEs from 1 to 4 for the purpose of observation. The 1-PE case has all tasks assigned to the single PE. The 2-PE case follows the same division as the original implementation (`fly_by_wire` on one PE and `autopilot` on another) shown in Figure 7.1. The task scheduling in the 4-PE case is shown in Table 7.1 along with the code size and uninterrupted worst-case runtimes of each task. The uninterrupted runtime values are obtained via WCET analysis of the program code assuming all instructions are fetched from the main memory. These values remain the same irrespective of PE assignment.

We assume uniform execution time of 1 cycle per instruction. The memory access latencies are set to the typical values in real systems: 1 cycle for an access to the scratchpad and 100 cycles for an access to the main (off-chip) memory, with the fetch width of 16 bytes. Total scratchpad size (for instructions) is varied from 512B to 8KB, distributed evenly among the PEs. The range is chosen to provide reasonable contention for memory space given the code size of the tasks. The ILP formulation for scratchpad content selection is solved using the tool CPLEX from ILOG [29].



Algorithm runtime (seconds)

	PK	IC	GC	CR
512B	10.1	10.1	18.6	14.9
1KB	11.0	11.1	20.7	17.0
2KB	10.8	11.0	15.9	17.0
4KB	12.0	11.9	15.0	17.8
8KB	11.6	11.5	15.7	18.6

Algorithm runtime (seconds)

	PK	IC	GC	CR
2 x 256B	10.0	10.8	15.3	13.8
2 x 512B	10.1	13.9	39.0	13.9
2 x 1KB	10.1	10.6	38.9	21.7
2 x 2KB	9.9	10.5	12.2	15.1
2 x 4KB	10.5	10.9	15.4	17.8

Algorithm runtime (seconds)

	PK	IC	GC	CR
4 x 128B	11.6	12.3	17.1	17.6
4 x 256B	11.5	11.8	12.9	13.2
4 x 512B	10.4	12.1	14.2	17.4
4 x 1KB	10.2	12.1	19.0	57.4
4 x 2KB	10.2	16.5	53.8	16.9

Figure 7.12: WCRT of the benchmark application after allocation by *Profile-based Knapsack (PK)*, *Interference Clustering (IC)*, *Graph Coloring (GC)*, and *Critical Path Interference Reduction (CR)*, along with algorithm runtime

PE_i	t_i	Codesize (bytes)	c_i (cycles)
1	fm_0	808	12,237
	fm_1	96	612
	fm_2	96	612
	fm_3	1,696	10,487
	fm_4	136	815
	fm_5	248	1,629
1	fv_0	520	3,866
	fv_1	656	52,957
2	fr_0	384	2,646
	fr_1	4,552	28,008
2	fs_0	272	1,932
	fs_1	992	16,597
	fs_2	1,840	11,606
3	am_0	768	10,186
	am_1	96	612
	am_2	96	612
	am_3	1,240	9,173
	am_4	1,536	9,579
3	ad_0	352	2,442
	ad_1	2,296	13,345
	ad_2	6,496	36,374
4	as_0	560	3,968
	as_1	2,744	17,726
	as_2	1,720	12,116
	as_3	168	1,221
	as_4	656	4,277
4	ag_0	400	2,748
4	ar_0	5,520	34,944

Table 7.1: Code size and WCET of tasks in the PapaBench application

Results and Discussion Figure 7.12 shows the final WCRT (in kilocycles) of the application for various scratchpad configurations, after applying the four discussed schemes. The three charts correspond to the cases where the tasks are distributed on 1, 2, and 4 PEs. Obviously, with more PEs, less interference is observed among the tasks. On the other hand, it also means less scratchpad space per PE for the same total scratchpad size, which limits the maximum space utilizable by a task. In some configurations, we do see no reduction or even an increase in WCRT as more PEs are employed.

When only 1 PE is utilized, most tasks are interfering with each other. We can see a drastic WCRT improvement from 1-PE to 2-PE setting for all schemes, which confirms the observation that task interferences significantly influence application response time. With 1 PE, the *Interference Clustering (IC)* method does not improve over the baseline *Profile-based Knapsack (PK)*, as the transitive interference places most tasks into the same space-sharing cluster. With more PEs employed, *IC* is able to perform better than *PK*. The *Graph Coloring (GC)* method performs no worse than *IC* in all cases shown here, as it has a more refined view of interference relation among individual tasks. The improvement is dramatic in the 1-PE case where task interferences are abundant. The

Critical Path Interference Reduction (CR) method is, in turn, able to further improve the performance of *GC*. Certainly, when task interferences are scarce as in the 4-PE case, the improvement that arises from the refined account for task interaction also lessens.

From these results, we can conclude that the proposed scheme *CR* gives the best WCRT improvement over all other schemes. This justifies the strategy of eliminating critical interferences via slack enforcement, whenever any additional delay that is incurred can be overshadowed by the gain through a better scratchpad sharing and allocation scheme.

Finally, the tables of Figure 7.12 show the comparison of algorithm runtimes when run on a 3.0 Ghz Pentium 4 CPU with 1MB cache and 2GB memory. We observe a wide variation of runtime in certain cases, for example the runtime of *GC* in the 2-PE case. The dominant component of the algorithm runtime is the ILP solution time, which highly depends on the complexity of the problem. The variation in runtime arises from the variation in the complexity of the formulation due to different configurations and scratchpad sharing decisions. Nevertheless, the runtimes of all schemes as shown here are reasonably efficient, ranging from 10 seconds to less than a minute.

7.6 Extension to Message Sequence Graph

Message Sequence Graph (MSG) is a finite state automaton where each state is described by an MSC. Multiple outgoing edges from a node in the MSG represent a choice, so that exactly one of the destination charts will be executed in succession. Figure 7.13 shows an example of an MSG, modeling the PapaBench application. The top left box shows the MSG, and the labeled boxes display the MSCs corresponding to the nodes. The MSC model used in the experiment presented earlier (Figure 7.1) corresponds to a single execution path through this MSG, that is, it shows only one of the possible scenarios represented by this MSG.

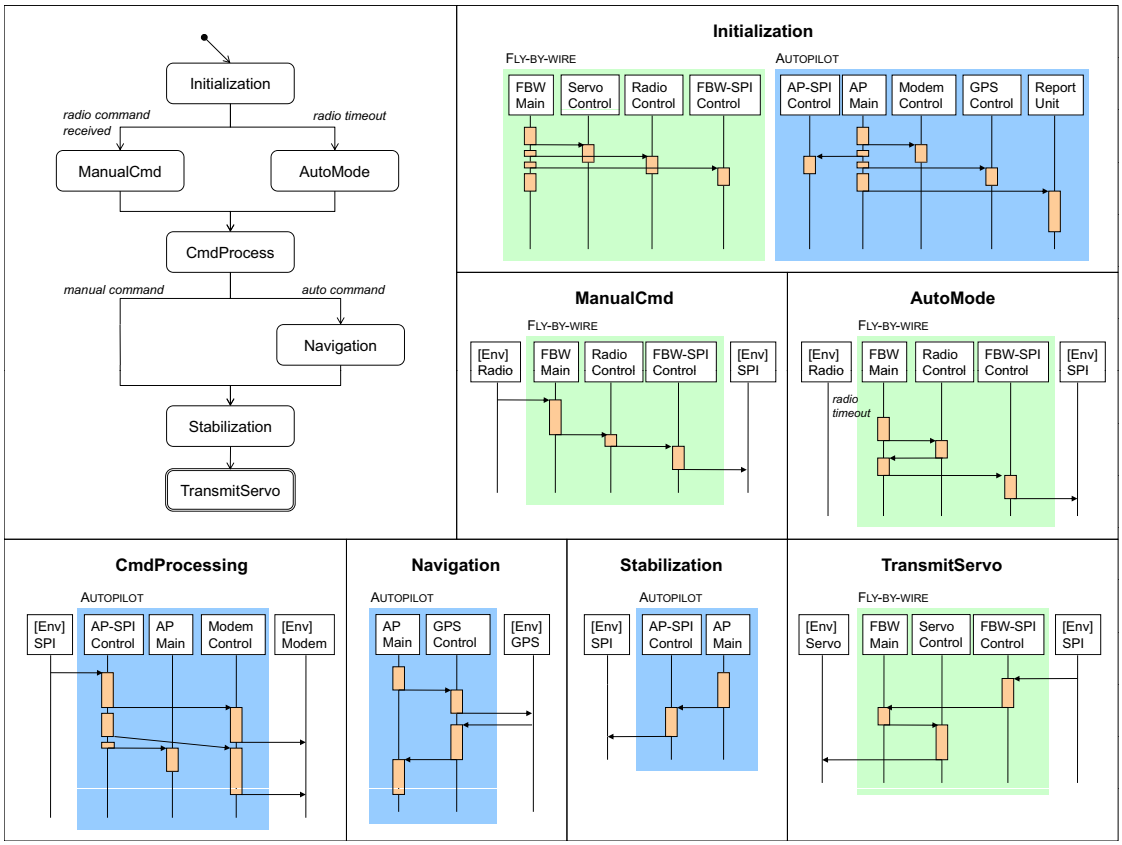


Figure 7.13: Message Sequence Graph of the PapaBench application

While an MSC describes a single scenario in the system execution, an MSG describes the control flow between these scenarios, allowing us to form a complete specification of the application. Therefore, to optimize the WCRT of the entire application, the analysis as well as the scratchpad allocation technique need to be extended to take into account the conditional control flow specified by the MSG model.

An execution of the modeled application traces a path in the MSG from an initial state to a terminal state (marked with double lining), and can be viewed as a concatenation of the MSCs along that path [45]. Here, we consider the *synchronous* concatenation of the MSG, where all the tasks in an MSC (that is, an MSG node) must complete before any task in a subsequent MSC can execute. Each MSC can thus be viewed independently. The extended scratchpad allocation technique proceeds as follows.

1. Perform scratchpad allocation on each MSC to obtain the WCRT-optimizing allocation along with the post-allocation WCRT value for each MSG node.

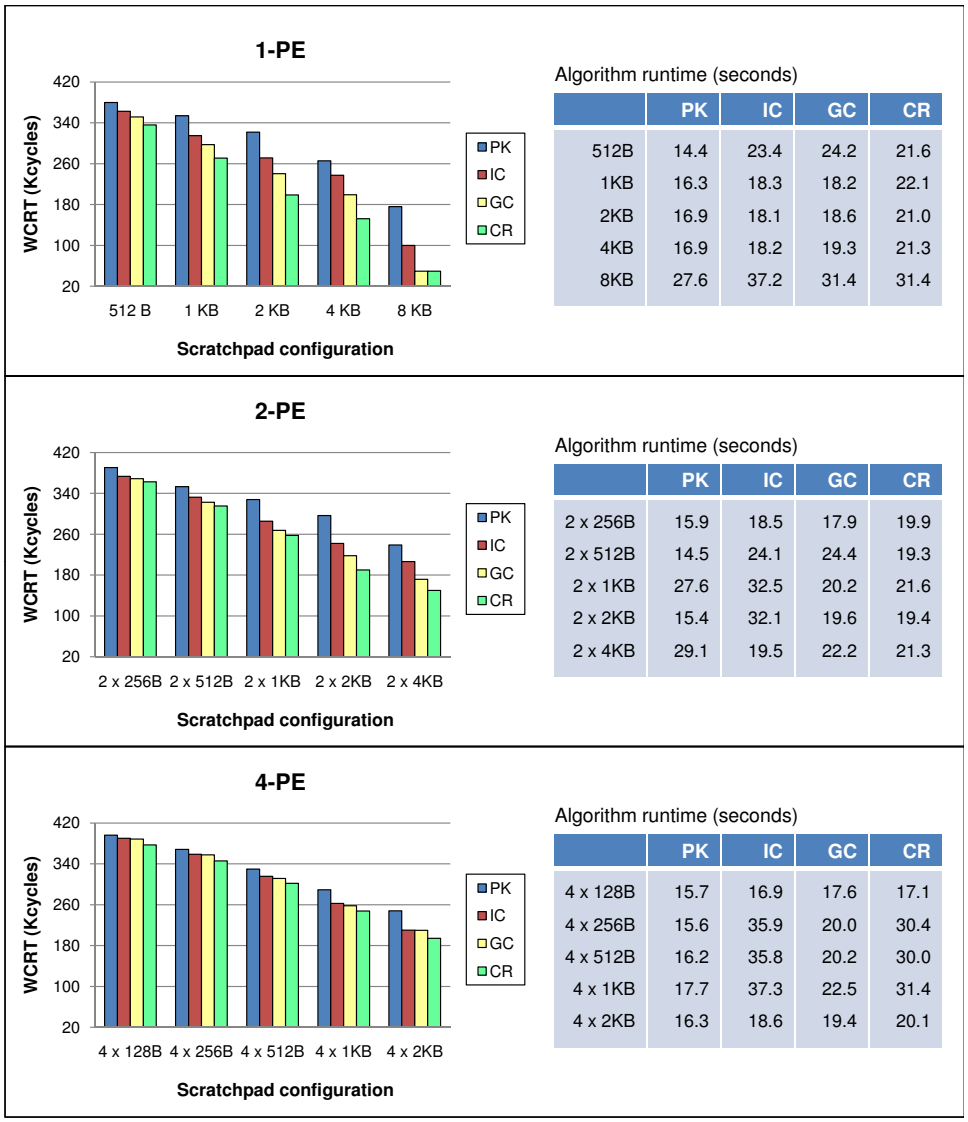
2. Traverse the MSG to find the longest path in terms of total post-allocation WCRT. Since there is an allocation for each MSG node (which is an MSC), after an MSG node finishes, the scratchpad contents may be reloaded.

