Mälardalen University Dissertations
No. 7

Cache and Compiler Interaction

how to analyze, optimize and time cache behavior

Xavier Vera

2003

MÄLARDALEN UNIVERSITY
Department of Computer Science and Engineering
Mälardalen University

ISSN 1651–4238
ISBN 91–88834–27–1
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

A DISSERTATION
PRESENTED TO THE FACULTY
OF MÄLARDALENS HÖGSKOLA
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

With every waking breath I breathe
I see what life has dealt to me
With every sadness I deny
I feel a chance inside me die

Give me a taste of something new
To touch, to hold, to pull me through
Send me a guiding light that shines
Across this darkened life of mine

Breathe some soul in me
Breathe your gift of love to me
Breathe your life to lay 'fore me
Breathe to make me breathe

For every man who built a home
A paper promise for his own
He fights against an open flow
Of lies and failures we all know

To those who have and who have not
How can you live with what you've got
Give me a touch of something sure
I could be happy ever more

Breathe some soul in me
Breathe your gift of love to me
Breathe your life to lay 'fore me
Breathe to make me breathe
Breathe your honesty to me
Breathe your innocence to me
Breathe your word and set me free
Breathe to make me breathe

This life prepares the strangest things
The dreams we dream of, what life brings
The highest highs can turn around
To sow love's seeds on stony ground

Midge Ure – "breathe"

'cause time is what you make of it

Per tu
Pels teus somriures
Perquè un dia la pluja ens acompanyarà

Abstract

Caches have become increasingly important with the widening gap between main memory and processor speeds. Small and fast cache memories are designed to bridge this discrepancy. However, they are only effective when programs exhibit sufficient data locality. In addition, caches are a source of unpredictability, resulting in programs sometimes behaving in a different way than expected.

Detailed information about the number of cache misses and their causes allows us to predict cache behavior and to detect bottlenecks. Small modifications in the source code may change memory patterns, thereby altering the cache behavior. Code transformations which take the cache behavior into account might result in a high cache performance improvement. However, cache memory behavior is very hard to predict, thus making the task of optimizing and timing cache behavior very difficult.

This dissertation proposes and evaluates a new compiler framework that analyzes and tunes cache behavior. The proposed framework is based on a new characterization of data reuse across multiple loop nests, which allows analyzing the cache behavior of whole programs with regular computations. The framework uses an accurate cost model that describes misses across different cache levels and considers the effects of other hardware components such as branch predictors, which drives the application of tiling and padding transformations. In order to select the best parameter values, we combine the cost model with a genetic algorithm to compute tile and pad factors that enhance the program performance.

Finally, our method explores the use of *cache partitioning* and *dynamic cache locking* to provide worst-case performance estimates in a safe and tight way for multitasking systems. We use cache partitioning, which divides the cache among tasks to eliminate inter-task cache interferences. We combine static cache analysis and cache locking mechanisms to ensure that all intra-task conflicts, and consequently, memory access times, are exactly predictable.

The results of our experiments demonstrate the capability of our framework to describe cache behavior at compile time. Extensive validation shows that our

accurate cost model is appropriate to achieve significant speedups compared to state-of-the-art techniques. This dissertation also compares our timing approach with a system with a non-partitioned but statically locked data cache. Our method outperforms static cache locking for all analyzed task sets under various cache architectures, demonstrating that our fully predictable scheme does not compromise the performance of the transformed programs.

Populärvetenskaplig

Vem har inte drömt om att äga en Ferrari? Detta italienska märke har alltid varit känt för glamour och prestanda. Dock är tillförlitligheten ett problem. Tänk dig, för ett ögonblick, att du äger en luxuös F40. Hur lång tid skulle det ta att köra tvärs över USA? Mitt svar är: "det beror på hur fort du kan få fram reservdelar varje gång bilen går sönder". Om vi har en långsam lastbil som transporterar reservdelarna, så är det i själva verket lastbilens hastighet som bestämmer tiden. Ett sätt att få fram delarna snabbare är att ha en snabbare bil (säg, en BMW) som transporterar delar från lastbilen. Problemet är att den har ett litet bagageutrymme. Denna lösning är alltså effektiv bara om vi kan förutsäga vilka problem vår Ferrari kommer att få, så vi kan lasta rätt reservdelar från början. Om den snabba bilen kommer fram till Ferrarin utan rätt reservdelar, så måste båda vänta på den långsamma lastbilen i alla fall.

Problemet är att denna information är svår att få. Vädret, resrutten, trafiken,... Alla dessa faktorer kan påverka bilens tillförlitlighet och alltså vilka reservdelar vi bör lägga i bagageutrymmet för att slippa vänta mer än nödvändigt på lastbilen. Men kan man förutsäga dessa faktorer kan man dels förutsäga bättre hur lång tid resan tar, och dels förkorta restiden genom att lasta rätt reservdelar.

Denna avhandling handlar dock inte om bilar, utan om datorer och hur man kan få dem att gå med förutsägbar, hög hastighet. Ferrarin motsvarar den snabba processor (t.ex. en Pentium-4) som sitter i en dator idag. Den långsamma lastbilen svarar mot datorns primärminne: ett problem i dagens datorer är att minnena är mycket långsammare än processorn, gör man ingenting kommer alltså processorns hastighet att begränsas av hur snabbt man får fram data från minnet. BMW:n svarar mot datorns cacheminne: ett litet, snabbt minne där man sparar de data som man tror att processorn kommer att använda härnäst. Kan processorn ta sina data därifrån behöver den inte vänta på de långsamma vanliga minnet. Vad avhandlingen handlar om är metoder för att analysera hur cacheminnet används i en dator, för att därigenom kunna generera programkod som utnyttjar det bättre

och därför får bättre prestanda. Resultaten kan tillämpas både i högpresterande och i säkerhetskritiska datorsystem.

Acknowledgments

I'll try to be quick. Sincere thanks once again to:

Björn Lisper, my main supervisor, who gets to see my eccentricities up close;

My parents, for buying that first computer when I was a kid; and very especially for encouraging me to continue my studies in Computer Science;

My wonderful brother and sister, for making me feel at home even though I was on the other side of the world;

My good friends, Toni Palau, Xavi Magrí, MAT, Manel Fernández, notably Jaume Abella (whom I've had the pleasure to work with during all these years), and to Nerina Bermudo for accompanying me on *our* extraordinary trip;

A special mention to Antonio González, Jan Gustafsson, Josep Llosa and Jingling Xue, who generously gave of their time and assisted me with part of the research of this thesis;

And finally, once again, thanks to everyone at IDt/MDH. Thanks to my roommate, Jan Carlson, "let's grumble" mate Markus Bohlin, and badminton (among others)-dude Baran. And of course, as always, to Harriet for the countless hours spent arranging all my trips and administrative problems (and they're not few).

To anyone who knows a PhD student, never underestimate the power of your encouragement.

All right! Now on with the show...

Sydney, August 2003
Barcelona, September 2003
Västerås, November 2003

Contents

List of Figures

List of Tables

Publications

Papers Included in the Thesis

The following is a list of all publications subject to peer review that are part of this thesis.

Cache Analysis

- "Let's Study Whole-Program Cache Behaviour Analytically", Xavier Vera, Jingling Xue. *Proceedings of 8th International Symposium on High-Performance Computer Architecture (HPCA-8).*

- "Efficient and Accurate Analytical Modeling of Whole-Program Data Cache Behavior", Jingling Xue, Xavier Vera. *To Appear in IEEE Transactions on Computers.*

- "Efficient Compile-Time Analysis of Cache Behaviour for Programs with IF Statements", Xavier Vera, Jingling Xue. *Proceedings of International Conference on Algorithms and Architectures for Parallel Processing (ICAAPP02).*

Cache Optimization

- "Optimizing Program Locality Through CMEs and GAs", Xavier Vera, Jaume Abella, Antonio González, Josep Llosa. *Proceedings of 12th International Conference on Parallel Architectures and Compilation Techniques (PACT03).*

- "Near-Optimal Padding for Removing Conflict Misses", Xavier Vera, Antonio González, Josep Llosa. *Proceedings of 15th International Workshop on Languages and Compilers for Parallel Computing (LCPC02).*

- "Near-Optimal Tiling by means of Cache Miss Equations and Genetic Algorithms", Jaume Abella, Antonio González, Josep Llosa, Xavier Vera. *Proceedings of 31st International Conference on Parallel Processing Workshops (ICPP02).*

Cache Timing

- "Data Caches in Multitasking Real-Time Systems", Xavier Vera, Björn Lisper, Jingling Xue. *Proceedings of 24th International Real-Time Systems Symposium (RTSS03).*

- "Data Cache Locking for Higher Program Predictability", Xavier Vera, Björn Lisper, Jingling Xue. *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS03).*

Published Papers not Included in the Thesis

The following is a list of publications subject to peer review that are *not* part of this thesis and which are related to this dissertation.

- "A Fast and Accurate Approach to Analyze Cache Memory Behavior", Xavier Vera, Josep Llosa, Antonio González, Nerina Bermudo. *Proceedings of 6th International Euro-Par Conference (EuroPar 2000).*

- "An Efficient Solver for Cache Miss Equations", Nerina Bermudo, Xavier Vera, Antonio González, Josep Llosa. *Proceedings of 1st IEEE International Symposium on Performance Analysis of Software and Systems (ISPASS00).*

- "A Fast and Accurate Framework to Analyze and Optimize Cache Memory Behavior", Xavier Vera, Nerina Bermudo, Josep Llosa, Antonio González. *To Appear in ACM Transactions on Programming Languages and Systems.*

- "Code Tiling for Improving the Cache Performance of PDE Solvers", Qinguang Huang, Jingling Xue, Xavier Vera. *Proceedings of 32nd International Conference on Parallel Processing (ICPP03).*

1

Introduction

CHAPTER 1

INTRODUCTION

With ever-increasing clock rates and the use of new architectural features, the speed of processors increases dramatically every year. However, memory speeds have lagged behind, thus increasing memory latency [68]. This wide performance gap affects all computer systems, and is a key obstacle to achieve high processor utilization due to memory stalls. The basic solution that almost all systems rely on is the use of cache memories.

Memory is organized hierarchically in such a way that the lower levels are smaller and faster. In order to fully exploit the memory hierarchy, one has to ensure that most of the memory references are handled by lowest levels of cache. Programmers spend a significant amount of time improving locality, which is tedious and error prone. Compilers apply useful loop transformations and data layout transformations to take better advantage of the memory hierarchy. In all cases, a fast and accurate assessment of a program's cache behavior is needed at compile time to make an appropriate choice of transformation parameters.

Unfortunately, cache memory behavior is hard to predict. That makes it very difficult to statically analyze the interaction between a program's referencing pattern and the memory subsystem. This thesis describes a static approach to characterizing whole programs' cache memory behavior. This framework is used to implement efficient compiler transformations and to time memory performance. The rest of this chapter provides a detailed motivation for this work and introduces the goals that have been achieved. Finally, we list the contributions of this work and present a road map for the remainder of the thesis.

1.1 Motivation

Data caches are widely used to bridge the increasing gap between processor and main memory speeds. The effectiveness of a cache memory depends not only on the hardware structure, but also on the code generated by the compiler.