The MSG may contain loops between the initial state and the terminal state, in which case we require that the maximum number of times the cyclic edges can be traversed is known, so that the longest path is well-defined. The WCRT of the application is then the sum of the WCRT of the MSCs along the longest path.

Experimental Evaluation We run the extended scratchpad allocation on the PapaBench MSG (Figure 7.13). The assignment of processes to PEs as well as latency settings are the same as in our experiments on the single MSC, presented in the previous section.

Figure 7.14 shows the resulting WCRT after allocation by the four schemes. Here, task interactions are limited within each MSG node, and we see that distributing the application on more PEs *does not* always result in overall WCRT reduction. This is due to the fact that the PEs are not necessarily fully utilized in each node, as processes have been mapped to PEs according to functionality without regard for load balancing. For example, in the 2-PE case where one PE takes up `fly_by_wire` tasks and the other executes `autopilot` tasks, then only one of them is active in MSG nodes other than `Initialization`. As we have observed earlier, for the same total scratchpad space, utilizing more PEs narrows down the available space on each PE, and tasks may contrarily take longer time to complete.

The results in terms of post-allocation WCRT of the complete application still confirm our hypothesis that allocation techniques with more sophisticated view of task interactions obtain better results. In particular, our proposed scheme *CR* still gives the best



Algorithm runtime (seconds)

	PK	IC	GC	CR
512B	14.4	23.4	24.2	21.6
1KB	16.3	18.3	18.2	22.1
2KB	16.9	18.1	18.6	21.0
4KB	16.9	18.2	19.3	21.3
8KB	27.6	37.2	31.4	31.4

Algorithm runtime (seconds)

	PK	IC	GC	CR
2 x 256B	15.9	18.5	17.9	19.9
2 x 512B	14.5	24.1	24.4	19.3
2 x 1KB	27.6	32.5	20.2	21.6
2 x 2KB	15.4	32.1	19.6	19.4
2 x 4KB	29.1	19.5	22.2	21.3

Algorithm runtime (seconds)

	PK	IC	GC	CR
4 x 128B	15.7	16.9	17.6	17.1
4 x 256B	15.6	35.9	20.0	30.4
4 x 512B	16.2	35.8	20.2	30.0
4 x 1KB	17.7	37.3	22.5	31.4
4 x 2KB	16.3	18.6	19.4	20.1

Figure 7.14: WCRT of the complete PapaBench application after allocation by *Profile-based Knapsack (PK)*, *Interference Clustering (IC)*, *Graph Coloring (GC)*, and *Critical Path Interference Reduction (CR)*, along with algorithm runtime

WCRT improvement. In a similar trend as in the single MSC case, the improvements are more pronounced in configurations with less number of PEs and hence heavier task interferences.

As charts are examined individually, the method does not experience noticable scalability problem. Given a total of 7 individual MSCs with varying degree of complexity in this application, we see only about 50% increase in total algorithm runtime as shown in the tables of Figure 7.14.

Extension to Asynchronous Concatenation An alternative way to form an execution path through an MSG is the *asynchronous* concatenation of the charts labeling the MSG nodes. In this case, a task in an MSC may start as soon as all tasks of the same process in the preceding MSC have completed. The result of an asynchronous concatenation forms an MSC, while it is not necessarily so in the synchronous case.

Our scratchpad allocation method can be extended for the case of asynchronous concatenation as follows. By enumerating all paths through the MSG, we obtain all possible execution scenarios of the application. If the MSG contains loops, then the bounds on the edge counts should also be supplied to enable the enumeration of all possible paths. Each path is formed by asynchronously concatenating the MSCs, thus resulting in an MSC. Our method then performs WCRT analysis along with scratchpad allocation for each resulting MSC as before. The MSG path (which is an MSC) with the maximum WCRT value will thus determine the WCRT of the application, and the allocation decision corresponding to that MSC is the worst-case-optimizing scratchpad allocation for the application.

The complexity of this concatenation method lies in the enumeration of the paths. The WCRT analysis and scratchpad allocation operate on each result of concatenation as a stand-alone MSC. As such, in our evaluation of this setting, the trend in results is similar

to the single-MSC case, that is, the application WCRT improves as more refined view of task interactions is employed in the allocation schemes.

7.7 Method Scalability

To evaluate the scalability of our proposed technique, we run it on a more complex application adapted from DEBIE-I DPU Software [39], an in-situ space debris monitoring instrument developed by Space Systems Finland Ltd. We model the software as an MSG, shown in Figure 7.15. The main functions involved have been broken down into several concurrent processes.

The DEBIE instrument utilizes up to four Sensor Units (SU) to detect particle impacts on the spacecraft. As the system starts up, it performs initializations and runs tests of the main functionalities. The system then enters the Standby state. When the command to start the acquisition of impact data is received via the Telecommand handler, the system goes into the Acquisition state and turns on at least one of the Sensor Units. In this mode, each particle impact will trigger a series of measurement, and the data are classified and logged for further transmission (telemetry) to the ground station. Data acquisition will continue until the stopping command is received, after which the system returns to the Standby state. In either mode, the Health Monitoring process periodically monitors the health of the instrument and runs housekeeping checks. If any error is detected, the system will reboot.

The code size and WCET of each task as well as its mapping to 4 PEs are listed in Table 7.2. When running the experiments for 2 PEs, the tasks in PE_1 and PE_2 are assigned to the first PE, and the rest of the tasks are assigned to the second PE. For this application, we use the main memory latency value of 10 cycles instead of 100 cycles as the previous setting, in order to lessen the skewness of execution time values among the tasks. The

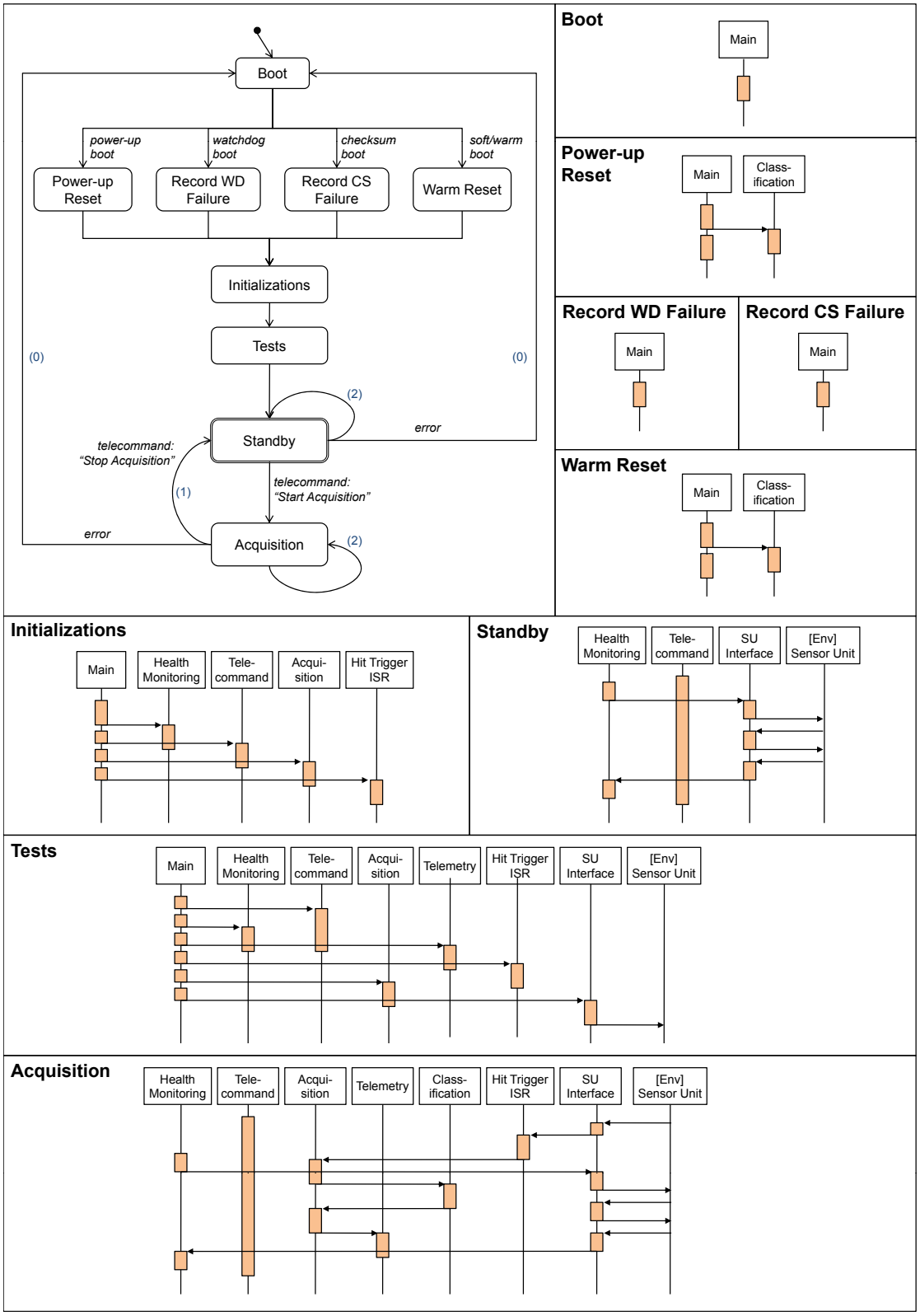


Figure 7.15: Message Sequence Graph of the DEBIE application

scratchpad latency remains at 1 cycle. Following the increase in average code size of the tasks, the total scratchpad size is now varied from 1 KB to 16 KB.

PE_i	t_i	Codesize (bytes)	c_i (cycles)
1	mn-boot	3,200	4,325
	$mn\text{-}pw_1$	9,456	14,589
	$mn\text{-}pw_2$	3,472	6,784
	mn-wd	3,400	6,745
	mn-cs	3,400	6,745
	$mn\text{-}wr_1$	3,408	10,757
	$mn\text{-}wr_2$	5,952	6,594
	$mn\text{-}ini_1$	320	260
	$mn\text{-}ini_2$	376	271
	$mn\text{-}ini_3$	376	271
	$mn\text{-}ini_4$	376	271
	$mn\text{-}tst_1$	240	180
	$mn\text{-}tst_2$	240	180
	$mn\text{-}tst_3$	240	180
	$mn\text{-}tst_4$	240	180
	$mn\text{-}tst_5$	240	180
	$mn\text{-}tst_6$	240	180
1	tm-tst	56,960	839,607
	tm-acq	3,768	5,560
2	ht-ini	616	507
	ht-tst	10,776	105,224,302
	ht-acq	8,016	1,008,155
2	su-tst	50,176	15,606,989
	$su\text{-}sby_1$	6,512	103,375,726
	$su\text{-}sby_2$	4,392	51,684,932
	$su\text{-}sby_3$	1,320	91,593
	$su\text{-}acq_0$	2,536	712
	$su\text{-}acq_1$	6,512	103,375,726
	$su\text{-}acq_2$	4,392	51,684,932
	$su\text{-}acq_3$	1,320	91,593