Various hardware approaches have been proposed for increasing the effectiveness of the memory hierarchy. Some of them try to hide the latency, such as multithreading [94] or dynamic instruction scheduling. Some other hardware techniques have been proposed to reduce conflict misses, such as the victim cache [79] or pseudo-random placement functions [74, 95, 146, 150, 151].

Prefetching data ahead of use has the potential to tolerate the growing processor-memory performance gap by overlapping long latency memory accesses with useful computation. It can be done either by hardware [10, 30] or software means [21, 120]. Sophisticated software prefetching techniques have been automated for scientific codes that access dense arrays in loop nests [120]. A similar level of success has been shown for general-purpose programs by means of a dynamic prefetching scheme [31, 32]. All these methods have in common that they increase the amount of memory traffic by fetching more data than needed, which may cause contention in the memory system.

On the other end, software techniques try to improve locality of programs, thereby reducing the number of cache misses. Thus, they do not increase the memory bandwidth requirement. Compilers increase locality either by changing data access pattern or data layout. They apply useful loop transformations such as tiling [24, 35, 93, 168], loop interchange [50, 113, 168, 170], loop fission/fusion [113], and data layout transformations [28, 80, 129, 131, 142].

When using caches in hard real-time systems there is an unacceptable possibility that a high cache miss penalty combined with a high miss ratio might cause a missed deadline, jeopardizing the safety of the controlled system. Besides, caches also increase the variation in execution time, causing jitter. Thus, many safety-critical systems either do not have caches or disable them. Nevertheless, a system with disabled caches will waste a lot of resources; the CPU will be underutilized, and also the power consumption will be larger since memory accesses that fall into the cache consume less power than accesses to main memory. Thus, bounding memory performance tightly in hard real-time systems with caches is important to use the system resources well.

1.1.1 Compiler Cache Optimizations

In order to tune the code so that it takes plenty advantage of cache memories, an accurate assessment of the number of misses and their causes is required. Simulators are used for describing memory behavior accurately. However, they are very slow and do not provide too much insight into the causes of the misses. Thus, current approaches are based on simple models (heuristics) for estimating locality [25, 35, 93,

129, 131]. However, modern architectures have very complex internal organization, with different levels of cache, branch predictors, etc. Such models provide very rough performance estimates, and in practice, are too simplistic to statically select the best optimizations.

In this thesis we tackle the problem of improving cache behavior by means of tiling and padding. Tiling has been shown to be useful for many algorithms in linear algebra. By restructuring the loop and changing the order in which memory references are executed, it reuses data in the faster levels of the hierarchy; thus it reduces the average latency. Nevertheless, finding the optimal tile sizes is a very complex task. The solution space is very large and non-uniform, and exploring all possible solutions is infeasible. Furthermore, there is no simple model that can be used to find a closed-form solution and the non-uniformity makes heuristics development very hard. Padding has a significant potential to remove conflict misses. In fact, it can remove most conflict misses by changing the addresses of conflicting data, and some compulsory misses by aligning data with cache lines. However, choosing the optimal data layout is an NP-complete problem [123].

1.1.2 Timing Cache Behavior

Hard real-time systems are those where a failure to meet a deadline can be fatal. To guarantee their behavior, the worst case behavior has to be analyzed. It will be used to ensure timely responses from tasks and as input to scheduling algorithms.

That implies that it is necessary to know the execution time for the tasks in a real-time system. The analysis from a high-level point of view is concerned with the possible paths through the program. Low-level analysis determines the effect on the program timing of machine-dependent factors, such like caches. While control flow can be modeled precisely, hardware (caches, branch predictors, etc.) can give rise to actual unpredictability. Therefore, real-time systems have to be analyzed as a whole, where software and hardware play their roles.

In order to get an accurate worst-case execution time (WCET), a tight worst-case memory performance (`WCMP`) is needed. Current WCET platforms are applied to rather simple architectures (they usually do not consider data caches) and make simplifying assumptions such that the tasks are not preempted. In order to consider the costs of task preemption, some studies incorporate into the schedulability analysis the costs of manipulating queues, processing interrupts and performing task switching [15, 16, 76, 82].

However, cache behavior is very hard to predict, which leads to an overestimation of the `WCMP`, and thus for the WCET as well. For this reason, many safety-critical

systems do not use caches since it is very hard to prove that the system is reliable under all circumstances.

In this thesis, our goal is to guarantee an exact prediction of hits or misses for all memory accesses. In modern processors, it may happen that a cache hit is more expensive than a cache miss [105]. If a memory access is not classified definitely as a hit or miss, a subsequent calculation pass in a WCET analysis would have to consider both situations to detect the worst-case path.

1.2 Problem Statement

There can be a wide difference in performance between programs that are designed to exploit the cache memory and those that are not. Data cache behavior is very difficult to analyze both efficiently and accurately. Moreover, due to the non-linear mapping between main memory and cache sets, data cache behavior may fluctuate due to variations in problem sizes and base addresses [93].

Recent studies [114] highlight the fact that most misses are inter-nest, whereas most reuse is intra-nest. This is especially significant for numerical codes. Thus, a model that gives a quantitative and qualitative measurement of cache misses for the whole program is required.

Previous works for analyzing cache memory behavior are based either on simulation or simple heuristics. Trace-driven simulation [148] and performance counters [6] can obtain accurately a broad range of information for a program. However, simulation is very time-consuming and may need a lot of space to store traces. The latter is restricted to measurements of existing caches. In addition, neither approach offers insight into the causes of the misses. On the other hand, analytical methods [29, 48, 52] are limited mainly to individual perfect nests.

We introduce an accurate model for statically predicting the cache behavior of whole programs. In particular, we use our model to address the following problems:

- How to characterize precisely the cache behavior for whole programs with regular computations?
- How to consider complex architectures, with multi-level caches and branch predictors?
- How to select the best transformation parameters to enhance performance applying different optimization techniques in concert?
- How to use cache behavior to time memory performance in a safe way for systems that allow preemption?

We now discuss these problems by comparing our solutions to some existing work.

1.2.1 Static Analysis of Whole Programs

There has been extensive research on describing cache memory behavior statically [29, 48, 52, 141, 168]. Analytical methods use mathematical formulas to provide a characterization of a program's cache behavior so that it is not only possible to obtain the number of cache misses but also to reason about the causes of such misses from these formulas. The ultimate goal is to develop an analytical method that can provide accurate assessments of when and why cache misses occur using a reasonable amount of computational resources (e.g., CPU time, memory and disk usage).

Porterfield [124] introduces the concept of overflow iteration for predicting the miss ratio for a fully set-associative LRU cache. Ferrante *et al.* [47] provide closed-form formulas to estimate the capacity misses of a loop nest. Temam *et al.* [141] also consider conflict misses but for a subset of array references. Wolf and Lam [168] propose to use vectors to describe data reuse for uniformly generated references in a perfect loop nest. They also use reuse vectors to derive an estimate of cache misses to guide their data locality algorithm. Gannon *et al* [50] and Wolfe [171] discuss the use of reference windows for predicting cache misses.

Ghosh *et al* [52] present a framework which is targeted at isolated perfect loop nests. They make use of the reuse vectors among uniformly generated references in the same nest. Fraguela *et al* [48] rely on a probabilistic analytical method to provide a fast estimation of cache misses. While allowing multiple nests, they exploit only the reuse between references contained in the same nest. Chatterjee *et al* [29] present an ambitious method for *exactly* modeling the cache behavior of loop nests. They can formulate Presburger formulas for a looping structure consisting of imperfect nests, IF statements, references with affine accesses and non-linear data layouts. However, due to the complexity of their method only small kernels can be analyzed.

The primary drawback of all these methods is that they only apply to small parts of the codes, and hence we need a tool that characterizes cache behavior for whole programs. On the other hand, they usually fail to describe accurately conflict misses, which shows the need for a more precise description of cache behavior.

1.2.2 Considering Complex Architectures

All modern compiler techniques that are used to improve cache behavior are based on simple models that usually only consider one level of cache [25, 35, 93, 129, 131].

Moreover, current static analyses [29, 48, 52, 141, 168] only estimate locality for the first level of cache. However, modern architectures have very complex internal organization. The increasing performance mismatch between main memory and processor speeds requires more levels of cache. Thus, we need a model that characterizes memory behavior not only for the first level of cache, but for any hierarchy.

On the other hand, obtaining cache miss ratios is not sufficient for estimating the cache performance on current machines since they include several techniques such as non-blocking caches, out-of-order execution and speculation for tolerating the latency of cache misses. Hence, a performance model that goes beyond cache misses and opens the possibility to include more architectural features is needed.

1.2.3 Applying Compiler Techniques in Concert

Caches improve the speed of programs by reducing the number of accesses to the slow upper levels of the memory hierarchy. Most of current compilers apply a set of transformations in sequence. However, selecting an optimal sequence of transformations is very difficult, and few results have been published [53, 113, 166]. These methods are based on heuristics and iterative methods, which do not solve the problem of optimality. Hence, we need an approach that tries to apply all optimizations at the same time.

1.2.4 Timing Memory Performance

The computation of WCET in the presence of instruction caches has progressed in such a way that it is now possible to obtain an accurate estimate of the WCET for non-preemptive systems [5, 7, 64]. These results can be generalized to preemptive systems [12, 17, 18, 22, 88, 97, 121, 125]. However, there has not been much progress with the presence of data caches. Instructions such as loads and stores may access multiple memory locations (such as those that implement array or pointer accesses), which makes the attempt to classify memory accesses as hits or misses very hard.

Current approaches [5, 45, 87, 102] provide an estimation of WCET by considering data caching where only memory references which are scalar variables are considered. Thus, they fail to study real codes with dynamic references (i.e., arrays and pointers). White *et al* [162] propose a method for direct-mapped caches based on static simulation. They categorize static memory accesses into (i) first miss, (ii) first hit, (iii) always miss and (iv) always hit. Array accesses whose addresses can be computed at compile-time are analyzed, but they fail to describe conflicts which are always classified as misses.

Hence, there is a need for a tool that computes the WCET in the presence of data caches for real programs.

1.3 Contributions

An accurate information of the cache memory behavior is essential to both optimize and time memory behavior. However, this is a very difficult task. This thesis contributes an analytical method which describes the cache misses of whole programs with regular computations. This analysis is the first step required by any compiler transformation or WCET analysis.

Our cache analysis is used to implement compiler cache optimizations in concert. We show how tiling and padding can be implemented at the same time, which avoids the problem of selecting the optimal sequence of transformations. This thesis also presents how the worst-case memory performance can be computed in the presence of data caches at compile time. The static analysis is used for those regions that are predictable. We combine it with cache partitioning and cache locking to avoid unpredictability caused by inter- and intra-task misses that are data-dependent.

The main contributions of this thesis are:

Analysis Framework We present an analytical method which builds on the top of the Cache Miss Equations [52].

We first generalize the concept of reuse vectors [168] to quantify reuse among references contained in multiple nests [160]. This simple characterization of reuse allows setting up a set of equalities and inequalities that describe the cache behavior of whole programs with regular computations, which may consist of subroutines, call statements, if statements and arbitrarily nested loops. We solve the equations by means of sampling techniques [155, 159], where the level of accuracy is decided by the user.

This provides a framework that can be integrated in any static tool, like compiler optimizations or WCET analysis tools. To the best of our knowledge, this is the first analytical method with demonstrated capability for analyzing whole programs, efficiently and with a high degree of accuracy.