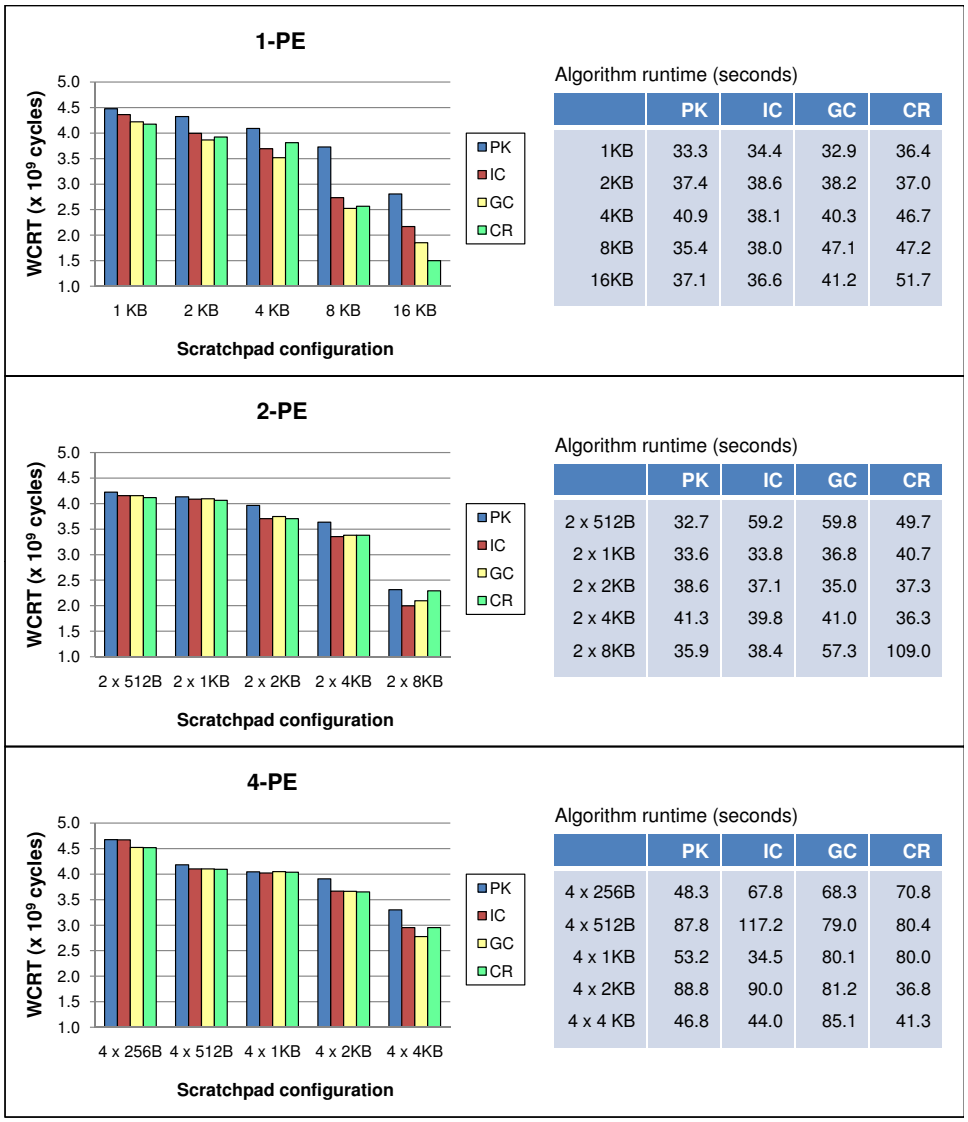
PE_i	t_i	Codesize (bytes)	c_i (cycles)
3	cl-pw	1,648	1,266
	cl-wr	1,648	1,266
	cl-acq	3,064	14,637
3	tc-ini	4,408	3,341
	tc-tst	45,368	38,009,988
	tc-sby	23,288	117,268
	tc-acq	23,288	117,268
4	aq-ini	200	165
	aq-tst	44,128	126,996,985
	$aq\text{-}acq_1$	3,136	2,400
	$aq\text{-}acq_2$	3,024	2,688
4	hm-ini	5,224	155,055,938
	hm-tst	44,176	743,373,112
	$hm\text{-}sby_1$	16,992	413,497,626
	$hm\text{-}sby_2$	448	343
	$hm\text{-}acq_1$	16,992	413,497,626
	$hm\text{-}acq_2$	448	343

Table 7.2: Code size and WCET of tasks in the DEBIE application

The DEBIE application model as shown in Figure 7.15 is a cyclic MSG, and we have indicated the bounds used for our experiments in the brackets labeling the cyclic edges. Each specified bound is an absolute count of the number of times the edge is taken in a complete scenario according to the MSG.

The optimization results in terms of post-application WCRT for the MSG along with algorithm runtimes for all four schemes are shown in Figure 7.16.

We observe two interesting phenomena here. First, we see that *GC* may sometimes perform worse than *IC* (for example, in the 2-PE, 2 x 2KB scratchpad configuration). In



Algorithm runtime (seconds)

	PK	IC	GC	CR
1KB	33.3	34.4	32.9	36.4
2KB	37.4	38.6	38.2	37.0
4KB	40.9	38.1	40.3	46.7
8KB	35.4	38.0	47.1	47.2
16KB	37.1	36.6	41.2	51.7

Algorithm runtime (seconds)

	PK	IC	GC	CR
2 x 512B	32.7	59.2	59.8	49.7
2 x 1KB	33.6	33.8	36.8	40.7
2 x 2KB	38.6	37.1	35.0	37.3
2 x 4KB	41.3	39.8	41.0	36.3
2 x 8KB	35.9	38.4	57.3	109.0

Algorithm runtime (seconds)

	PK	IC	GC	CR
4 x 256B	48.3	67.8	68.3	70.8
4 x 512B	87.8	117.2	79.0	80.4
4 x 1KB	53.2	34.5	80.1	80.0
4 x 2KB	88.8	90.0	81.2	36.8
4 x 4 KB	46.8	44.0	85.1	41.3

Figure 7.16: WCRT of the DEBIE application after allocation by *Profile-based Knapsack (PK)*, *Interference Clustering (IC)*, *Graph Coloring (GC)*, and *Critical Path Interference Reduction (CR)*, along with algorithm runtime

these cases, the performance drop is caused by the decision to allocate less space to a color which is used by only one task, which then becomes the critical task in the particular chart. Recall that the allocation of scratchpad space and the content selection are simultaneously performed to optimize for the approximated definition of total WCRT. As coloring imposes a "grouping" among tasks, the evaluation of the gain function may be biased towards a group with more tasks, moreover when it is difficult to predict which tasks may turn out critical after all task interactions are considered.

Secondly, there are also cases where *CR* results in worse performance than *GC* or *IC* (for example, in the 1-PE, 4KB scratchpad configuration). Investigation reveals that this deterioration happens over the iterative improvements for certain charts in the application (recall that allocation is done independently on each chart). The decision to eliminate selected interferences does yield better reduction in the first few iterations of the allocation scheme. However, as it continues, it hits a point when the refined allocation/content selection does not improve the WCRT, and thus the iteration ends. Meanwhile, the *GC* or *IC* method, while achieving longer WCRT at first, continues to refine the allocation decision over the iteration and finally reaches a better overall WCRT. The choice to eliminate certain interferences certainly caters to the application WCRT as best as it can predict based on the critical path at that moment, yet the non-interference constraint that is carried forward to the next iterations restricts the "movement" of tasks in the application. In these cases, the shifting of tasks after the iterative improvement ultimately gives a chance to *GC* and *IC* to obtain more beneficial allocation even with less overlay.

As far as scalability is concerned, we see that all schemes can complete their computation below 2 minutes, even though the charts to be optimized are considerably more complex than the previous PapaBench benchmark. Again, the algorithm runtime is dominated by ILP solution for scratchpad content selection, which increases in complexity as the application has more task code blocks to consider for optimal allocation. We can therefore conclude that there is no evident scalability problem in all four schemes.

7.8 Chapter Summary

We have presented a detailed study of scratchpad allocation schemes for concurrent embedded software running on single or multiple processing elements. The novelty of this work stems from taking into account both concurrency and real-time constraints in our scratchpad allocation. Our allocation schemes consider (1) communication or interaction among the threads or processes of the application, as well as (2) interference among the threads or processes due to preemptive scheduling in the processing elements. The Message Sequence Chart (MSC) model is chosen as it shows the process interaction explicitly. We have also presented an extension of our scheme to the Message Sequence Graph (MSG) model, where charts labeling the nodes in the graph are concatenated to form a complete execution scenario.

As the interactions and interference among the processes can greatly affect the worst-case response time (WCRT) of a concurrent application, our scratchpad allocation methods achieve substantial reduction in WCRT as evidenced by our experiments on two real-world embedded case studies.

Chapter 8

Integrated Scratchpad Allocation and Task Scheduling

In this chapter, we expand our view to look at how scratchpad allocation may interact with other multiprocessing aspects. One particular factor that affects application response time is the mapping/scheduling of tasks on multiple processing cores in a multiprocessor system-on-chip (MPSoC). We design an integrated task mapping, scheduling, scratchpad partitioning, and data allocation technique as an ILP formulation to explore the optimal performance limit.

8.1 Introduction

Increasing concerns about the energy and thermal behavior of embedded systems are leading to designs with multiple homogeneous/heterogeneous cores or processors on a single chip. Significant research efforts have been invested in partitioning, mapping, and scheduling the tasks corresponding to an embedded application onto multiple processors. However, the increasing performance gap between on-chip and off-chip memory

implies that the design of on-chip memory hierarchy has the maximum impact on the performance of an application. In this chapter, we focus on customization of on-chip scratchpad memory for multiprocessor system-on-chip (MPSoC) architectures.

An MPSoC is an integrated circuit containing multiple instruction-set processors on a single chip that implements most of the functionality of a complex electronic system. An MPSoC architecture is, in general, customized for an embedded application. A critical component of this customization process is the on-chip memory system configuration — in this case, the scratchpad memory management.

We consider an MPSoC architecture where each processor has its private scratchpad. In addition to this, a processor can also access another processor's private scratchpad albeit with an increased latency. Given an application and a budget for total on-chip scratchpad, our goal is to determine the appropriate configuration and data mapping for the private scratchpads of all processors so as to maximize the performance of the application. The appropriate configuration of a processor's private scratchpad critically depends on the tasks mapped to that processor. Therefore, task mapping/scheduling and scratchpad configuration are dependent on each other. Traditional design space exploration frameworks implement these two phases separately, leading to sub-optimal performance. In the following, we propose an integer linear programming (ILP) based technique for integrated task mapping/scheduling, scratchpad partitioning, and data mapping.

8.2 Task Mapping and Scheduling

Before addressing the problem of integrated task mapping/scheduling and scratchpad allocation, let us first manage a comprehensive view on the problem of task mapping and scheduling and review the state of the art in this area.

Uniprocessor scheduling is a well-researched area. Liu and Layland in their classic paper [81] prove that *Rate Monotonic Scheduling (RMS)* is optimal for preemptive uniprocessor scheduling with static priorities where task deadlines equal task periods, while *Earliest Deadline First (EDF)* is optimal when dynamic priorities are used. Further, they show that the least upper bound of the processor utilization factor for a set of N tasks to be schedulable is $U = N(2^{1/N} - 1)$ for RMS and 1 for EDF.

Tasks can be scheduled on multiprocessing systems using a *global scheduling (non-partitioning)* or a *partitioning* scheme [70]. In a global scheduling scheme, tasks can be scheduled on any processor and, after preemption, can be resumed on a different processor. In a partitioning scheme, each task is assigned to a processor and is only executed on this processor.

Global Scheduling A class of global multiprocessor scheduling algorithms with relatively acceptable complexity is Proportionate-fair (Pfair) [17, 115, 7]. It is optimal for scheduling recurrent real-time tasks on a multiprocessor system, with the goal of ensuring fairness. The method allocates processor time in *slots* of quantum length. In each slot, each processor can be assigned to at most one task. The quantum allocation is done by sub-dividing each task into a sequence of quantum-length subtasks. The disadvantage of Pfair scheduling is that it tends to incur frequent preemptions and migrations that may increase cache misses.

Partitioning In the partitioning scheme, tasks are first assigned to processors (*task mapping*). Within each processor, tasks can then be scheduled using optimal uniprocessor scheduling policies such as RMS or EDF. While RMS and EDF are optimal within the individual processors, the partitioning scheme is not optimal for multiprocessors [20]. However, it is presently favored as it is efficient and reasonably effective. In

particular, this scheme incurs less scheduling overhead and enables the application of well-researched uniprocessor scheduling strategies on each processor individually [82].

The task mapping problem is intractable, thus a number of heuristics have been developed for this purpose, including *First-Fit (FF)*, *Next-Fit (NF)*, *Best-Fit (BF)*, *Worst-Fit (WF)*, *First-Fit Decreasing (FFD)* and *Best-Fit Decreasing (BFD)* [1, 83]. Lopez in [82] proves that the combination of FF task allocation and EDF scheduling scheme (EDF-FF) achieves the highest worst-case achievable utilization among all pairs of uniprocessor task allocation–scheduling algorithm.

The problem of scheduling a task graph on multiple homogenous processors in order to minimize execution time (or energy) has been studied extensively. In its general form, this problem is NP-complete; a comprehensive comparison of existing heuristics is presented by Kwok and Ahmad [68]. These works mostly consider non-pipelined scheduling. Benini et al. [18] propose a hybrid constraint programming and integer programming based approach for finding the optimal pipelined schedule. The related problem of mapping and scheduling tasks to a set of heterogeneous processing elements has been studied in the context of hardware/software co-design [144], which shares technical similarities with our multiprocessor scheduling problem. Among proposed solutions to this co-design problem is an ILP-based solution by Niemann and Marwedel [96]. Recently, various research groups have also proposed *pipelined* scheduling solutions to this problem, especially in the context of streaming applications. Chatha and Vemuri [25] propose a branch-and-bound solution whereas Kuang et al. [67] propose an ILP-based solution. Both solutions aim to minimize the total component cost and hence the number of pipeline stages.

For a comprehensive review of task scheduling covering uniprocessor/multiprocessor systems, periodic/aperiodic processes and static/dynamic algorithms, readers may refer to the paper by Burns [20].

8.3 Problem Formulation

We shall now present the problem formulation.

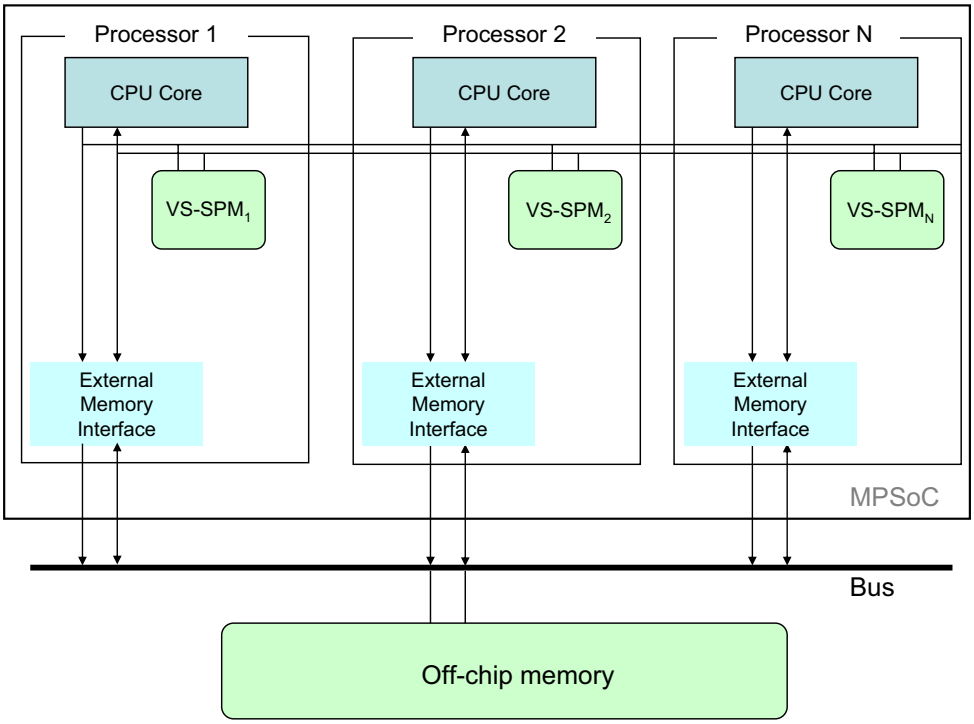


Figure 8.1: Embedded single-chip multiprocessor with virtually shared scratchpad memory

Architectural Model In this work, we focus on embedded single-chip multiprocessor architecture with scratchpad memory, as shown in Figure 8.1. The architecture contains multiple processor cores on chip. The cores can be homogeneous or heterogeneous. The processor cores communicate with the shared off-chip memory via a bus. In the single-chip multiprocessor setting, each processor core can access its *private scratchpad* as well as *remote scratchpads* attached to other processors. Such a setup, as described in [63], is called *virtually shared scratchpad memory (VS-SPM)*.

A processor core has a dedicated access link to its private scratchpad with minimum latency — usually a single cycle. Access to a remote scratchpad also incurs low latency

(*e.g.*, 4–16 cycles) due to the fast on-chip communication link among the processor cores. However, off-chip memory access has very high latency (*e.g.*, 100 cycles) due to the performance gap between processor and DRAM technology. Access conflicts between multiple processors may also arise in the bus, adding non-trivial variable delays to the off-chip memory accesses. For simplicity, in this work we assume that the latency incurred by every off-chip memory access is a constant. To avoid coherency issues, we also make the assumption that a memory location can be mapped to at most one scratchpad. The Cell processor [51] is an example of a real system with similar architecture even though its recommended programming model is somewhat different.

In this work, we focus on data scratchpad, but our strategy applies to code scratchpad as well. That is, our formulation can be easily configured for allocating general memory objects in the form of data variables or blocks of program code.

Task Graph We assume that the embedded application is specified as a task graph. The task graph is a directed acyclic graph that represents the key computation blocks (tasks) of an application as nodes and the communication between these tasks as edges. A task can be mapped to any of the processing cores. Therefore, associated with each task t are the execution times corresponding to running the task t on each of the processing cores. In case of homogeneous cores, there is only one execution time associated with each task. Note that for our problem, the execution time of a task t on a processor P depends on the placement of its data variables in the scratchpad. Therefore, we estimate the execution time assuming that all the data variables are accessed from the off-chip memory. An edge from task t to u in the task graph represents data transfer from t to u. Therefore, each edge is labeled with the amount of data transferred along it.

As memory hierarchy design is the main focus of this paper, each task is also associated with the sizes and access frequencies of data variables obtained through profiling. Note that the data access pattern of a task can be different depending on which processor it

gets mapped to (if the processors are heterogeneous). In that case, for each task we maintain multiple access patterns corresponding to the different processor cores.

Pipelined Scheduling Most streaming applications, such as multimedia and digital signal processing (DSP) applications, are iterative in nature. For these applications, the execution of the task graph is evoked repeatedly for a stream of input data. Hence, these applications are amenable to pipelined implementation for greater throughput [25]. The pipelined implementation benefits from allowing multiple processors execute multiple iterations of the task graph at the same time. We shall consider pipelined scheduling in our problem formulation. Note that the objective for sequential implementation is to minimize the execution time of a single iteration of the task graph. In contrast, the objective of pipelined implementation is to minimize the *initiation interval (II)*, which is the length of time between the start of two consecutive iterations of the task graph (see Figure 8.3). Minimizing the initiation interval results in optimal throughput for a streaming application.

Problem Statement Given a task graph, the architectural model and a bound on the total available on-chip SRAM, our goal is to find the optimal scratchpad configuration that results in minimum initiation interval (II). This problem can be decomposed into three sub-problems.

- *Mapping/scheduling* of the tasks to the processors as well as communication among the tasks
- *Scratchpad partitioning* to find the optimal size for each private scratchpad
- *Data allocation* of variables accessed by the tasks to the scratchpads

We present a flexible approach that explores the solution space of possible task mapping/scheduling, scratchpad configuration and data allocations together.

8.4 Method Illustration

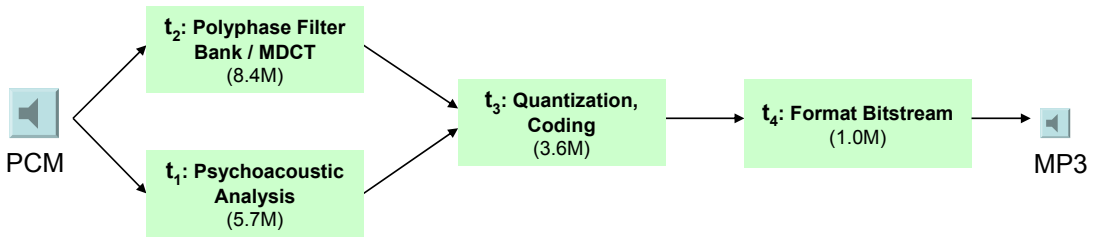


Figure 8.2: Task graph of LAME MP3 encoder

To illustrate the interaction among task scheduling, scratchpad partitioning and data allocation, we will use the task graph shown in Figure 8.2. The task graph corresponds to LAME MP3 encoder from the MiBench benchmark suite. It consists of four tasks and encodes a sequence of MP3 audio frames. Due to the task level parallelism, this application can take advantage of task pipelining as well as multiprocessing. The execution time of the tasks (assuming all the variables are located in off-chip memory) are obtained through profiling. The values (in cycles) are indicated as bracketed numbers in Figure 8.2. Our MPSoC architecture has two homogeneous on-chip processors and a total on-chip scratchpad budget of 4KB.

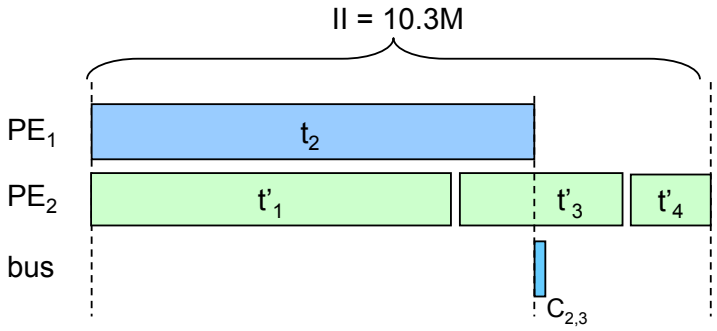


Figure 8.3: Optimal pipelined schedule for the task graph in Figure 8.2 without considering data allocation

Existing design space exploration strategies will first map and schedule the tasks and communication onto the two processors without considering the allocation of variables

to the scratchpads. Figure 8.3 shows the optimal pipelined schedule for the task graph in Figure 8.2. Non-primed labels indicate tasks/communications from the current instance, while primed labels indicate tasks/communications from the previous instance. This schedule can process one audio frame every 10.3M cycles, which is the initiation interval II for the schedule.

Now, we consider allocating the data variables to on-chip scratchpads. The common practice is to partition the total scratchpad budget equally between the two processors, that is, each processor has 2KB private scratchpad. The variables of task t_2 are allocated to the 2KB scratchpad of processor PE_1. Similarly, the variables of tasks t_1, t_3, and t_4 are allocated to the 2KB scratchpad of processor PE_2. We call this the ***EQ*** strategy, which stands for *equal partitioning* of the scratchpad. The allocation reduces the total execution time of t_1, t_3, t_4 to 8.2M cycles. Therefore, the II reduces to 8.2M cycles. However, we notice that the combined execution time of t_1, t_3, t_4 on PE_2 determines II in both cases (with or without allocation). That is, the reduction of task t_2's execution time does not have any effect on the global throughput of the application. This example indicates that allocating a bigger share of scratchpad to processor PE_2 would have been a better strategy.

We proceed to explore an integrated scratchpad partitioning and data allocation strategy. We call this the ***PF*** strategy, as it is a *partially flexible* strategy. Note that in this case the task mapping and scheduling have also been performed beforehand. As expected, this strategy allocates a larger scratchpad space to processor PE_2 and reduces II to 7.6M cycles. The 1152 bytes allocated to PE_1 is used to keep its execution time below this II value. Given the schedule shown in Figure 8.3, this is the optimal II achievable with 4KB on-chip scratchpad.

However, fixing the task schedule a-priori without considering the effect of data allocation on the execution time may miss the global optima. For example, task t_2 has the

longest execution time without data allocation and hence it is mapped onto a separate processor. However, its execution time may reduce considerably after data allocation and hence it may not be optimal to allocate this task on a separate processor. Exploring the design space of task scheduling, scratchpad partitioning and data allocation together could potentially reach a design point that is not possible through decoupled scheduling and scratchpad allocation.

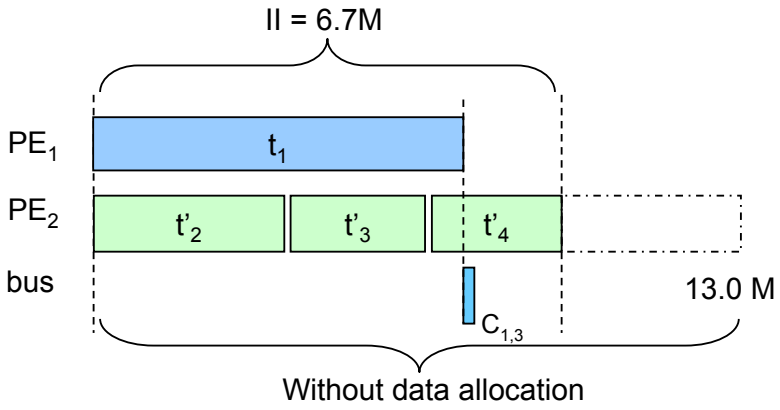


Figure 8.4: Optimal pipelined schedule for the task graph in Figure 8.2 through integrated task scheduling, scratchpad partitioning and data allocation

Therefore, we devise a flexible approach that essentially combines the task scheduling, scratchpad partitioning, and data allocation phases. We will call this the *completely flexible* (***CF***) strategy. Figure 8.4 shows the schedule produced by the CF strategy for the same task graph. This schedule is different from the schedule shown in Figure 8.3, which does not consider data allocation. In particular, task t_1 has been mapped to a separate processor instead of task t_2. A decoupled scheduling phase can never produce the schedule in Figure 8.4 as its performance without data allocation is extremely poor (II = 13M cycles). However, with data allocation, this schedule produces an optimal II of 6.7M cycles. Incidentally, the entire scratchpad space is allocated to processor PE_2.

8.5 Integer Linear Programming Formulation

In this section, we present the ILP formulation for an integrated task mapping/scheduling, scratchpad memory partitioning, and data allocation. We assume that the application is specified as a task graph. We first formulate the problem of scheduling the tasks on multiple processors without considering the presence of scratchpad. This formulation is then extended to handle the pipelined scheduling. Finally, we formulate the problem of scratchpad partitioning and data allocation and integrate it with the formulation for task scheduling.

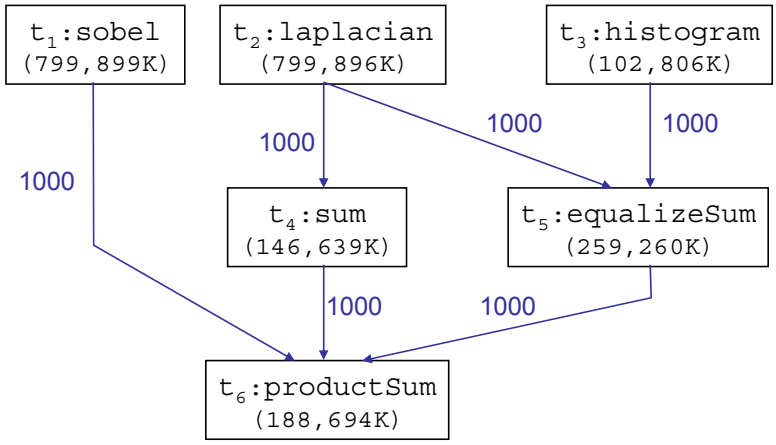


Figure 8.5: An example task graph

Throughout our discussion on task mapping and scheduling, we will use the task graph shown in Figure 8.5 for illustration purposes. The numbers in brackets indicate the execution time of the tasks (in cycles). The edges are labeled with the communication costs (in cycles) between the tasks. For simplicity of illustration, we assume an MPSoC architecture consisting of four homogeneous processors so that the execution time of a task is the same across all processors. However, we note that our formulation handles heterogeneous processors as well.