Memory Hierarchy Behavior Our model applies to a wide range of memory hierarchies. We show how to describe the behavior of multi-level caches, partitioned caches and caches with locking features.

Data Locality Optimizations We detail an approach that can be used to drive program transformations oriented to enhance data locality. The centerpiece

of the proposed method is an accurate cost model combined with a genetic algorithm [154].

In particular, we improve the order of memory accesses via tiling [1], whereas conflict misses that tiling cannot eliminate are removed via padding [158]. Our approach selects appropiate tile and pad factors at the same time. The cost model considers different levels of the memory hierarchy and the performance cost of the miss-predicted branches. To the best of our knowledge, our approach is the first one that considers applying more than one transformation at the same time. In addition, it tries to optimize the overall performance rather than only consider the first level of cache.

Time Cache Behavior This thesis presents a compiler algorithm to estimate the worst-case memory performance (`WCMP`) for multitasking systems in the presence of data caches [157]. We use cache partitioning to eliminate inter-task conflicts, thus we can analyze each task in isolation. Tasks now use a smaller cache. Hence, we apply compiler cache optimizations such as tiling and padding to reduce the number of misses.

We have developed a compile-time algorithm that identifies those regions of code where we cannot exactly determine the memory accesses. In those situations, the cache is locked so we do not jeopardize the cache analysis [156]. We use a locality analysis based on Wolf and Lam's reuse vectors [168] to select the data to be loaded. Since the state of the cache is known when leaving the region, we can apply our static analyzer [160] for the next regions of code, thus having both predictability and good performance. To the best of our knowledge, this is the first framework that obtains an exact `WCMP` for multitasking systems.

Implementation Our system consists of components on normalizing loop nests, inlining calls (abstractly), generating reuse vectors, sampling memory accesses and forming and solving the equations for obtaining cache misses for multi-level and locking caches. It also contains a module that inserts lock/unlock/load instructions automatically in order to have predictable codes without compromising performance. We have also implemented a genetic algorithm in order to explore the solution space for optimizing cache behavior.

Validation and Experimental Results We have validated our analysis method against cache simulation using programs from SPECfp95, Perfect Suite, Livermore kernels, Linpack and Lapack.

In order to show our ability to improve programs' performance, we report results for a set of modern processors that represent current architectural paradigms. We have chosen the Pentium-4 [23] (CISC), Alpha-21264 [86] and UltraSparcIII [140] (RISC), and Itanium [135] (EPIC).

Finally, in order to show to what extent our method can estimate the `WCMP`, we present results for a collection of task sets consisting of programs drawn from several related papers [5, 87, 162]. This collection includes kernels operating on both arrays and scalars, such as SQRT or FIBONACCI. We have also used FFT to show the feasibility of our approach for typical DSP codes. For the sake of concreteness, we have chosen the cache architectures of a set of modern processors widely used in the real-time area: microSPARC-IIep [139], PowerPC 604e [119], IDT 79RC64574 [75] and MIPS R4000 [117]. We also include results for the recently released NEC V850E/ME2 32-bit high-performance microcontroller.

1.4 Organization

Chapter 2 gives an introduction about cache memories. We start by presenting the basic cache architectures. Next, some terminology for program analysis used in this work is presented: we introduce the reuse vectors as a metric for reuse analysis and the original CMEs framework.

Chapter 3 presents the underlying model on which this work is based. We present the programs and architectures we deal with. Then, we explain the different transformations we apply to the codes in such a way that we get a new version suitable for our analysis. Some important statistical notions that are used to model the number of misses are introduced.

We give an overview of our tool and introduce our experimental framework in Chapter 4. We detail the benchmarks that we chose and the real architectures used for our measurements.

We distribute the main contributions of this thesis in the next three chapters. Chapter 5 introduces our new approach to analyzing whole program cache behavior. We start by explaining how we deal with call statements and subroutines. Next, we discuss a reuse metric for whole programs, and provide a mathematical formulation for expressing cache misses across multiple loops and `if` statements. We show how to extend these equations to deal with locking caches and multi-level memory hierarchies. Finally, we present our extensive validation. We have evaluated both the accuracy and feasibility of our analysis.

Chapter 6 presents our model to develop automatic compiler cache optimizations. We start by presenting our cost model function. Then, we introduce our genetic algorithm, and give complete details of its implementation. Finally, we present results of our optimization scheme comparing it with state-of-the-art techniques.

Chapter 7 discusses our approach to timing cache memory behavior for multitasking real-time systems. First, we give an overview of a general WCET tool. Then, we present our solution to have predictable programs on a unitask environment. Later, we extend our approach to deal with multitasking systems. We present extensive evaluations of our method. We evaluate the accuracy and contributions of all different components independently. Finally, we give performance measurements for a multitasking system.

We discuss some related work in Chapter 8. Chapter 9 contains a summary of the main contributions of this dissertation. It also points out some future work that may be addressed using this thesis as a starting point.

Appendix A contains some codes from the benchmarks we have used to evaluate our analysis technique. Appendix B has all the codes used to evaluate our optimization framework.

2

Data Cache Analysis: Background and Overview

CHAPTER 2

DATA CACHE ANALYSIS: BACKGROUND AND OVERVIEW

Processing capabilities of microprocessors used in current computer systems continually improve their processing capabilities due to increasing clock frequencies and the exploitation of parallelism at different levels. However, such processor performance is meaningless without a fast and broad memory subsystem.

Unfortunately, memory subsystem has become the bottleneck of current high-performance computers, limiting how fast data is brought (received) to (from) the processor. The gap between main memory and processor performance is increasing every year. Whereas processor performance has improved about 60% per year for the last 20 years, memory access time has only decreased at less than 10% per year [68]. This large latency, which is increasing at a rate of 45% per year, is a primary obstacle to improve general system's performance.

Memory hierarchy and cache memories were introduced as a hardware solution to hide this gap, becoming more and more significant with the widening performance difference. This chapter introduces the memory hierarchy and the original Cache Miss Equations (CMEs) [52]. The goal is to provide an overview of the CMEs framework and cache memories to help the reader fully understand the work presented in this thesis.

We first introduce the memory hierarchy and describe different kinds of cache memories. Then, we explain the concepts of *reuse* and *locality*, which are essential for analyzing cache memory behavior. Finally, we introduce the original CMEs framework.

2.1 Memory Hierarchy

We can see a wide variety of storage in a computer system, which can be organized in a hierarchy (see Figure 2.1) according to either their speed or their cost.

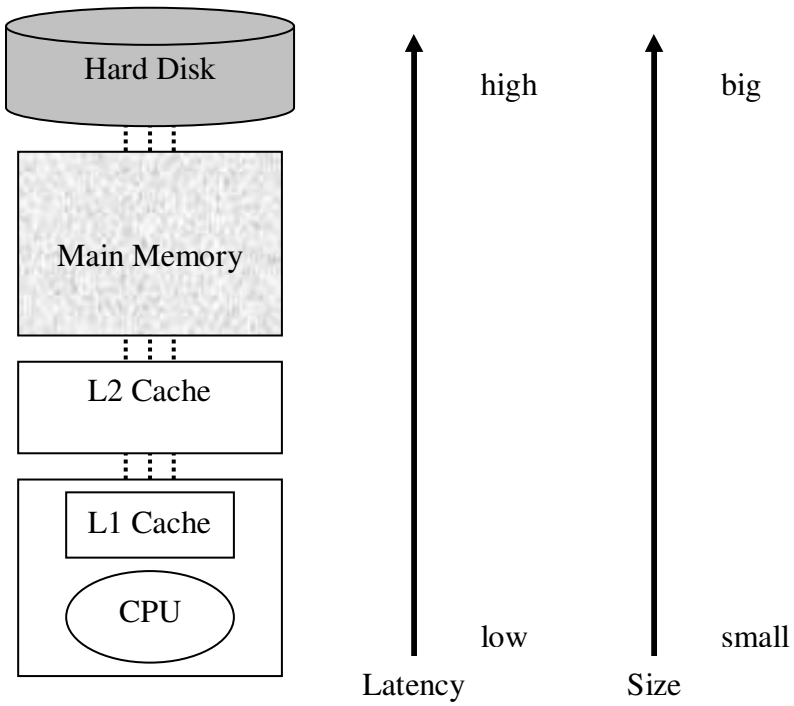


Fig. 2.1: Memory hierarchy.

The closer the memory is to the processor, the faster it is. The small high-speed memory is generally referred to as cache memory, or *cache* for short. Since caches are smaller than the main memory, they can be designed using faster and more expensive techniques.

The main property that makes the memory hierarchy work well in the general case is called *locality*:

Locality There exists temporal locality (i.e., it is very likely that instructions access data that has been already fetched), and spatial locality (i.e., it is very likely that instructions access data nearby data that have been already fetched).

The memory hierarchy aims at having data that it is supposed to be accessed very often in memory levels very close to the processor. Since these memories are faster, the latency is smaller. Thus, the general performance is increased. The system first copies the data needed by the processor from memory into the cache (see Figure 2.2), and then from the cache into a register in the CPU. Storage of

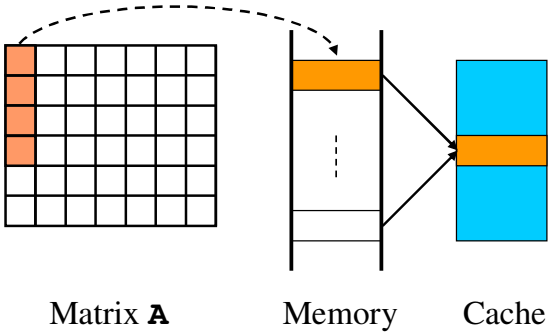


Fig. 2.2: How data is stored in both main memory and cache.

results is in the opposite direction. First the system copies the data into the cache. Depending on the cache architecture details, the data is then immediately copied back to memory (write-through), or deferred (write-back). If an application needs the same data again, data access time is reduced significantly if the data is still in the cache.

2.1.1 Cache Memories

Cache memories try to capture the most frequently accessed data items. Placed between processor and main memory, they bridge queries from the processor. When a memory request is generated, the request is first presented to the lowest level of cache, and only in the case it cannot respond the request is presented to the upper levels of the memory hierarchy. This can be summarized as follows:

1. If the cache has the data that the processor needs (*cache hit*), it brings the data to the processor.

2. Otherwise, the processor suffers a *cache miss* and the next level in the memory hierarchy must be accessed.

Cache misses are classified into three groups: compulsory, capacity, and conflict. Compulsory misses, also referred to as cold misses, are mostly independent of the cache size, and occur because data must be loaded at least once into the cache before it can be used. Capacity misses correspond to the misses that happen because the cache is not large enough to hold the working set of the program. Conflict misses are generally the result of how the cache is being managed, and can occur even though

the cache is sized sufficiently to hold the program's data; they occur when multiple data are competing for the same location in cache.[1]

Let us look at the various parameters that affect the design of a cache memory. We explain in the following sections how data is organized in the cache and different policies that keep the coherence between main memory and caches [68].

2.1.2 Cache Organization

Caches are characterized by the following parameters. *Cache size* (C) defines the total number of bytes it has. The *line size* (L) determines how many contiguous bytes are fetched from memory.

Definition 2.1 **(Memory Line)** A memory line refers to a cache-line-sized block in main memory.