8.5.1 Task Mapping/Scheduling

We present here an ILP formulation to optimize performance through integrated task mapping and scheduling. We shall present extensions for pipelined scheduling and scratchpad partitioning in the next subsections.

Setting The task graph for an application has N tasks denoted as $t_1, \ldots, t_N$. Without loss of generality, let t_N be the last task (the task without successors) in the task graph. If there are multiple last tasks in the task graph, then we add a dummy last task as the successor of all the original last tasks. We have Q available processors (homogeneous or heterogeneous) denoted as $PE_1, \ldots, PE_Q$. Associated with each task is its execution time on each of the available processors — $time_{i,j}$ denotes the execution time of task t_i on processor PE_j, assuming that all the data variables are available in off-chip memory (no scratchpad data allocation is considered at this point).

Let the binary decision variable $X_{i,j} = 1$ if task t_i is mapped to processor PE_j and 0 otherwise. A task can be mapped to exactly one processor.

$$\sum_{j=1}^{M} X_{i,j} = 1 \tag{8.1}$$

The execution time of task t_i is given by

$$Time_i = \sum_{j=1}^{M} X_{i,j} \times time_{i,j} \tag{8.2}$$

Let $StartTask_i$ and $EndTask_i$ denote the starting time and the completion time, respectively, of task t_i. Then

$$EndTask_i = StartTask_i + Time_i - 1 \tag{8.3}$$

Objective Function The objective is to minimize the critical path through the task graph, that is, to minimize the completion time $EndTask_N$ of the last task t_N.

Task Dependencies Let $preds(t_i)$ denote the set of predecessors of task t_i in the task graph. t_i can only start execution after all the tasks $t_h \in preds(t_i)$ have completed execution. Further, if t_i and t_h are mapped to two different processors, then t_i has to wait for the completion of any data transfer from t_h to itself, incurring a communication cost $comm_{h,i}$.

We model inter-task communications as special tasks running on a shared bus. Let $C_{h,i}$ be the communication task between t_h and t_i, and let $StartComm_{h,i}$ and $EndComm_{h,i}$ be the starting time and the completion time of $C_{h,i}$. Then we have the following constraints.

$$StartComm_{h,i} \geq EndTask_h + 1 \tag{8.4}$$

$$StartTask_i \geq EndComm_{h,i} + 1 \tag{8.5}$$

Note that task dependencies are indirectly enforced via the communications between the tasks. To reflect the fact that the communication cost between t_h and t_i is incurred only when t_i and t_h are mapped to different processors, we have the following constraint.

$$EndComm_{h,i} = StartComm_{h,i} + L_{h,i} \times comm_{h,i} - 1 \tag{8.6}$$

where $L_{h,i} = 1$ if and only if t_h and t_i are mapped to different processors. Recall that $X_{i,j} = 1$ if task t_i is mapped to processor PE_j and 0 otherwise. The definition of $L_{h,i}$ is linearized as follows.

$$\forall j : 1 \ldots M \quad L_{h,i} \leq 2 - X_{h,j} - X_{i,j}$$

$$\forall j : 1 \ldots M \quad \forall k : 1 \ldots M, k \neq j \quad L_{h,i} \geq X_{h,j} + X_{i,k} - 1$$

Resource Constraint The previous constraints effectively prevent two dependent tasks from competing for processor time. We should also ensure that any two *independent* tasks mapped to the same processor have disjoint lifetimes. Mirroring our treatment of dependent tasks, for every pair of independent tasks t_i and $t_{i'}$, let the binary variable $L_{i,i'} = 1$ if and only if t_i and $t_{i'}$ are mapped to different processors. If $L_{i,i'} = 0$, either task t_i executes before $t_{i'}$ or vice versa. Let binary variable $B_{i',i} = 0$ if t_i and $t_{i'}$ are mapped to the same processor and $t_{i'}$ executes after t_i. Similarly let $B_{i,i'} = 0$ if t_i and $t_{i'}$ are mapped to the same processor and t_i executes after $t_{i'}$. Then we ensure disjoint lifetime for the two tasks through the following constraints.

$$B_{i,i'} + B_{i',i} - L_{i,i'} = 1 \tag{8.7}$$

$$StartTask_i \geq EndTask_{i'} - \infty \times B_{i,i'} + 1 \tag{8.8}$$

$$StartTask_{i'} \geq EndTask_i - \infty \times B_{i',i} + 1 \tag{8.9}$$

As all the communications take place on a shared bus, we should also ensure that the communications do not overlap with each other. Analogous to constraints (8.7) through (8.9), for all pairs of distinct communication tasks $C_{h,i}$ and $C_{f,g}$, we have the following constraints.

$$V_{h,i,f,g} + V_{f,g,h,i} = 1 \tag{8.10}$$

$$StartComm_{h,i} \geq EndComm_{f,g} - \infty \times V_{h,i,f,g} + 1 \tag{8.11}$$

$$StartComm_{f,g} \geq EndComm_{h,i} - \infty \times V_{f,g,h,i} + 1 \tag{8.12}$$

Here binary variable $V_{h,i,f,g} = 1$ if $C_{f,g}$ happens after $C_{h,i}$ and 0 otherwise. Similarly, binary variable $V_{f,g,h,i} = 1$ if $C_{h,i}$ happens after $C_{f,g}$ and 0 otherwise.

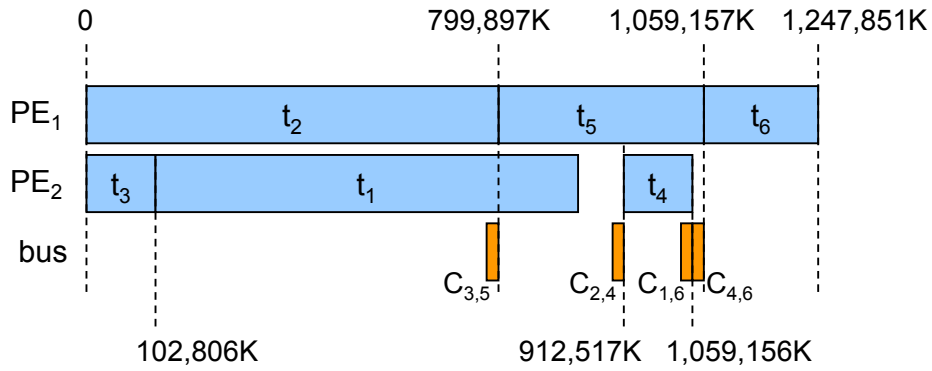


Figure 8.6: An optimal non-pipelined schedule for the task graph in Figure 8.5 on four processors

Example Figure 8.6 shows an optimal schedule for the task graph in Figure 8.5 on four processors, obtained using the ILP formulation. Note that only two out of four available processors are utilized: processors PE_3 and PE_4 are not assigned any task. This is because utilizing more processors will increase the length of the critical path through additional communication costs. We assume that computation and communication can proceed in parallel (*e.g.*, execution of task t_1 on processor PE_2 in parallel with communication $C_{3,5}$ from PE_2 to PE_1). Also note that the tasks on processor PE_1 determine the critical path; therefore, tasks on processor PE_2 are scheduled with slacks.

8.5.2 Pipelined Scheduling

In this section, we extend the task scheduling formulation in the previous subsection to take into account pipelined scheduling. In a synchronous pipelined execution, tasks are distributed across pipeline stages of uniform length. The length of the pipeline stages, known as the *initiation interval* (II), is determined by the maximum time needed to complete all the tasks in any of the stages. Thus the objective of pipelined scheduling is to distribute tasks into stages so as to minimize the II, while respecting task dependencies and resource constraints.

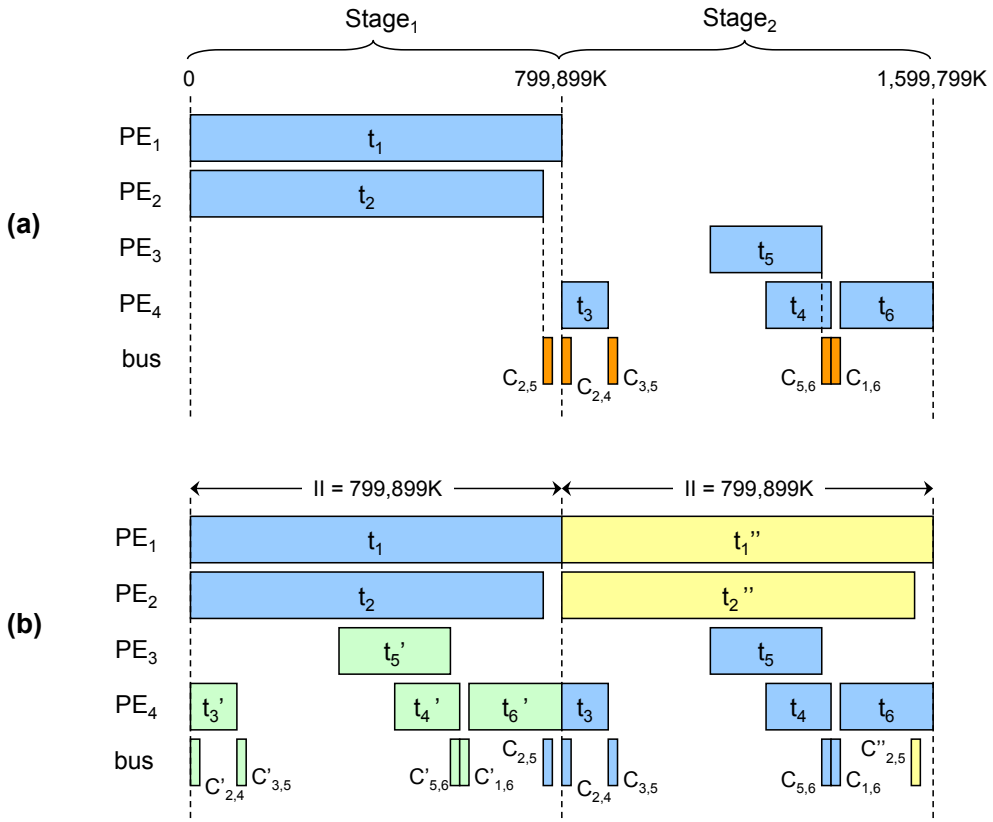


Figure 8.7: An optimal pipelined schedule for the task graph in Figure 8.5 with (a) single-instance execution view, and (b) steady-state execution view

Example Figure 8.7 shows an optimal pipelined schedule for the task graph in Figure 8.5 on four processors. Recall that non-primed labels indicate tasks/communications from the current instance, primed labels indicate tasks/communications from the previous instance, and double-primed labels indicate tasks/communications from the next instance. Comparing this pipelined schedule with the non-pipelined schedule in Figure 8.6, we can notice the difference in the objective function. The time taken to execute a single instance of the task graph increases from 1,248M cycles for non-pipelined execution to 1,600M cycles for pipelined execution (Figure 8.7a). However, in the steady state the pipelined execution can process a task graph every 800M cycles (Figure 8.7b), which is the II for this schedule. That is, the throughput increases significantly compared to the non-pipelined execution.

An important constraint that becomes apparent here is that any processor can be exploited in exactly one pipeline stage, because all the stages will execute in parallel (processing different task instances) in the steady state. On the other hand, a stage can exploit more than one processor. This also implies that the maximum number of pipeline stages we can have is equal to the number of processors, Q. Based on this observation, we extend our previous formulation to solve the problem of pipelined scheduling. Our strategy is to perform task mapping and scheduling onto processors as before, and then assign processors to the pipeline stages. Communication tasks, which are scheduled on a shared bus, will be mapped directly to pipeline stages as we will elaborate later.

Let the binary variable $W_{j,s} = 1$ if processor PE_j is assigned to s^{th} pipeline stage (denoted $Stage_s$). Each processor is mapped to exactly one pipeline stage.

$$\sum_{s=1}^{M} W_{j,s} = 1 \tag{8.13}$$

Note that in the summation term we have implicitly defined the number of pipeline stages to be Q, the maximum allowed number. This is necessary to keep the formulation linear. The solution may contain stages which have no processor assigned to them (that is, one of the other stages may have more than one processor assigned to it); these are "invalid" stages which will be disregarded.