Definition 2.2 **(Cache Line)** A cache line refers to the actual cache block in which a memory line is mapped.

Definition 2.3 **(Cache Set)** is a collection of cache lines.

Definition 2.4 **(Associativity (k))** is the number of cache lines in a cache set.

Sometimes, a cache configuration is identified as a triple (C,L,k).

Depending on the location in the cache that a block from memory can reside in, we have different cache organizations.

1. **Direct mapped**. A given block can only reside in one exact location: the memory address of the incoming cache line controls which cache location is going to be used. Thus, the candidate location is controlled by the memory address and not the usage. The cache size is

$$\mathtt{C} = \mathtt{L} \times Number\_Lines$$

   Implementing this organization is straightforward and it is relatively easy to make it scale with the processor clock. Even though this design works well in many cases, this policy has the potential downside of replacing a cache line that contains information needed shortly afterwards.

[1] A conflict miss is a memory reference that hits in a fully-associative cache but misses on a set-associative cache (see Section 2.1.2).

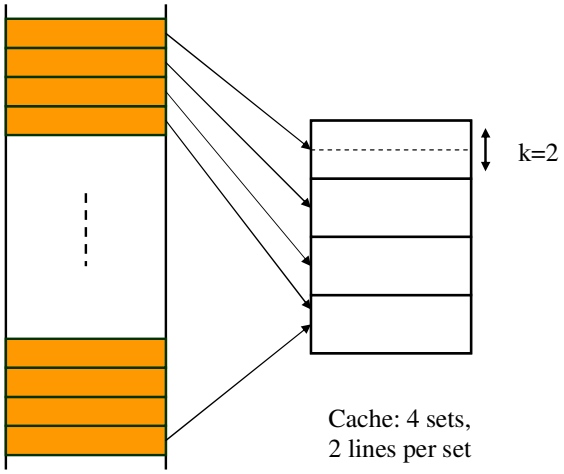


Fig. 2.3: Mapping of such a 2-way set-associative cache.

2. **Fully associative**. The fully-associative cache design solves the potential problem of thrashing with a direct-mapped cache. The replacement policy (see Section 2.1.3) is no longer a function of the memory address, but considers usage instead. A block from memory can reside in any location in the cache.

   However, this comes at a price, which is cost. Additional logic is required to track usage of lines. The larger the cache, the higher the cost. Therefore, it is difficult to scale this organization to large caches.

3. **`k`-way set-associative**. A set-associative cache design uses several small fully-associative caches, where each cache is referred to as a set. Each set has exactly `k` cache lines. On an incoming request, the cache controller decides which set the line will go into. Within the set, a memory block may reside in exactly any of the `k` locations. Figure 2.3 shows a 2-way set-associative cache, with 4 sets.

Two factors are critical at the moment of choosing an index scheme that maps memory lines onto cache lines (sets); firstly, the chosen function should have a cheap and easy implementation in hardware, and secondly it is important that it has a good behavior on any kind of regular address patterns. Modulo function (modulo the number of cache sets) is the most common index function used. However, alternative schemes may help programs to take advantage of cache memories. Prime-modulus

functions [95] and skewing functions [74] have been tested successfully. The use of XOR functions was proposed by Frailong *et al.* [49], and some pseudo-random placements by Rau *et al.* [128].

2.1.3 Replacement Policies

For those operations that result in a cache miss, the datum is retrieved from the main memory and copied into the cache, resulting in some other datum being removed from the cache to make room for this new datum. The cache replacement policy (i.e., which item to remove from the cache) is crucial for performance.

The most common algorithms are:

LRU The least-recently used line is removed from cache.

Random One line is chosen randomly.

FIFO The line that was brought in the first time is replaced (even though it may happen that it is accessed very often).

Nearly all caches in commercial products have LRU replacement policy to manage the different lines we have in a set.

2.1.4 Writing to the Cache

Special actions should be taken in case of WRITE operations. 10–30% of the memory accesses are WRITEs. Handling them is somewhat tricky because of the interaction of the cache with input/output systems. Keeping cache coherence with the main memory is very important. Different solutions have been presented, depending on whether there exists a cache miss or a cache hit.

When having a cache hit:

Write through We write the modified item both in the cache and in the memory. Since main memory is rather slow, hardware solutions like buffering are implemented in order to speed up the process of writing to main memory.

Copy back Data is written in the main memory only when the cache line is replaced. Thus, the memory does not contain updated information.

In case of suffering a cache miss:

Write allocate The memory line is modified and it is brought to the cache afterwards.

No write allocate The memory line is modified, but the data is not brought to the cache.

When specifying a writing policy, both hit and miss policies should be provided. The two most common configurations are:

- Copy back with Write allocate.
- Write through with No write allocate.

2.1.5 Locking Caches

Cache locking allows some or all of the contents of the cache to be locked in place. Disabling the normal replacement mechanism, provided that the cache contents are known, makes the time required for a memory access predictable. This ability to lock cache contents is available on several commercial processors (PowerPC 604e [119], 405 and 440 families [73], Intel-960, some Intel x86, Motorola MPC7400 and others). Each processor implements cache locking in several ways, allowing in all cases *static locking* (the cache is loaded and locked at system start) and *dynamic locking* (the state of the cache is allowed to change during the system execution).

2.2 Locality Analysis

Cache memories usually present very low associativity, which may result in data being replaced before it is reused. Furthermore, there exists programs without sufficient locality in their accesses, which makes them spend a lot of time transferring data between main memory and cache.

Understanding data reuse is essential to predict cache behavior, since a datum will only be in the cache if its line was referenced some time in the past. *Reuse* happens whenever the same data item is referenced multiple times. This reuse results in *locality* if it is actually realized; reuse will result in a cache hit if no intervening reference flushes out the datum.

As an example, consider a `k`-way set-associative cache with an LRU replacement policy. A memory access m_a to memory line l will result in a cache miss if l is accessed for the first time. Otherwise, let m_b be the most recent previous access also to l. Then, m_a will result in a miss if there are `k` distinct memory lines that are accessed between m_b and m_a that are also mapped to the same cache set as l, and a hit otherwise.

Definition 2.5 **(Memory Reference)** A memory reference is a static read or write.

```
double a[N][N], b[N][N], c[N][N];
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    for(k=0;k<N;k++) {
      a[i][j] = a[i][j] + b[i][k] * c[k][j];
    }
  }
}
```

Fig. 2.4: A running example: matrix multiplication algorithm.

We introduce a running example that we will use through this thesis. We consider the matrix multiplication algorithm without initialization as shown in Figure 2.4, where we highlight the four memory references.

In order to model cache behavior, we need to know whether two consecutive accesses fetch data from the same memory line, and all intervening accesses between such a pair of consecutive accesses.

Formally stated, we split a data cache analysis into the following steps:

1. *Reuse Analysis* describes the intrinsic data reuse among all different memory references.

2. *Data Locality Analysis* describes the subset of reuses that actually results in locality.

Now, we describe each step in detail for codes consisting of a single perfectly nested loop. We first introduce some terminology that is used through the thesis. Next, we present the *reuse vectors* as a metric to describe reuse. Finally, we give an overview of the CMEs, which are used to describe locality.

2.2.1 Iteration Space

This section introduces some important terminology that is later used.

Definition 2.6 **(Iteration Point)** Let us consider an n-dimensional nested loop with loop indexes $I_1, \ldots, I_n$. An execution of the loop body when $I_1 = i_1, \ldots, I_n = i_n$ is identified by the vector $\vec{\imath} = (i_1, \ldots, i_n) \in \mathbb{Z}^n$, which we call iteration point.

Definition 2.7 **(Iteration Space)** The iteration space of an n-dimensional loop nest is the polytope bounded by the bounds of the n enclosing loops, and

represents the collection of all the iteration points. It possesses the (strict) total order, $\prec$, which is known as the lexicographic or dictionary order.

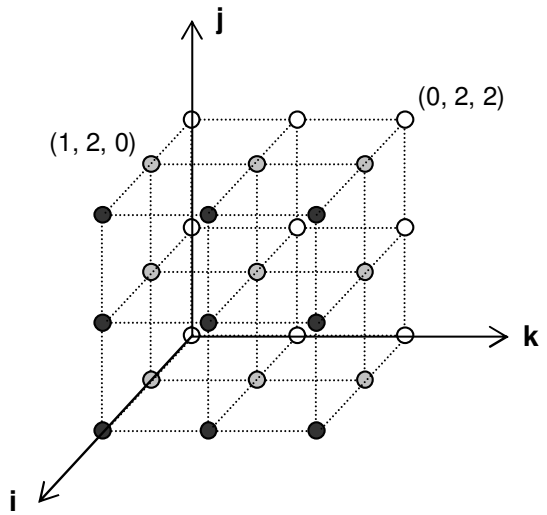


Fig. 2.5: Diagram representing the iteration space of Figure 2.4 when (N=3).

Example 1 *Recall our running example from Figure 2.4. Its iteration space is the set* $\{(i,j,k) \mid 0 \leq i < N, 0 \leq j < N, 0 \leq k < N\}$. *Figure 2.5 illustrates the shape of the iteration space when N is 3. The first iteration point would be* $\vec{\imath}_0 = (0,0,0)$, *and the last one* $\vec{\imath}_{26} = (2,2,2)$. *Using the lexicographic order, we can express that* $\vec{\imath}_0 \prec \vec{\imath}_{26}$.

Definition 2.8 **(Memory access)** Let R be a memory reference. A particular execution of that read or write at run time is a *memory access*. We write $ma_R(\vec{\imath})$ to represent the memory address of that memory access when executing iteration point $\vec{\imath}$. The following expression computes the memory line (ml) and cache set (cs) it is mapped to (we use the modulo-index scheme, see Section 2.1.2):

$$ml_R(\vec{\imath}) = \lfloor ma_R(\vec{\imath})/\mathtt{L} \rfloor \quad (2.1)$$

$$cs_R(\vec{\imath}) = ml_R(\vec{\imath}) \mod \mathtt{N} \quad (2.2)$$

where $\mathtt{N} = \mathtt{C}/(\mathtt{L} \times \mathtt{k})$ is the number of cache sets.

2.2.2 Reuse Vectors

Reuse happens whenever the same data item is referenced multiple times. Trying to determine all iterations that use the same data is extremely expensive. Thus, we use a concrete mathematical representation that describes the direction as well as the distance of the reuse in a methodical way. The shape of iterations that reuses the same data is represented by a *reuse vector space* [168].

Definition 2.9 **(Reuse Vector)** Let R be a memory reference such that $ml_R(\vec{i}_1) = ml_R(\vec{i}_2)$, where $\vec{i}_1 \preceq \vec{i}_2$, we define $\vec{r} = \vec{i}_2 - \vec{i}_1$ as a reuse vector.

Example 2 *Recall reference $c[k][j]$ from our running example. At iteration point $\vec{i}_0 = (1, 2, 3)$, it accesses the array element $c[2][3]$, which is the same array element accessed at $\vec{i}_1 = (3, 2, 3)$. Thus, we have a self-temporal reuse vector $\vec{r} = \vec{i}_1 - \vec{i}_0 = (3, 2, 3) - (1, 2, 3) = (2, 0, 0)$.*

2.2.3 Uniformly Generated References

Definition 2.10 **(Affine Expressions)** An expression in a program is affine if it has the form $c_1 I_1 + \ldots + c_n I_n + b$, where $I_1 \ldots, I_n$ are the loop variables of the n enclosing loops (if any) and $c_1, \ldots, c_n, b$ are compile-time or runtime constants.