Objective Function The objective function is to minimize the Initiation Interval II, which is determined by the longest time needed to complete all the tasks in any of the stages. Let $StartStage_s$ and $EndStage_s$ denote the starting and completion time, respectively, of $Stage_s$. Then:

$$II \geq EndStage_s - StartStage_s + 1 \quad \forall s : 1 \ldots M \tag{8.14}$$

Overlap among Pipeline Stages A pipeline stage must not overlap with another stage. Analogous to constraints (8.7) through (8.9), for all pairs of stages $Stage_s$ and $Stage_t$, we have the following constraints.

$$B'_{s,t} + B'_{t,s} = 1 \tag{8.15}$$

$$StartStage_s \geq EndStage_t - \infty \times B'_{s,t} + 1 \tag{8.16}$$

$$StartStage_t \geq EndStage_s - \infty \times B'_{t,s} + 1 \tag{8.17}$$

Here $B'_{s,t} = 1$ if $Stage_t$ executes after $Stage_s$ and 0 otherwise. Similarly $B'_{t,s} = 1$ if $Stage_s$ executes after $Stage_t$ and 0 otherwise.

Length of Pipeline Stages Now we express the constraints on the length of each pipeline stage. The length of $Stage_s$ has to encompass the entire execution period of the processor(s) assigned to it. For all pipeline stages $s : 1 \ldots Q$ and all processors $j : 1 \ldots Q$ we require:

$$StartStage_s \leq StartProc_j + \infty \times (1 - W_{j,s}) \tag{8.18}$$

$$EndStage_s \geq EndProc_j - \infty \times (1 - W_{j,s}) \tag{8.19}$$

where $StartProc_j$ and $EndProc_j$ denote the starting and completion time of processor PE_j's execution, which are in turn determined by the earliest start time and the latest end time over all the tasks mapped to processor PE_j. For example, the execution time of processor PE_4 in Figure 8.7a spans from the start of task t_3 until the completion of task t_6. These terms are related to the task mapping constraint as follows. Recall that the decision variable $X_{i,j}$ has value 1 if t_i is scheduled on PE_j and 0 otherwise.

For all processors $j : 1 \ldots Q$ and all tasks $i : 1 \ldots N$:

$$StartProc_j \leq StartTask_i + \infty \times (1 - X_{i,j}) \tag{8.20}$$

$$EndProc_j \geq EndTask_i - \infty \times (1 - X_{i,j}) \tag{8.21}$$

Overlap among Communication Tasks Communication tasks must also be accounted for in the pipeline stages. Unlike normal tasks, all the communications take place on a shared bus, *which will be utilized in all the stages*. As illustrated in Figure 8.7b, communications from different pipeline stages (*i.e.*, from different instances of the task graph) execute simultaneously within an II. Constraints 8.10 through 8.12 only ensure that the communication tasks within a single stage (*i.e.*, from the same instance of the task graph) do not overlap with each other. However, we now need to ensure that the communication tasks do not overlap *across stages*.

This is accomplished by first *normalizing* the execution intervals of the communication tasks. The normalized interval of a communication task is its execution interval (start time to completion time) relative to the start time of the pipeline stage that it is mapped onto. For example, the interval of communication task $C_{2,4}$ in Figure 8.7(a) is $[799,899K, 799,900K]$, while its normalized interval is $[0, 1000]$, *i.e.*, relative to the start of $Stage_2$ at $799,899K$.

Let us define a binary variable $F_{h,i,s} = 1$ if $C_{h,i}$ is mapped to stage $Stage_s$ and 0 otherwise.

$$\sum_{s=1}^{M} F_{h,i,s} = 1 \tag{8.22}$$

Each communication task is then included in the interval of the stage it is mapped to.

$$StartStage_s \leq StartComm_{h,i} + \infty \times (1 - F_{h,i,s}) \tag{8.23}$$

$$EndStage_s \geq EndComm_{h,i} - \infty \times (1 - F_{h,i,s}) \tag{8.24}$$

Finally, we require mutual exclusion for all pairs of distinct communication tasks $C_{h,i}$ and $C_{f,g}$:

$$\begin{aligned} &(StartComm_{h,i} - StartStage_s) \\ &\geq (EndComm_{f,g} - StartStage_t) - \infty \times V'_{h,i,s,f,g,t} + 1 \end{aligned} \tag{8.25}$$

$$\begin{aligned} &(StartComm_{f,g} - StartStage_t) \\ &\geq (EndComm_{h,i} - StartStage_s) - \infty \times V'_{f,g,t,h,i,s} + 1 \end{aligned} \tag{8.26}$$

where $V'_{h,i,s,f,g,t} = 0$ ($V'_{f,g,t,h,i,s} = 0$) if and only if $C_{h,i}$ is scheduled in $Stage_s$, $C_{f,g}$ is scheduled in $Stage_t$, and the normalized interval of $C_{h,i}$ is scheduled after (before) the normalized interval of $C_{f,g}$. These auxiliary variables are related via the following linearization.

$$V'_{h,i,s,f,g,t} + V'_{f,g,t,h,i,s} + F_{h,i,s} \geq 2$$

$$V'_{h,i,s,f,g,t} + V'_{f,g,t,h,i,s} + F_{f,g,t} \geq 2$$

$$V'_{h,i,s,f,g,t} + V'_{f,g,t,h,i,s} + F_{h,i,s} + F_{f,g,t} \leq 3$$

8.5.3 Scratchpad Partitioning and Data Allocation

We now present the ILP formulation for the scratchpad partitioning and data allocation problem. Let the total number of variables for all the tasks be R. Some variables may be shared by several tasks. Associated with each variable v are its size in bytes, denoted $area_v$, and the number of times it is accessed in each task, obtained through profiling. The latter value may vary depending on which processor the task gets mapped to (due

to difference in ISA as well as compiler). As such, let $freq_{v,i,j}$ specify the number of accesses of variable v by task t_i when executing on processor PE_j.

Each access to the variable v incurs different latencies depending on the location of v:

- 0, if v is located in the private scratchpad of PE_j, or
- a constant latency of $cross\_penalty$, if v is located in scratchpad of another processor (remote scratchpad), or
- a constant latency of $penalty$, if v is located in the off-chip memory

The value of $cross\_penalty$ will generally be less than $penalty$.

Let the binary decision variable $S_{v,j} = 1$ if variable v is allocated in the scratchpad of processor PE_j and 0 otherwise. In our architectural model, a variable can be mapped to at most one processor's scratchpad.

$$\sum_{j=1}^{M} S_{v,j} \leq 1 \tag{8.27}$$

We have a constraint on the total available scratchpad area. Let $total\_area$ be the total available scratchpad area given as input to this problem.

$$\sum_{v=1}^{R} \sum_{j=1}^{M} S_{v,j} \times area_v \leq total\_area \tag{8.28}$$

Objective Function The objective of task scheduling with scratchpad configuration and data allocation is also to minimize the overall completion time $EndTask_N$ where t_N is the last task, or in the pipelined setting, the initiation interval II, where II is specified in Constraint 8.14.

Task Execution Time The only effect of data allocation to on-chip memory is that the execution time of each task is potentially reduced. Previously, each task takes constant execution time — $time_{i,j}$ denotes the execution time of task t_i on processor PE_j assuming all the variables are in off-chip memory. In that case, the execution time of a task is given by Equation 8.2. Now we replace this equation with

$$Time_i = \sum_{j=1}^{M} \left(X_{i,j} \times time_{i,j} - \sum_{v=1}^{R} freq_{v,i,j} \times gain_{v,i,j} \right) \tag{8.29}$$

$$gain_{v,i,j} = Y_{v,i,j} \times penalty + Z_{v,i,j} \times (penalty - cross\_penalty) \tag{8.30}$$

where $Y_{v,i,j} = 1$ if and only if task t_i and variable v have both been mapped to processor PE_j; and $Z_{v,i,j} = 1$ if and only if t_i has been mapped to processor PE_j and variable v has been mapped to the scratchpad of a processor other than PE_j. In other words,

$$Y_{v,i,j} = 1 \Leftrightarrow ((X_{i,j} = 1) \quad AND \quad (S_{v,j} = 1))$$

$$Z_{v,i,j} = 1 \Leftrightarrow ((X_{i,j} = 1) \quad AND \quad (\exists k \neq j \quad S_{v,k} = 1))$$

The above two constraints can be linearized as follows.

$$Y_{v,i,j} \leq X_{i,j}; \;\; Y_{v,i,j} \leq S_{v,j}$$

$$Y_{v,i,j} \geq X_{i,j} + S_{v,j} - 1$$

For the second constraint, we need to introduce an additional binary variable $U_{v,j} = 1$ iff $\exists k \neq j \;\; S_{v,k} = 1$. We first linearize the definition of $U_{v,j}$.

$$\sum_{k=1,k\neq j}^{M} S_{v,k} - \infty \times U_{v,j} \leq 0;$$

$$\sum_{k=1,k\neq j}^{M} S_{v,k} - U_{v,j} \geq 0$$

Then we linearize the original constraint in terms of $U_{v,j}$.

$$Z_{v,i,j} \leq X_{i,j}; \quad Z_{v,i,j} \leq U_{v,j}; \quad Z_{v,i,j} \geq X_{i,j} + U_{v,j} - 1$$

Deadline The discussion so far focuses on obtaining the optimal overall runtime or II given the available scratchpad area. We can also modify the formulation to find the optimal scratchpad area/configuration given a deadline on the execution time or II. Instead of the constant $total\_area$ we now have the total memory area as a variable, denoted as $TotalArea$, which we want to minimize. The constraint on total scratchpad area given by Equation (8.28) is replaced with

$$TotalArea = \sum_{v=1}^{R} \sum_{j=1}^{M} S_{v,j} \times area_v \tag{8.31}$$

The deadline on execution time is imposed by adding the following constraint.

$$EndTask_N \leq deadline \tag{8.32}$$

where $deadline$ is the given deadline, a constant.

8.6 Experimental Evaluation

The ILP formulation for integrated pipelined task scheduling, scratchpad partitioning and data allocation generates the optimal solution. The goal of our ILP formulation is to explore the interaction among the different stages of the design space exploration process. This helps us to identify the performance limit of MPSoC architectures for embedded applications.

As explained in Section 8.4, we attempt three different techniques for optimal data allocation to scratchpad memory in increasing order of flexibility and complexity.

EQ: Task scheduling ignores the effect of data allocation to scratchpad, and the on-chip scratchpad budget is equally partitioned among the different processors. This is simply a knapsack problem, for which optimal solutions (*e.g.*, dynamic programming) are known.

PF: Task scheduling ignores the effect of data allocation to scratchpad. However, scratchpad partitioning and data allocation are performed simultaneously through a simplified ILP formulation derived from Section 8.5. As task scheduling is performed a-priori, the assignments to the ILP variables X, B, V, W, F, and V' are known. In other words, the design space is more restricted.

CF: Task scheduling, scratchpad partitioning and data allocation are performed simultaneously through the ILP formulation described in Section 8.5.

Setup We use five applications in our experiments, four of which are taken from embedded benchmark suites MiBench [44] and Mediabench [71]. `cjpeg`, `mpeg2enc` and `osdemo` from Mediabench perform JPEG encoding, MPEG-2 encoding and 3D graphics rendering (part of Mesa 3D graphics library), respectively. `lame` is an MP3 encoder from the MiBench consumer suite. `enhance`, the fifth benchmark, is a slightly modified version of the image enhancement application from [122].

We profile the applications to identify the key computation blocks. Each application is then divided into a number of tasks where each task corresponds to a computation block. The control/data flow information are used to identify the dependencies among the tasks and estimate the communication costs. This process generates the task graph for each application.

We use the SimpleScalar cycle-accurate architectural simulation platform [12] for our experiments. An instrumented version of the SimpleScalar profiler extracts the variable sizes and access frequencies as well as the execution time (in processor cycles) of each task individually. As mentioned earlier, the profiler assumes that off-chip access latencies are constant, and does not account for bus conflicts.

We consider both scalar and array variables for allocation to scratchpad memories. All the shared variables (*i.e.*, variables that are written by one task and read by another) contribute towards the communication cost in the task graph. These shared variables are not considered for allocation into the scratchpad memories.

Table 8.1 shows the characteristics of the benchmarks. The profile is shown only for the non-shared (*i.e.*, non-communicating) variables. In this work, we choose data variables for allocation, but our strategy can be applied to blocks of program code as well.

Table 8.1: Benchmark characteristics

Benchmark	# Tasks	# Variables	Variable Size (bytes)	
			Total	Average
`enhance`	6	45	7,355,920	163,464
`lame`	4	124	301,672	2,432
`mpeg2enc`	6	30	12,744	424
`osdemo`	6	45	80,000	1,777
`cjpeg`	4	20	702,296	35,114

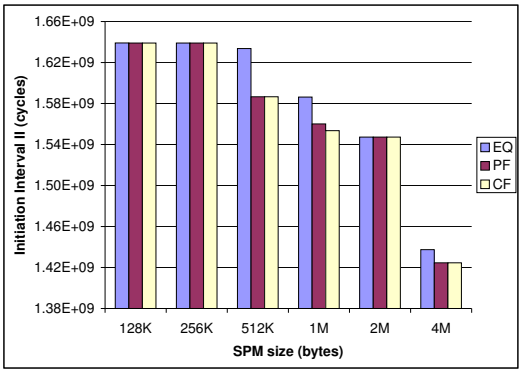
We use a 2-processor configuration for the experiments. Total on-chip scratchpad memory budget varies from 256 bytes to 4MB for different benchmarks. We give smaller scratchpad budgets for applications with smaller total memory requirement (*e.g.*, `mpeg2enc`) and larger scratchpad budget for applications with large average variable size (*e.g.*, `enhance` with 163KB). We assume 100-cycle latency for off-chip memory access and 4-cycle latency for accessing a remote scratchpad.

For the ILP-based techniques, our method constructs the ILP formulation for (separate or integrated) task scheduling and data allocation as described in Section 8.5 and inputs

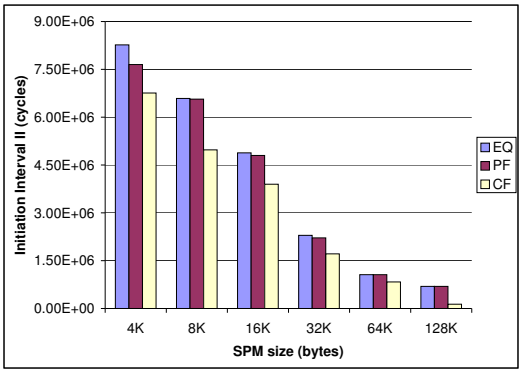
this to CPLEX [29], a commercial ILP solver. Upon solving the ILP problem, CPLEX returns the objective value as well as the valuation of the decision variables that leads to the objective. The experiments are conducted on a 3.0 GHz Pentium 4 CPU with 2GB of memory.

Results and Discussion Figure 8.8 shows the initiation interval (II) obtained by applying *EQ*, *PF*, and *CF* techniques on the five benchmarks with varying scratchpad sizes. First let us compare *EQ* with *PF*. The main advantage of *PF* is that it allows flexible partitioning of scratchpad space among the processors. This flexibility can potentially improve the performance dramatically over *EQ*. For example, *PF* results in up to 35% performance improvement for `mpeg2enc`, 53% improvement for `osdemo`, and 60% improvement for `cjpeg` over *EQ*. `lame` enjoys a modest improvement of up to 7.5% due to the flexible scratchpad allocation. The only exception is `enhance`, which achieves very little improvement due to *PF*. This is because `enhance` has large variables that are harder to allocate. This is also clear from the fact that the II hardly improves with increasing scratchpad sizes.

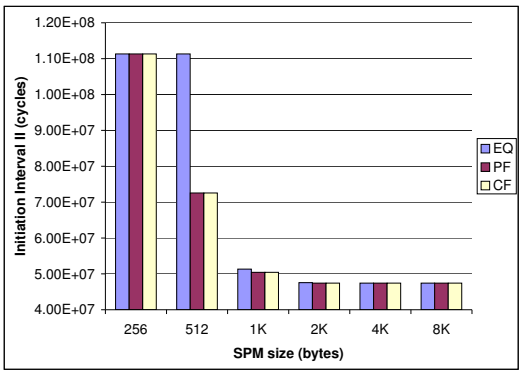
It is important to note that flexibility is most important when resources are not too limited or too generous. With a restricted scratchpad budget, there is not much room for improvement irrespective of the scratchpad partitioning strategy. Similarly, when the scratchpad budget is bigger than the one necessary to accommodate all the frequently accessed variables, the strategy employed for scratchpad partitioning becomes immaterial. With a larger scratchpad budget, *PF* allocates more variables than *EQ*; but these variables are accessed less frequently. The *PF* strategy shows maximum improvement when the on-chip scratchpad budget is neither too big nor too small. In those cases, only the most important variables should be accommodated and the flexibility guarantees that the most important variables can indeed be allocated.



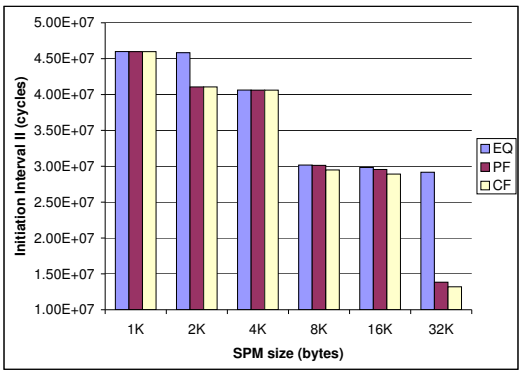
(a) enhance



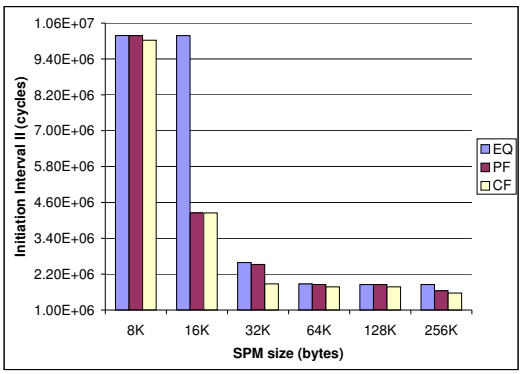
(b) lame



(c) mpeg2enc



(d) osdemo



(e) cjpeg

Figure 8.8: Initiation interval (II) for the different benchmarks with *EQ*, *PF*, and *CF* strategies given varying on-chip scratchpad budgets on a 2-processor configuration

We now compare *PF* with *CF*. The results indicate that the improvements depends heavily on the characteristics of the applications. For example, `lame` achieves as high as 80% improvement over *PF* by considering data allocation during scheduling. This is highlighted in Figure 8.9, which shows the performance improvement of *PF* and *CF* over *EQ*. Similarly, `cjpeg` enjoys up to 25% additional performance improvement. `osdemo` shows 2–5% improvement. `enhance` and `mpeg2enc`, on the other hand, show hardly any improvement. As previously explained, `enhance` has larger variables that are difficult to allocate, while `mpeg2enc` is a compute-intensive application with significant amount of communication among the tasks. The results shown in Table 8.1 imply that efficient use of scratchpad space is not so important in these cases because non-shared variable accesses do not contribute significantly towards the execution time.

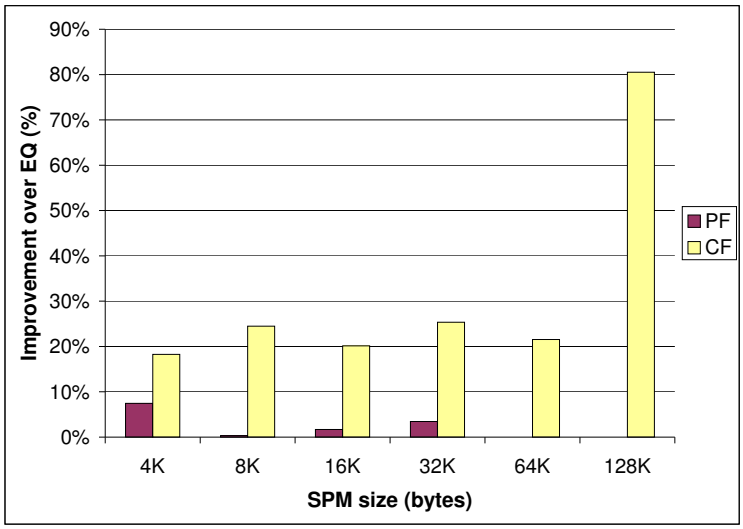


Figure 8.9: Improvement in initiation interval (II) due to *PF* and *CF* over *EQ* for benchmark `lame`

Finally, Table 8.2 shows the runtime for our scheduling and allocation techniques. *Scheduling* denotes the task scheduling time irrespective of scratchpad consideration. This is required as input to *EQ* and *PF*. The data allocation time for *EQ* as well as scratchpad partitioning and data allocation time for *PF* assuming a given task schedule generated a-priori are shown in the next two columns. Finally, the column `CF` gives the runtime for integrated task scheduling, scratchpad partitioning and data allocation. We show both

the best-case runtime and worst-case runtime for each benchmark.

Table 8.2: Best-case and worst-case algorithm runtimes for the benchmarks

Benchmark	Best Runtime (sec)				Worst Runtime (sec)			
	Scheduling	**EQ**	**PF**	**CF**	Scheduling	**EQ**	**PF**	**CF**
`enhance`	12.9	0.01	0.01	23.60	13.07	0.32	0.06	20+ mins
`lame`	0.20	0.03	0.04	15.26	0.22	0.81	0.52	20+ mins
`mpeg2enc`	1.56	0.01	0.01	3.75	6.69	0.02	0.07	69.98
`osdemo`	2.81	0.02	0.01	105.16	2.86	0.04	0.06	1467.33
`cjpeg`	0.33	0.01	0.01	0.55	0.37	0.04	0.05	3.6

We observe that the worst-case occurs when the scratchpad budget is neither too big nor too small. This is expected as these cases are the most difficult ones to schedule. Note that for `enhance` with 4MB scratchpad budget and `lame` with 16KB, 128KB scratchpad budget, the ILP solver did not terminate within 20 minutes (shown as 20+ min in the table). For these cases, we use the intermediate solution returned by the ILP solver. Interestingly, even the intermediate solutions obtained for *CF* strategy outperforms the optimal result obtained using *PF* strategy (Figure 8.8). This clearly indicates that it is important to consider memory optimization during task scheduling.

8.7 Chapter Summary

We have explored optimization of scratchpad memory in the context of embedded chip multiprocessors. We propose an ILP formulation to capture the interaction between task scheduling and memory optimization. Our evaluation shows that flexible partitioning of the scratchpad budget among the processors can achieve up to 60% performance improvement compared to equal partitioning. Further, integrating memory optimization with task scheduling can improve performance by up to 80%.

Chapter 9

Conclusion

9.1 Thesis Contributions

Real-time systems require that applications meet their timing specifications first and foremost. The main objective of optimization efforts in such systems is thus the system performance in the worst case rather than in the average case. The timing constraints impose inevitable limits to existing methods that are mainly targeted at average-case performance optimizations, and introduce new concerns that have to be accounted for in order for the optimization to be truly effective.

This thesis considers the problem of enhancing the worst-case performance of real-time applications through memory optimizations, which include caches and scratchpad memories. We propose techniques that preserve timing predictability in the interest of meeting real-time constraints, while optimizing the system performance in terms of the worst-case response time of the applications. We also consider interaction between memory optimization techniques and multiprocessing aspects, including task interaction and the impact of scheduling decisions.

The concrete contributions of this thesis are:

- scratchpad allocation techniques specifically targeted at improving the worst-case performance of the application
- scratchpad allocation techniques that improve the worst-case response time in the presence of process interaction and preemptions
- general guidelines and detailed performance evaluation of shared cache management schemes that preserve timing predictability
- integrated scratchpad allocation and task scheduling for multiprocessors
- a timing analysis method that incorporates the effect of scratchpad allocation with enhanced accuracy

9.2 Future Directions

The embedded computing world is undoubtedly moving in the direction of multiprocessing, which opens a whole new set of dimensions to explore in terms of performance enhancement. The interactions among these dimensions often produce non-trivial effects on the end result of optimization efforts. Most systems rely on simulation for an estimate of their deliverance. This is certainly not strict enough for hard real-time requirements.

Our thesis has looked at pairwise combinations of several of these dimensions in analysis, namely scratchpad memory management, process interactions, and task scheduling. While a complete analysis that takes into account all available multiprocessing aspects is expectedly too complex to be feasible, it is still instructive to first identify subsets

that relate closely to the characteristics of the application at hand, then attempt an integrated approach that builds on known time-predictable techniques for the components. We envision that researches along this direction will prove invaluable to the future of embedded real-time software given the growing demands for enhanced user experience.

Bibliography

[1] T. A. AlEnawy and H. Aydin. Energy-aware task allocation for rate monotonic scheduling. In *Proc. 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. *Lecture Notes in Computer Science*, 1145:52–66, 1996.

[3] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 1996.

[4] R. Alur and M. Yannakakis. Model checking message sequence charts. In *Proc. International Conference on Concurrency Theory (CONCUR)*, 1999.

[5] Analog Devices, Inc. Blackfin Processor. Available on: `http://www.analog.com/processors/processors/blackfin/`, 2006.

[6] Analog Devices, Inc. TigerSHARC Processor. Available on: `http://www.analog.com/processors/processors/tigersharc/`, 2006.

[7] J. H. Anderson, J. M. Calandrino, and U. C. Devi. Real-time scheduling on multicore platforms. In *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2006.

[8] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2004.

[9] ARM Ltd. White Paper: Architecture and Implementation of the ARM Cortex-A8 Processor. Available on: `http://www.arm.com/pdfs/TigerWhitepaperFinal.pdf`, 2005. Release October 2005.

[10] ARM Ltd. ARM Processor Cores Documentation. Available on: `http://www.arm.com/documentation/ARMProcessor_Cores/index.html`, 2006.

[11] A. Arnaud and I. Puaut. Dynamic instruction cache locking in hard real-time systems. In *Proc. 14th International Conference on Real-Time and Network Systems (RNTS)*, 2006.

[12] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[13] O. Avissar, R. Barua, and D. Stewart. Heterogeneous memory management for embedded systems. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

[14] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.

[15] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Comparison of cache- and scratch-pad-based memory systems with respect to performance, area and energy consumption. Technical Report 762, University of Dortmund, September 2001.

[16] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proc. International Conference on Hardware/Software Codesign (CODES)*, 2002.

[17] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, 1993.

[18] L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation and scheduling for MPSoCs via decomposition and no-good generation. In *Proc. International Joint Conferences on Artificial Intelligence (IJCAI)*, 2005.