Two references are called *uniformly generated* when their array subscripts are affine and differ at most in the constant term [50]. Let n be the depth of a loop nest, and d be the dimensions of an array R. Formally, two references $R(f(\vec{i}))$ and $R(g(\vec{i}))$, where f and g are indexing functions $\mathbb{Z}^n \rightarrow \mathbb{Z}^d$ are called uniformly generated if

$$f(\vec{i}) = H\vec{i} + \vec{c}_f \quad g(\vec{i}) = H\vec{i} + \vec{c}_g$$

where H is a linear transformation and $\vec{c}_f$ and $\vec{c}_g$ are constant vectors.

Example 3 *For instance, reference $a[i][j]$ is uniformly generated with respect to references $a[i][j+1]$ and $a[i-1][j]$, but not with respect to reference $a[j][i]$.*

2.2.4 Classifying Reuse

Let R_p (p for 'producer') and R_c (c for 'consumer') be two uniformly generated references $R_p[H\vec{i} + \vec{c}_p]$ and $R_c[H\vec{i} + \vec{c}_c]$, respectively. Let $\vec{r} \succeq \vec{0}$ be a reuse vector. Thus, applying the definition of reuse vector, R_c at iteration $\vec{i}$ (with the memory

access $R_c[H\vec{\imath} + \vec{c}_c]$) reuses potentially from R_p at $\vec{\imath} - \vec{r}$ (with the memory access $R_p[H(\vec{\imath} - \vec{r}) + \vec{c}_p]$):

$$ml_{R_c}(\vec{\imath}) \quad = \quad ml_{R_p}(\vec{\imath} - \vec{r}) \tag{2.3}$$

The reuse vector $\vec{r}$ represents a *potential* reuse in the cache between the two memory accesses since the memory line touched in the cache at the first access (at $\vec{\imath} - \vec{r}$) may have been evicted from the cache before it gets reused at the second access (at $\vec{\imath}$). $\vec{r}$ is *temporal* (reusing the same element) if the following equality also holds:

$$ma_{R_c}(\vec{\imath}) \quad = \quad ma_{R_p}(\vec{\imath} - \vec{r}) \tag{2.4}$$

and *spatial* (reusing the same cache line but not the same element) otherwise. In addition, the reuse is said to be a *self-reuse* if R_c and R_p are identical and a *group-reuse* otherwise. Thus, there are four kinds of reuse: self-temporal, group-temporal, self-spatial and group-spatial.

- **Self-temporal** A self-temporal reuse takes place when a reference accesses the same data element in different iterations of the loop.
- **Self-spatial** A self-spatial reuse takes place when a reference accesses the same memory line in different iterations of the loop.
- **Group-temporal** A group-temporal reuse takes place when two different references access the same data element.
- **Group-spatial** A group-spatial reuse takes place when two different references access the same memory line.

If we take a closer look to the last definitions, we can see that temporal reuse is a subset of spatial reuse. Whereas the temporal locality can be exploited as many times as we want, the possibility of exploiting spatial locality depends on the cache line size `L`.

2.2.5 Computing the Reuse Vector Space

Given a loop nest, the number of reuse vectors is very large. Wolf and Lam [168] discuss how to compute a basis of the reuse vector space for perfect loop nests with straight-line assignments. Whereas self reuse (both spatial and temporal) and group-temporal reuse are computed in an exact way, group-spatial reuse is only considered among uniformly generated references.

Reusing reference	Reused reference	Reuse Vector
a[i][j] (*R*)	Self-spatial	(0,1,0)
	Self-temporal	(0,0,1)
b[i][k]	Self-spatial	(0,0,1)
	Self-temporal	(0,1,0)
c[k][j]	Self-spatial	(0,1,0)
	Self-temporal	(1,0,0)
a[i][j] *(W)*	a[i][j] *(R)* Group-temporal	(0,0,0)
	Self-spatial	(0,1,0)
	Self-temporal	(0,0,1)

Table 2.1: Reuse vectors for references in Figure 2.4. *R* stands for READ, *W* for WRITE.

If the rows[2] of every array are aligned at the memory line boundaries, Wolf and Lam's reuse framework provides all reuse vectors required. Otherwise, some extra reuse vectors are needed to represent cross-row reuse cases.

As an example, let us derive temporal reuse vectors for the two uniformly generated references $\{R_p = a[i-1][j], R_c = a[i][j]\}$ in an iteration space (i, j). The subscript expressions for both references are affine:

$$\begin{aligned} H\vec{I} + \vec{c_p} &= \left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]\left[\begin{smallmatrix}i\\ j\end{smallmatrix}\right] + \left[\begin{smallmatrix}-1\\ 0\end{smallmatrix}\right] \\ H\vec{I} + \vec{c_c} &= \left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]\left[\begin{smallmatrix}i\\ j\end{smallmatrix}\right] + \left[\begin{smallmatrix}0\\ 0\end{smallmatrix}\right] \end{aligned}$$

In this case, (2.4) becomes:

$$\left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]\left[\begin{smallmatrix}x_1\\ x_2\end{smallmatrix}\right] = \left[\begin{smallmatrix}-1\\ 0\end{smallmatrix}\right]$$

which has the unique solution $(-1, 0)$. Thus, the unique temporal reuse vector from $a[i-1][j]$ to $a[i][j]$ is $(-1, 0)$.

Table 2.1 lists all reuse vectors for the references of our running example shown in Figure 2.4. The group-spatial reuse vector (0,1,0) between $a[i][j](W)$ and $a[i][j](R)$ is not presented since it is redundant with the self-spatial reuse vector.

The reference $a[i][j](W)$ may reuse from the same datum (hence, temporal reuse) that $a[i][j](R)$ (hence, group reuse) accessed at the same iteration. References $a[i][j](R)$, $c[k][j]$ and $a[i][j](W)$ are associated with the self-spatial reuse vector (0,1,0), since they may reuse the same cache line (thus, spatial reuse) that they accessed one iteration before of the loop j. The other reuse vectors can be understood in a similar way.

[2]We assume a row-major data layout.

2.3 Cache Miss Equations (CMEs)

CMEs [52] are an analysis framework that describes the cache memory behavior of perfectly nested loops which consist of straight-line assignments. For each memory reference a set of constraints and equations define those situations where cache misses arise.

The potential reuse is described by the reuse vectors (see Section 2.2.5). In order to find out whether a reuse translates to locality we need to know all data brought to the cache between the two accesses and the particular cache architecture we are analyzing.

CMEs are a set of equations[3] that describe the iteration points where the reuse is not realized. For each reuse vector, two kinds of equations are generated:

- **Compulsory equations:** Given a reference, they represent the first time a memory line is touched. We may distinguish between spatial and temporal reuse:
  - **Temporal reuse:** The reuse is not realized when the studied reference reuses from an iteration point outside the iteration space.
  - **Spatial reuse:** The reuse does not hold when either the analyzed reference reuses from data mapped in another cache line, or the reference reuses data from an iteration point outside the iteration space.
- **Replacement equations:** Given a reference, replacement equations represent the interferences with any other reference, included itself.

Next, we explain each kind of equation in more detail.

2.3.1 Compulsory Equations

There are two different kinds of compulsory equations:

- **Cold Miss Equations (Temporal or spatial reuse).** These equations describe the iteration points where a reuse does not hold because the reference reuses from an iteration point outside the iteration space (see Figure 2.6).

  The Cold Miss Equations put a restriction on the possible values of one of the variables inside the iteration space. For a given $l \in [1, \ldots, m]$, an iteration point $\vec{i}$ and a reuse vector $\vec{r}$, we obtain:

[3] The term equation has been used loosely to represent a set of simultaneous constraints.

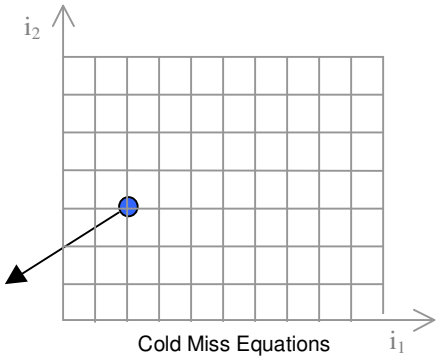


Fig. 2.6: Cold Miss Equations

(CM)

$$\begin{aligned} i_l &\leq d_l \\ lb_k \leq i_k \leq ub_k, &\qquad k = 1 \ldots m \end{aligned}$$

where i_l corresponds to the l-th variable of the iteration space, $d_l = ub_l + r_l \in \mathbb{Z}$, the first equation represents an additional restriction on one of the variables[4] and for each induction variable i_k $(k = 1, \ldots, m)$, ub_k and lb_k stand for the upper and lower bounds of this variable in the iteration space.

Note that this equation could introduce a lower bound of the variable i_k, instead of an upper bound. The other $2m$ constraints determine the iteration space.

Example 4 *Recall our running example from Figure 2.4. In Table 2.1 we have shown that reference $b[i][k]$ has a self-spatial reuse in the direction $\vec{r} = (0, 0, 1)$. Thus, at iteration point $\vec{i} = (0, 0, 0)$, it may reuse from $\vec{i} - \vec{r} = (0, 0, 0) - (0, 0, 1) = (0, 0, -1)$ which is not in the iteration space.*

- **Cold Miss Bounds (Spatial reuse).** These equations describe the iteration points where a reuse is not realized because the reference reuses data that is

[4] $i_l - r_l$ must belong to the domain of the l_th variable of the iteration space.

mapped in a different cache line. Given a reference R_c, there is a cold miss along the reuse vector $\vec{r}$ in the iteration point $\vec{i}$ if the following equation holds:

$$ml_{R_c}(\vec{i}) \neq ml_{R_p}(\vec{i} - \vec{r})$$

where R_p is R_c (if $\vec{r}$ is a self-reuse vector), or a different reference (if $\vec{r}$ is a group-reuse vector).

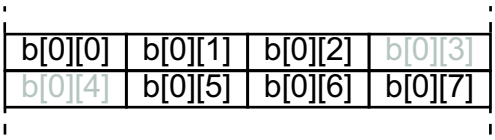


Fig. 2.7: Cold Miss Bounds

Example 5 *Consider reference* $b[i][k]$ *from our running example, and its self-spatial reuse in the direction* $\vec{r} = (0, 0, 1)$*. At iteration point* $\vec{i} = (0, 0, 4)$*, it accesses* $b[0][4]$*, and it may reuse from* $\vec{i} - \vec{r} = (0, 0, 4) - (0, 0, 1) = (0, 0, 3)$*, which accessed* $b[0][3]$*. If we consider a memory layout that maps to cache as shown in Figure 2.7, this reuse is not possible, since* $b[0][3]$ *and* $b[0][4]$ *are mapped to different cache lines.*

2.3.2 Replacement Equations

Given a reference, Replacement Equations represent the iteration points where any other reference accesses data which is mapped onto the same cache set as the data accessed by the current reference.