[19] J. Brown. Application-customized CPU design: The Microsoft Xbox 360 CPU story. Available on: `http://www-128.ibm.com/developerworks/power/`

library/pa-fpfxbox/?ca=dgr-lnxw07XBoxDesign, 2005. Release Dec 6, 2005.

[20] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3), 1991.

[21] F. Burns, A. Koelmans, and A. Yakovlev. Wcet analysis of superscalar processors using simulation with coloured petri nets. *Real-Time Systems*, 18(2-3):275–288, 2000.

[22] A. M. Campoy, I. Puaut, A. P. Ivars, and J. V. B. Mataix. Cache contents selection for statically-locked instruction caches: an algorithm comparison. In *Proc. 17th Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[23] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman Hall/CRC Press, 2004.

[24] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2006.

[25] K. S. Chatha and R. Vemuri. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Transactions on VLSI*, 10(3), 2002.

[26] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[27] D. T. Chiou. *Extending the reach of microprocessors: column and curious caching*. PhD thesis, MIT, 1999.

[28] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2–3):249–274, May 2000.

[29] CPLEX. The ILOG CPLEX Optimizer v7.5, 2002. Commercial software, http://www.ilog.com.

[30] Ctrl Computer Systems. Network bookcase, 2000. http://www.bookcase.com/library/software/msdos.devel.lang.c.html.

[31] C. Cullmann and F. Martin. Data-flow based detection of loop bounds. In *Proc. 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[32] J.-F. Deverge and I. Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Proc. 19th Euromicro Conference on Real-Time Systems (ECRTS)*, 2007.

[33] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 2005.

[34] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S.L. Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2006.

[35] B. Egger, J. Lee, and H. Shin. Dynamic scratchpad memory management for code in portable systems with an MMU. *ACM Transactions on Embedded Computing Systems*, 7(2), 2008.

[36] A. Ermedahl and J. Engblom. Modeling complex flows for worst-case execution time analysis. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 2000.

[37] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proc. 3rd International Euro-Par Conference on Parallel Processing (Euro-Par)*, 1997.

[38] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proc. 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[39] European Space Agency. DEBIE – First standard space debris monitoring instrument, 2008. http://gate.etamax.de/edid/publicaccess/debie1.php.

[40] H. Falk and M. Verma. Combined data partitioning and loop nest splitting for energy consumption minimization. In *Proc. 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2004.

[41] Freescale Semiconductor, Inc. MMC2114/MMC2113 M-CORE Microcontroller Product Brief. Available on: `http://www.freescale.com/files/32bit/doc/prod_brief/MMC2114PB.pdf`, 2008.

[42] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[43] J. Gustafsson and A. Ermedahl. Merging techniques for faster derivation of wcet flow information using abstract execution. In *Proc. 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2008.

[44] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE Annual Workshop on Workload Characterization (WWC)*, 2001.

[45] D. Harel and P. S. Thiagarajan. Message sequence charts. *UML for real: Design of embedded real-time systems*, pages 77–105, 2003.

[46] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. V. Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2-3):129–156, 2000.

[47] C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan 1999.

[48] C. A. Healy and D. B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28(8), 2002.

[49] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface, 2nd Ed.* Morgan Kaufmann Publishers Inc., 1998.

[50] T. A. Henzinger, R. Jhala, R. Majumder, and G. Sutre. Lazy abstraction. In *Proc. Symposium on Principles of Programming Languages (POPL)*, 2002.

[51] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[52] IBM Systems and Technology Group. Cell Broadband Engine Architecture Version 1.0. Available on: `http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA/\$file/CBEA_01_pub.pdf`, 2005. Release Aug 8, 2005.

[53] IBM Systems and Technology Group. PowerPC: IBM Microelectronics. Available on: `http://www-306.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC`, 2006.

[54] Intel Corporation. Intel Multi-core. Available on: `http://www.intel.com/multi-core/`, 2006.

[55] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *Proc. ACM Design Automation Conference (DAC)*, 2006.

[56] ITU-T. 120: Message sequence chart (MSC). *ITU-T, Geneva*, 1996.

[57] J. Robertson and K. Gala. *Instruction and Data Cache Locking on the e300 Processor Core*. Freescale Semiconductor, Inc., 2006.

[58] A. Janapsatya, A. Ignjatovic, and S. Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proc. Conference on Asia South Pacific Design Automation (ASP-DAC)*, 2006.

[59] M. Kandemir. Data locality enhancement for CMPs. In *Proc. International Conference on Computer Aided Design (ICCAD)*, 2007.

[60] M. Kandemir and N. Dutt. Memory systems and compiler support for MPSoC architectures. In A. Jerraya and W. Wolf, editors, *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, 2005.

[61] M. Kandemir, I. Kadayif, and U. Sezer. Exploiting scratch-pad memory using presburger formulas. In *Proc. 14th International Symposium on Systems Synthesis (ISSS)*, 2001.

[62] M. Kandemir, O. Ozturk, and M. Karakoy. Dynamic on-chip memory management for chip multiprocessors. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2004.

[63] M. Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Proc. ACM Design Automation Conference (DAC)*, 2002.

[64] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. A compiler based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on CAD*, 23(2), 2004.

[65] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2004.

[66] D. B. Kirk. SMART (Strategic Memory Allocation for Real-Time) cache design. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 1989.

[67] S.-R. Kuang, C.-Y. Chen, and R.-Z. Liao. Partitioning and pipelined scheduling of embedded system using integer linear programming. In *Proc. International Conference on Parallel and Distributed Systems (ICPADS)*, 2005.

[68] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3), 1999.

[69] M. S. Lam and M. E. Wolf. A data locality optimizing algorithm. *SIGPLAN Notices*, 39(4):442–459, 2004.

[70] S. Lauzac, R. Melhem, and D. Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Proc. Euromicro Workshop on Real-Time Systems*, 1998.

[71] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1997.

[72] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[73] J. W. Lee and K. Asanovic. METERG: Measurement-based end-to-end performance estimation technique in QoS-capable multiprocessors. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2006.

[74] R. L. Lee, P. C. Yew, and D. H. Lawrie. Multiprocessor cache design considerations. In *Proc. International Symposium on Computer Architecture (ISCA)*, 1987.

[75] S. Lee, J. Lee, C. Y. Park, and S. L. Min. A flexible tradeoff between code size and WCET using a dual instruction set processor. In *Proc. International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2004.

[76] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007.

[77] X. Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *Proc. 40th ACM Design Automation Conference (DAC)*, pages 466–471, 2003.

[78] Y. Li and W. Wolf. A task-level hierarchical memory model for system synthesis of multiprocessors. In *Proc. ACM Design Automation Conference (DAC)*, 1997.

[79] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. ACM Design Automation Conference (DAC)*, 1995.

[80] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proc. 17th IEEE Real-Time Systems Symposium (RTSS)*, 1996.

[81] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[82] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 2000.

[83] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia. Utilization bounds for multiprocessor rate-monotonic scheduling. *Real-Time Systems*, 24(1), 2003.

[84] T. Lundqvist and P. Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3), 1999.

[85] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[86] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig. Fast, predictable and low energy memory references through architecture-aware compilation. In *Proc. Conference on Asia South Pacific Design Automation (ASP-DAC)*, 2004.

[87] T. Mitra and A. Roychoudhury. Worst case execution time and energy analysis. In Y. Srikant and P. Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation, 2nd Ed.*, chapter 1. CRC Press, 2007.

[88] A. M. Molnos, M. J. M. Heijligers, S. D. Cotofana, and J. T. J. van Eijndhoven. Cache partitioning options for compositional multimedia applications. In *Proc. 15th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, 2004.

[89] F. Mueller. Compiler support for software-based cache partitioning. In *Proc. ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 1995.

[90] F. Mueller. Generalizing timing predictions to set-associative caches. In *Proc. 9th Euromicro Workshop on Real-Time Systems*, pages 64–71, 1997.

[91] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3), 2000.

[92] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proc. International Symposium on Computer Architecture (ISCA)*, 1994.

[93] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proc. 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2003.

[94] F. Nemer, H. Cass, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. PapaBench: A free real-time benchmark. In *Proc. International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[95] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2005.

[96] R. Niemann and P. Marwedel. Hardware/software partitioning using integer programming. In *Proc. Conference on Design, Automation and Test in Europe (DATE)*, 1996.

[97] O. Ozturk, G. Chen, M. Kandemir, and M. Karakoy. An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multiprocessors. In *Proc. IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI)*, 2006.

[98] O. Ozturk, M. Kandemir, G. Chen, M. J. Irwin, and M. Karakoy. Customized on-chip memories for embedded chip multiprocessors. In *Proc. Conference on Asia South Pacific Design Automation (ASP-DAC)*, 2005.

[99] O. Ozturk, M. Kandemir, and I. Kolcu. Shared scratch-pad memory space management. In *Proc. 7th International Symposium on Quality Electronic Design (ISQED)*, 2006.

[100] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1999.

[101] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):682–704, 2000.

[102] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1), 1993.

[103] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 2002.

[104] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *Proc. 18th Euromicro Conference on Real-Time Systems (ECRTS)*, 2006.

[105] I. Puaut, A. Arnaud, and D. Decotigny. Performance analysis of static cache locking in multitasking hard real-time systems. Technical Report 0, IRISA, October 2003.

[106] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS)*, 2002.

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

[108] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technical University of Vienna, 1995.

[109] R. A. Ravindran, P. D. Nagarkar, G. S. Dasika, E. D. Marsman, R. M. Senger, S. A. Mahlke, and R. B. Brown. Compiler managed dynamic instruction placement in a low-power code cache. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2005.

[110] R. Reddy and P. Petrov. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2007.

[111] J. E. Sasinowski and J. K. Strosnider. A dynamic programming algorithm for cache memory partitioning for real-time systems. *IEEE Transactions on Computers*, 42(8), 1993.

[112] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 System Microarchitecture. Available on: `http://researchweb.watson.ibm.com/journal/rd/494/sinharoy.html`, 2005. Received March 2, 2005; accepted for publication June 27, 2005; Published online September 7, 2005.

[113] J. Sjodin and C. von Platen. Storage allocation for embedded processors. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

[114] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), 1993.

[115] A. Srinivasan, P. Holman, J. H. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Proc. 17th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2003.

[116] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest execution path search for programs with compelx flows and pipeline effects. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

[117] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *Proc. International Conference on Embedded Software (EMSOFT)*, 2004.

[118] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proc. 15th International Symposium on System Synthesis (ISSS)*, 2002.

[119] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proc. Design, Automation and Test in Europe Conference and Exposition (DATE)*, 2002.

[120] G. E. Suh, S. Devadas, and L. Rudolph. Dynamic cache partitioning for simultaneous multithreading systems. In *Proc. 13th IASTED International Conference on Parallel and Distributed Computing System (PDCS)*, 2001.

[121] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *Proc. 26th IEEE International Real-Time Systems Symposium (RTSS)*, 2005.

[122] F. Sun, N. K. Jha, S. Ravi, and A. Raghunathan. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *Proc. International Conference on VLSI Design (VLSI)*, 2005.

[123] Sun Microsystems, Inc. UltraSPARC T1 Overview. Available on: `http://www.sun.com/processors/UltraSPARC-T1/index.xml`, 2006.

[124] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The wavescalar architecture. *ACM Transactions on Computer Systems*, 25(2):4, 2007.

[125] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proc. ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1994.

[126] Texas Instruments, Inc. TMS470R1x System Module Reference Guide. Available on: `http://focus.ti.com/lit/ug/spnu189h/spnu189h.pdf`, 2004. Release November 2004.

[127] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3), 2000.

[128] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[129] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proc. International Conference on Hardware/Software Codesign (CODES)*, 2000.

[130] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2003.

[131] J. van Eijndhoven, J. Hoogerbrugge, M. N. Jayram, P. Stravers, and A. Terechko. *Cache-Coherent Heterogeneous Multiprocessing as Basis for Streaming Applications*, volume 3 of *Philips Research Book Series*, pages 61–80. Springer, 2005.

[132] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *Proc. International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2003.

[133] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[134] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *Proc. 3rd Workshop on Embedded Systems for Real-Time Multimedia (EstiMedia)*, 2005.

[135] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proc. Design, Automation and Test in Europe Conference and Exposition (DATE)*, 2004.

[136] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004.

[137] WCET benchmarks. Benchmarks from C-LAB and Uppsala University, 2004. `http://www.c-lab.de/home/en/download.html`.

[138] L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized usage of partitioned memories. In *Proc. 3rd Workshop on Memory Performance Issues (WMPI)*, 2004.

[139] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proc. Conference on Design, Automation and Test in Europe (DATE)*, 2005.

[140] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–87, 1967.

[141] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proc. 5th International Conference on Quality Software (QSIC)*, 2005.

[142] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. 3rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 1997.

[143] J. Xue and X. Vera. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Transactions on Computers*, 53(5):547–566, 2004.

[144] T.-Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(10), 1998.

[145] P. Yu and T. Mitra. Satisfying real-time constraints with custom instructions. In *Proc. ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2005.

[146] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, 1993.

[147] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. Improving WCET by optimizing worst-case paths. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.

[148] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET code positioning. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 2004.