For each pair of references (R_A and R_B), the following expression gives the condition for a cache set contention in a set-associative cache:

$$\begin{aligned} Cache\_Set(\vec{i})_{R_A} &= Cache\_Set(\vec{j})_{R_B} \\ \vec{j} &\in \mathcal{J} \end{aligned}$$

where $\mathcal{J}$ represents the set of iteration points between $\vec{i}$ (the current one) and the iteration point from which R_A reuses, $\vec{i} - \vec{r}$, where R_B is executed (see Figure 2.8).

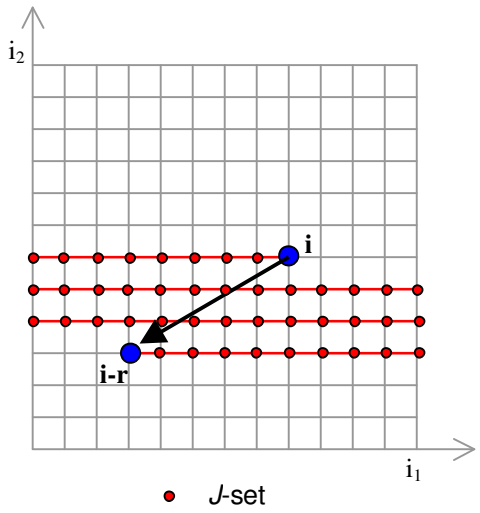


Fig. 2.8: $\mathcal{J}$ set.

Example 6 *Consider references* $a[i][j](W)$ *and* $a[i][j](R)$ *from our running example. According to Table 2.1, the former reuses from the latter along* $\vec{r} = (0, 0, 0)$*. Let us consider iteration point* $\vec{i} = (0, 0, 0)$*. Reference* $a[i][j](R)$ *will reuse data brought by reference* $a[i][j](W)$ *at* $\vec{i} - \vec{r} = (0, 0, 0)$*. Thus,* $\mathcal{J}$ *will be* $\{(0, 0, 0)\}$ *for references* $b[i][k]$ *and* $c[k][j]$*, since it may happen that* $b[0][0]$ *or* $c[0][0]$*, which are executed before* $a[0][0](W)$*, access data that maps to the same cache set as* $a[0][0](R)$*. The set* $\mathcal{J}$ *is* $\emptyset$ *for* $a[i][j](W)$ *and* $a[i][j](R)$*, since they are not executed in the interval that spans between* $a[0][0](R)$ *and* $a[0][0](W)$ *are executed.*

2.3.3 Solving CMEs

Cache miss equations contain precise information about cache behavior, but obtaining the information such as number and causes of the misses is not straightforward.

Each equation represents a convex polyhedron in $\mathbb{R}^n$ [14, 52], where n depends on the type of equation. The integer points inside each convex polyhedron represent the potential cache misses. We may consider two different ways of computing them, either by solving the equations (analytical method) or by checking whether a point is solution or not of the equations that define the polyhedron (traversing the iteration space). Note that the first approach only works for direct-mapped caches, whereas

the second one may be applied to any cache architecture. Next, we describe both approaches in detail.

Analytical method

The solution set of the original CMEs represents the cache misses of a reference, and its volume the number of misses. We construct the solution set based on the following two theorems [52]:

- **Theorem 1:** The set of all misses of a reference along a reuse vector is given by the union of all the solution sets of the equations corresponding to that reuse vector.

Given a reference and a reuse vector, an iteration point produces a miss if it is either a compulsory or a replacement miss.

- **Theorem 2:** The set of all miss instances of a reference is given by the intersection of all the miss-instance sets along the reuse vectors.

Given a reference, an iteration point results in a hit if it exploits the locality of at least one of the reuse vectors.

Thus, given a reference R with m reuse vectors and n_k equations for the k-th reuse vector, the polyhedron that contains all the iteration points that result in a miss is [52]:

$$Set\_Misses = \cap_{k=1}^{m} \cup_{j=1}^{n_k} Solution\_Set\_Equation_j$$

For this approach we need to count the number of points inside the polyhedra, which is an NP-hard problem.[5] Writing the set of misses as a function of the complementary sets, we obtain:

$$Set\_Misses^C = \cup_{j=1}^{n_k} \cap_{k=1}^{m} Solution\_Set\_Equation_j^C$$

where $Solution\_Set\_Equation_j^C$ represents the set of all points in $\mathbb{Z}^n \subset \mathbb{R}^n$ that are not solution to the j-th equation. Thus, according to measure theory, the union of

[5]It is equivalent to deciding whether a solution exists to a system of equalities and inequalities, which is NP-complete [11].

s sets can be computed as follows:

$$\begin{aligned}\mu(\cup_{i=1}^{s} A_i) &= \mu(A_i) + \ldots + \mu(A_s) - \\ &\quad - \sum_{i \neq j} \mu(A_i \cap A_j) + \\ &\quad + \sum_{i \neq j \neq k} \mu(A_i \cap A_j \cap A_k) + \\ &\quad + \ldots + \\ &\quad (-1)^{(i-1)} \mu(\cap_{i=1}^{s} A_i)\end{aligned}$$

where $\mu(P)$ is the number of points inside polyhedron P. As the expression shows, the number of polyhedra that must be counted is 2^s [159], making this problem infeasible due to its huge computing time.

Traversing the iteration space

The second method is based on the fact that every iteration point can be studied independently from the rest of the iteration space.

Given a reference, all iteration points are tested independently, studying the equations in order: from the equations generated for the shortest reuse vector to the equations generated by the longest one [52].

Let us consider a reference R. We study the reuse vectors in a lexicographic ascendent order. Figure 2.9 shows the analysis for a particular reuse vector. After one reuse vector has been treated, some iteration points will be identified as resulting in a miss or a hit. Others rest undetermined and are studied by the remaining reuse vector.

The iteration points are studied as follows:

- If an iteration point is a solution to a compulsory equation, the reuse along the reuse vector $\vec{r}$ is not realized in this iteration point, but we cannot take any definitive decision about the character of this iteration point until all reuse vectors have been studied. Thus, this point will be considered as undetermined.

- If an iteration point is not a solution to any of the compulsory equations, it will be declared as a miss if it is a solution to a replacement equation, and as a hit otherwise.

The algorithm stops when all iteration points have been characterized.

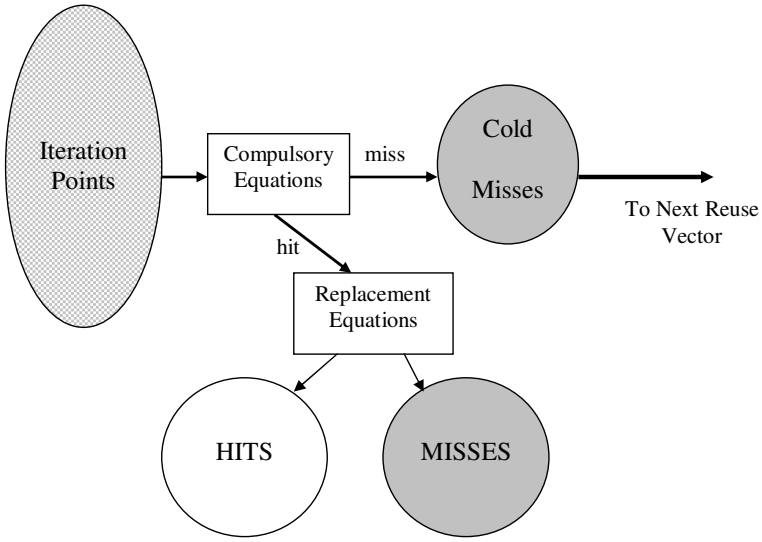


Fig. 2.9: Studying iteration points through a reuse vector.

Counting the total number of solutions is a very time-consuming process, since we may expect very large iteration spaces. In addition, we need to know whether a polyhedron is empty after substituting the iteration point in the equation. This is still an NP-hard problem [11]. Nevertheless, only $s * number\_of\_points$ polyhedra must be analyzed.

This makes it rather impractical to be included in production compilers. In Section 3.4 we introduce a probabilistic method based on sampling [159] to solve the equations in a fast and accurate way. Moreover, we use a highly optimized polyhedra representation [14, 155] to further speed up the process of capturing the distribution of misses.

3

Underlying Model

CHAPTER 3

UNDERLYING MODEL

This chapter describes the different models we have used to model programs' cache memory behavior. We first provide the cache architecture that is assumed. Then, we introduce the program model we use to represent typical programs found in scientific applications and real-time systems. Later, we discuss the statistical model that may be applied to speed up the solving process. Finally, we give an overview of the general framework.

3.1 Architecture Model

We assume a uniprocessor with a memory hierarchy. We focus on multi-level data caches, where each cache level is a `k`-way set-associative cache using an LRU replacement policy. If a cache level is unified, we ignore the effects of the instruction accesses; as the total number of instruction misses compared to data misses is very small, our model is still valid for real architectures. In the case of *write* misses, we assume a fetch-on-write policy so that *writes* and *reads* are modeled identically. Our current analysis assumes a memory hierarchy indexed either by virtual or physical addresses. Some systems index the first level cache virtually, and the upper levels with physical addresses, making cache behavior strongly dependent on page placement [85].

One of our goals is to optimize overall program performance. As the issue rate and depth of pipelining of high performance superscalar processors increase, the importance of control-flow speculation becomes more vital to achieve the potential performance of current processors. There is a serious performance degradation in deep-pipelined machines caused by prediction misses due to the large amount of speculative work that has to be discarded [19]. For that purpose, we also model the branch predictor performance. We consider a *Backward Taken and Forward not Taken* scheme, which has proven to be good enough for our purposes (see Section 6.6).

3.2 Program Model

Our model applies to programs consisting of subroutines, calls, arbitrarily nested loops, and assignments possibly guided by if conditionals.

In this thesis, all programs are written either in C or FORTRAN77. All code snippets are written in a C-like style. When dealing with C codes, all arrays are assumed to be stored in row-major order. Otherwise, they are in column-major order. However, the techniques apply to any *linear* data layout.

The following restrictions define the scope of programs that are analyzable statically:

- Calls are non-recursive.
- All loop bounds, if conditionals and array subscript expressions must be either known or in terms of the loop indices of the enclosing loops.
- The base addresses of all non-register variables including actual parameters (scalars or arrays) must be known at compile time.
- The sizes of an array in all but the first[1] dimension must be known statically.

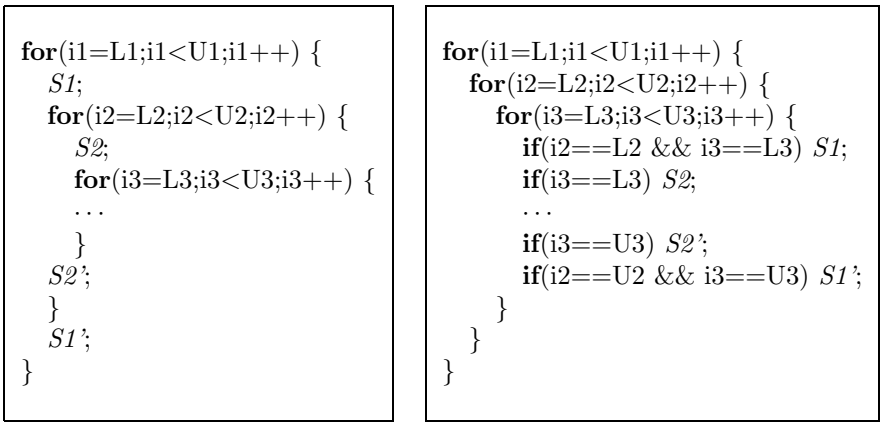
In order to ensure that our analysis can be done in the polyhedral model [44], we add the following constraint:

- The subscript expressions of array references and loop bounds are affine.

Thus, our analytical method can deal with any if conditional involving loop indices and compile-time constants. In loop-oriented programs with regular computations, almost all data-independent conditionals are affine expressions of loop indices and compile-time constants involving possibly operators such as abs, mod, max and min. In all programs that we have analyzed from SPECfp95, Perfect Suite, Livermore Kernels, Linpack, Lapack, and real-time codes, we have not found any if conditional that is data-independent but not also affine. Our program model excludes all and only data-dependent constructs (e.g., variable bounds, data-dependent if conditionals and indirection arrays).

We rely on the compiler to identify compile-time and runtime constants. Detecting these constants allows more expressions to be analyzed statically. Standard compile-time techniques can be very helpful in gathering missing information. To

[1] This applies to C codes. For FORTRAN77 codes it would be the last dimension.



(a) Before sinking (b) After sinking

Fig. 3.1: Loop sinking transformation.

address the symbolic loop bound problem, we use interprocedural constant propagation to eliminate as many symbolic loop bounds as possible. This may also be helpful to know statically which recursive calls are made, thus enlarging the scope where the static analysis is applied.

3.3 Compilation Model

In order to solve the CMEs, we require all the memory references to be in innermost loop nests. Besides, our compilation model needs all the innermost loops to have the same depth. In order to transform user codes in such a way they are suitable for being analyzed, we first apply loop sinking, which moves all references to innermost loop nests. Thereafter, we apply loop normalization, which adds dummy loop nests in order to have all the innermost loop nests in the same depth.

3.3.1 Loop Sinking

We present a strategy to analyze a subset of imperfectly nested loops (see Section 2.3). The original CMEs can only deal with perfectly nested loops. Thus, there are many important imperfectly nested loops that cannot be analyzed (as matrix multiplication with initialization). Our strategy focuses on transforming imperfectly nested loops into perfectly nested loops with conditionals [167]. The loop nests

we consider are those without two loops at the same level. Figure 3.1(a) shows the form of those loop nests.

The technique consists in sinking all the statements to the innermost loop, obtaining a perfectly nested loop. Then, the statements are enclosed within an if statement following Abu-Sufah's non-basic-to-basic-loop transformation [2]. The code obtained is shown in Figure 3.1(b). In the case that the code is already protected by an if statement, this is another constraint that is taken into account. A loop transformation is called *legal* when the transformed code produces the same output as the original one. In order to obtain a loop nest semantically equivalent to the original one, the following conditions must hold [173]:

- The order of the references must be preserved.
- The innermost loop nest must be executed at least once, so if an iteration of a statement would have executed, then it is executed in the transformed program.

Loop sinking may introduce some new memory references that are necessary to evaluate the new conditionals. In order to obtain the cache behavior of the original program, we only analyze the original memory references and assume loop indices are register allocated. Thus, we apply this transformation for analysis purpose, but we do study the original code.

3.3.2 Loop Nest Normalization

Loop normalization consists in adding different for statements that iterate only one iteration, in such a way that all innermost loop nests have the same depth. After normalization, all loop nests are n-dimensional, and, in addition, all loop variables at depth k are normalized [4] to I_k.

Unlike loop sinking, there is no restriction for applying this transformation, since the new statements cannot modify the semantics of the program. Again, we apply this transformation only for analysis purpose, but we do actually study the original code.

3.4 Statistical Model

Even though generating the equations that describe the cache memory behavior is very fast, solving them, i.e., obtaining the information contained in those equations such as number of misses can be time consuming. This may be especially true for

very large iteration spaces. In order to obtain the cache behavior in a reasonable amount of time, we use a statistical approach to determining the cache behavior.

First we present the basic statistical concepts used to model the cache behavior, and secondly the model itself.

3.4.1 Discrete Random Variables

Let $\mathcal{S} = (\Omega, \mathcal{A}, P)$ be a probability space (where Ω is the sample space, $\mathcal{A} \subset \wp(\Omega)$,[2] and P is the probability function). We may define random variables (RV) [37] $X : \Omega \to \mathbb{R}$ over $\mathcal{S}$. We review two of the most studied discrete RV which have been used in our model.

Let X be a real discrete random variable:

- We say that it follows a Bernoulli distribution ($X \sim B(p)$) when the image set has only two elements. Bernoulli RVs describe the random experience in which only two things can happen: success or failure. We define $\mathcal{T} \subset \Omega$ as the set of results obtained that we consider as "success". Thus:

$$\begin{array}{rcl} X : \Omega & \longrightarrow & \mathbb{R} \\ \omega & \longmapsto & \left\{ \begin{array}{ll} 0 & \Longleftrightarrow \ w \notin \mathcal{T} \\ 1 & \Longleftrightarrow \ w \in \mathcal{T} \end{array} \right. \end{array}$$

  The probability $P[X = 1]$ is p. Therefore, the probability $P[X = 0]$ is $q = 1-p$, since $p + q$ must be 1.

- Binomial distribution (represented by $X \sim Bin(n, p)$) models phenomena where n different and independent experiments modeled by Bernoulli-RVs take place. This RV represents the number of successes.

  Once $\mathcal{T} \subset \Omega$ is defined, we obtain:

$$\begin{array}{rcl} X : \Omega^n & \longrightarrow & \mathbb{R} \\ (\omega_1, \ldots, \omega_n) & \longmapsto & card\{i | \omega_i \in \mathcal{T}\} \end{array}$$

  The probability $P[X = k]$, $k = 0 \ldots n$ represents the probability that k experiments over the n succeed. Thus,

$$P[X = k] = \left(\begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{(n-k)}$$

[2] $\wp(X)$, the power set of X, is the set of all the possible subsets of X

3.4.2 Modeling the Cache Behavior with Random Variables

We are interested in finding the number of misses that a program results in (said $\#misses$). In order to obtain it, we model the problem as follows: for each reference we define a RV that returns the number of misses.

We may model the behavior of a reference using a Binomial-RV, where the different experiments consist in taking an iteration point and checking whether it results in a miss.

Next, we prove that this RV actually follows a Binomial distribution. For each memory instruction, we may define a Bernoulli-RV $X \sim B(p)$ as follows:

$$\begin{array}{rccc} X : Iteration & Space & \longrightarrow & \mathbb{R} \\ \vec{i} & & \longmapsto & \{0,1\} \end{array}$$

such that $X(\vec{i}) = 1$ if the memory instruction results in a miss for iteration $\vec{i}$, $X(\vec{i}) = 0$ otherwise. Note that X describes the experiment of choosing an iteration point and checking whether the memory instruction produces a miss for it, and p is the probability of success. The value of p is $p = \frac{\#m}{N}$, where N is the number of times this instruction is executed and $\#m$ the number of misses.

Then, we repeat the experiment N times, using different iteration points in each experiment, obtaining $X_1, \ldots, X_N$ different RV-variables. We note that:

- All the $X_i, i = 1 \ldots N$ have the same value of p.
- All the $X_i, i = 1 \ldots N$ are independent.[3]

The variable $Y = \sum X_i$ represents the total number of misses in all N experiments. This new variable follows a binomial distribution with parameters Bin(N,p) [37] and it is defined over all the iteration space.

3.4.3 Estimation of Parameters

Although a random variable describes a certain property, it may sometimes happen that it is impossible to obtain the parameters that define the RV. This may happen in the cases where population is very large, as in our case, where iteration spaces may have millions of iteration points. In order to overcome this limitation, a subset of the population we try to describe can be analyzed and the results obtained can be inferred to the population. Now, we explain how the parameters that describe a Binomial-RV can be inferred.

[3] We assume that the iteration space is sampled in an independent way.

Let $X \sim Bin(n, p)$, and assume that p (the probability of success) is unknown. We obtain an approximation of p evaluating the behavior of a subset of the population (called sample). The RV that describes the property we are interested in is then computed for the sample. Finally, we infer the sample-RV parameters to the population-RV.

Let $\mathcal{Q} \subset \Omega^n$ be the sample, $N = card(\Omega^n)$ and $k = card(\mathcal{Q})$. The value $\hat{p}$ is defined as

$$\hat{p} = \frac{successes \in \mathcal{Q}}{k}$$

If the sample is randomly chosen among the population, the RV that describes the behavior of the sample is $Y \sim Bin(k, \hat{p})$, and we have that:[4]

$$\frac{(\hat{p} - p)}{\sqrt{\frac{pq}{k}}} \sim N(0, 1)$$

provided that the sample does not contain repeated elements and the following conditions hold [37]:

- $\frac{k}{N} \leq 0.05$
- $\hat{p}k \geq 5$ and $1 - \hat{p}k \geq 5$
- $k \geq 30$

Once a confidence level[5] is chosen, a confidence interval for the value of p is given by the following expression, where $\alpha = 1$-*confidence*:

$$p \in \hat{p} \pm z_{\frac{\alpha}{2}} \sqrt{\frac{\hat{p}(1 - \hat{p})}{k}}$$

3.5 Framework Overview

Figure 3.2 depicts the structure of our framework. We show the different steps involved in our analysis, as well as the interactions among all components. Our framework consists of three big components represented by dashed boxes, which are fully automated and performed during compilation.

First, the program being analyzed is translated into an internal representation, which we call CMEs IR. Currently, there are available three different front-ends:

[4] Z∼N(0,1) is the Normal or Gauss distribution

[5] e.g., if the percentage is 95% (0.95), it represents that for 95 out of every 100 different samples, $\hat{p}$ will belong to the confidence interval

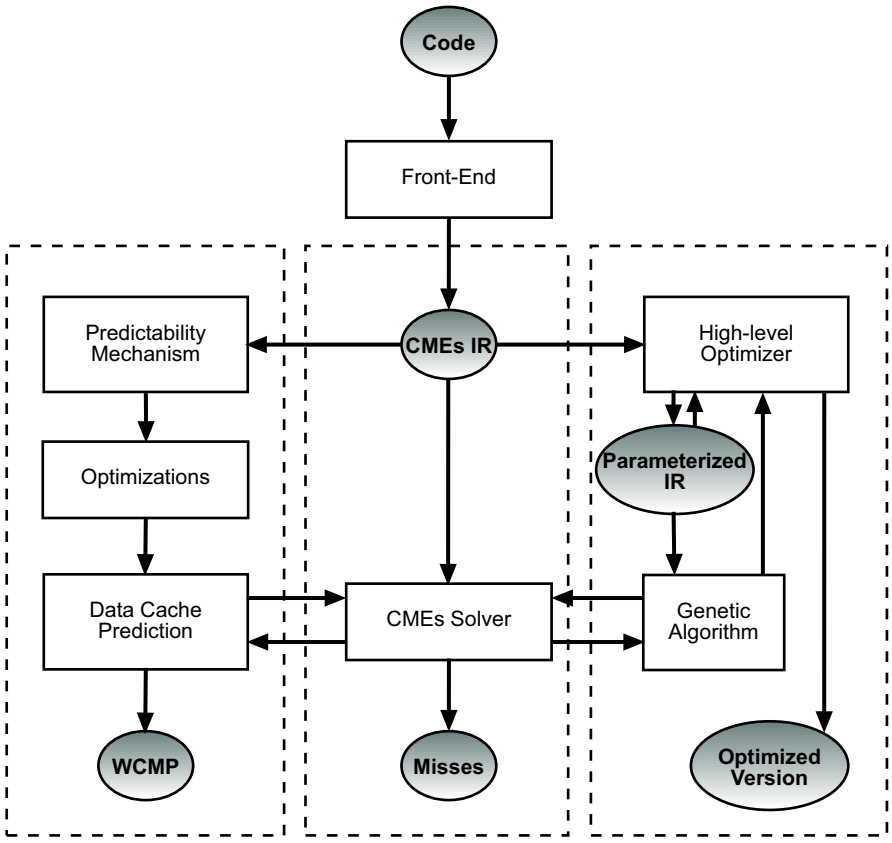


Fig. 3.2: General framework.

- **Polaris**. The program is translated by a Polaris compiler into Polaris IR [42], which is subsequently translated by Ictineo [8] into a so-called Ictineo load/store IR. The lowering of the Polaris IR serves two purposes. First, some standard compiler optimizations such as constant propagation and induction variable elimination are performed so that the estimated miss ratio is a realistic representation of the miss ratio of the compiled code. Second, the load/store array references are identified and the scalars and temporaries mapped to virtual registers. Once both are done, the load/store IR is converted to an optimized Polaris IR in which all load/store array references are clearly indicated.

- **SUIF1 & SUIF2**. The program is translated by the corresponding SUIF [111] compiler into a high-level SUIF IR. The SUIF infrastructure includes many of

Compiler	C	C++	FORTRAN77
Polaris			✓
SUIF1	✓		✓
SUIF2	✓	✓	

Table 3.1: Different front-ends used in this thesis.

the standard compiler optimizations, which allows us to obtain a code competitive to production compilers. Using SUIF, we identify high-level information (such as array accesses and loop constructs) that can be further passed down to the low-level passes as annotations.

It should be pointed out that our model is not limited to either internal representation. The model can be applied to any IR as long as the information required by the model is available. Table 3.1 shows which programming languages are supported by each of the available front-ends.

Once the program has been translated into the IR, different steps may be applied. The three main steps are:

- **Cache analysis.** The key component is the one that computes the equations and solves them, which in fact describes the cache behavior. It basically consists of components inlining calls (abstractly), obtaining reuse vectors, forming the mathematical expressions that describe cache misses and solving them.

- **Optimizing w.r.t average execution time.** We use a genetic algorithm with an accurate cost model function to reduce programs' execution time. The genetic algorithm generates different solutions, which are analyzed by our CMEs solver to compute potential cache misses for each level of cache.

- **Optimizing w.r.t predictable timing.** We introduce a method that combines both static cache analysis and cache locking in order to achieve both predictability and good performance. Furthermore, it allows computing a worst-case memory performance `WCMP` estimate of tasks in a fast and tight way. Our approach first transforms the original program issuing lock/unlock instructions to ensure a tight analysis of the `WCMP` at static time. In order to keep a high performance, load instructions are added when necessary. Later, the actual computation of the `WCMP` estimate is performed.

4

Experimental Framework

CHAPTER 4

EXPERIMENTAL FRAMEWORK

This chapter presents the architectures used to verify and obtain performance results for our approach. We also introduce the benchmark applications and the simulation environment.

4.1 High-Performance Architectures

We have considered four modern processors that represent current architectural paradigms. To experiment with our optimizations, we have considered the Pentium-4 [23] (CISC), Alpha-21264 [86] and UltraSparcIII [140] (RISC), and Itanium [135] (EPIC). Next, we review in detail the architectures adopted by the processors used in our study.

4.1.1 Pentium-4

Pentium-4 is Intel's most advanced processor for desktop PCs. It uses the hyper-pipelined technology in order to increase frequency and scalability of the processors, which leads to a 20-stage pipeline. It includes an *execution trace cache* that stores up to 12K decoded micro-operations. This increases performance by removing the decoder from the main execution loop. Additionally, the trace cache increases the instruction fetch rate since instructions are stored following the dynamic order instead of the static order.

Memory hierarchy. Caches are physically-indexed and physically-tagged (PIPT). The second level on-die cache consists of a 256-bit (32-byte) interface that transfers data on each core clock. As a result, it can deliver a data transfer rate of 48 GB/s. The processor always reads a cache line from the memory system on a 64-byte boundary. A cache line can be filled from memory with a 8-transfer burst transaction. Caches do not support partially-filled cache lines. Since caches are write-through, all writes are write out to memory.

Branch Predictor. It dynamically predicts the direction and target of branches using a 4K branch target buffer combined with a branch history table. If no dynamic prediction is available, it predicts statically (taken for backwards looping branches, not taken for forward branches). Besides, it makes use of the trace cache to alleviate the cost of a miss-predicted branch.

4.1.2 Alpha-21264

The 21264 (EV6) is the third generation superscalar Alpha microprocessor. Four instructions are fetched every cycle to be delivered to the out-of-order execution engine, whose pipeline has 6 stages. The memory system is high-bandwidth, supporting many in-flight memory references.

Memory hierarchy. The L1-cache is virtually indexed, physically tagged (VIPT). Obviously, the off-chip L2 cache is PIPT. The L1 data cache operates at twice the frequency of the processor clock, which gives full support for two memory references every cycle without conflict. The L2 cache is a unified off-chip cache. The Alpha processor supports mini-speculations: in order to achieve a low first level cache latency, it is necessary to speculatively issue consumers of integer load data before knowing if the load hit or missed in the cache.

Branch Predictor. It uses a hybrid predictor that dynamically chooses between two types of branch predictors. Both predictors are two-level predictors. One of them has 1024 branch-history registers in the first level and 1024 pattern-history entries in the second level. The other predictor has a single branch-history register in the first level and 4096 pattern-history entries in the second level. The choice predictor is implemented by means of a 4096-entry table that is indexed by the global branch-history register.

4.1.3 UltraSparc-III

The UltraSPARC-III Cu processor is a high-performance superscalar processor that implements the 64-bit SPARC V9 RISC architecture. It is a 4-way superscalar processor with nine execution units and six execution pipelines, which can sustain the execution of up to four instructions per cycle. The non-stalling pipeline has 14 stages, with 64-bit data paths, 64-bit ALUs and 64-bit address arithmetic.

Memory hierarchy. The L1 cache is VIPT. It is write-through and write no allocate. The L2 cache is off-chip and unified. In order to minimize the number of transactions, it incorporates a 2 KB, 4-way set associative PIPT prefetch cache for software prefetch, and a 2 KB, 4-way set associative PIPT write cache that reduces store bandwidth to L2 cache.

Branch Predictor. It uses a modified *gshare* [112].

4.1.4 Itanium

The Itanium processor is the first implementation of the IA-64 instruction set architecture. The new processor employs EPIC (explicitly parallel instruction computing) for a tighter coupling between hardware and software; the hardware-software interface lets the software exploit all compile-time information and deliver this information to the hardware. Itanium provides a 6-wide and 10-stage pipeline, with 4 integer units, 2 load/store units and 4 floating-point units. This gives abundant resources to exploit instruction-level parallelism (ILP).

Memory hierarchy. Three levels of PIPT on-chip cache minimize the overall impact of memory latency. The L3 cache, which is accessed at core frequency, can provide over 12GB/s. L2 and L3 are unified, whereas there are two split L1 caches.

Branch Predictor. It has the most sophisticated branch prediction mechanism among the processors considered in our study. It consists of a hierarchy of three levels of branch predictors, which need an important input from the compiler in order to achieve a high hit rate [135].

Level 1 It is a branch target buffer which uses hints set by the compiler to decide the branch direction.

Level 2 For scalar codes, the processor uses a 2-level prediction scheme. A return stack buffer provides predictions for return instructions.

Level 3 Defined as *branch address calculation and correction* level, it applies a correction for the exit condition of modulo-scheduled loops. It uses a special structure that keeps track of the loop count obtained during the loop initialization. However, it does not work well with tiling due to the *min* expressions in the loop bounds.

Processor	Freq.	L1 (C,L,k)	L1 Replacement	H	L2 (C,L,k)	H/M	P
Pentium-4	1.6GHz	(8,64,4)	LRU	2	(512,128,8)	24/150	20
Alpha-21264	525MHz	(64,64,2)	FIFO	3	(4096,64,4)	6/84	7
UltraSparc-III	750MHz	(64,32,4)	Random	2	(8192,512,4)	10/100	8
Itanium	800MHz	(96,64,6)	LRU	2	(2048,64,4)	21/117	15

Table 4.1: High-performance processors used for the experimentation. C stands for cache size in KB, L stands for cache line size in bytes, and k stands for the degree of associativity. H/M is the hit/miss cycles for each cache level. P is the misprediction penalty.

Name	Freq.	L1(C,L,k)	H/M
microSPARC II-ep	100MHz	(8,16,1)	1/10
PowerPC 604e	300MHz	(16,32,4)	1/38
MIPS R4000	250MHz	(16,16,1)	1/40
IDT 79RC64574	200MHz	(32,32,2)	1/16
NEC V850E/ME2	100MHz	(8,16,2)	1/10

Table 4.2: Microprocessors and microcontrollers used for the experimentation. C stands for cache size in KB, L stands for cache line size in bytes, and k stands for the degree of associativity. H/M is the hit/miss cycles for each cache level.

4.2 Embedded Processors

Embedded systems are commonly considered very specific systems based on a microprocessor with memory integrated on the chip. Nowadays, emerging areas like car computers demand real-time capabilities but at a relative low cost. Yet, some of these new applications require a high throughput.

Microprocessor companies are moving from 4-, 8- and 16-bit processors to the current 32-bit architectures. This includes the new ARM processors, some downscaled x86, and simplified PowerPC and MIPS processors. Caches are mainly used on many high-end embedded 32-bit processors. We have chosen a set of modern processors widely used in the real-time area: microSPARC-IIep [139], PowerPC 604e [119], IDT 79RC64574 [75] and MIPS R4000 [117]. We also include results for the recently released NEC V850E/ME2 32-bit high-performance microcontroller.

Name	Description
Analysis	
MGRID_K	3-D loop nest from MGRID (SPECfp95)
HYDRO_K	Three 2-D loop nests from kernel 18 (Livermore)
MMT	3-D blocked loop nest that computes $A \times B^T$ [48]
MMI	Matrix multiplication with initialization as shown in [63]
LWSI_K	4-D imperfect loop nest from LWSI
APPLU	Parabolic/elliptic partial differential equations
Optimization	
MATMUL	Matrix multiplication
MATVEC	Matrix vector multiplication
T2D	2D matrix transposition
ADI	2D ADI integration
VPENTA	Invert 3 pentadiagonals
Analysis & Optimization	
TOMCATV	Generation of grids
SWIM	Solver for the shallow water equations

Table 4.3: Benchmarks used to evaluate our optimization framework.

4.3 Summary of Characteristics

We summarize the characteristics of the processors used in two tables. Cache latencies and miss-prediction penalties[1] have been obtained from vendors' specifications.

Table 4.1 shows the configurations of the high-performance machines used in our analysis, whereas Table 4.2 describes the embedded processors. For each of them, we give the frequency and cache memory configuration that apply to each family. Note that for the Itanium processor, we ignore the L1 cache since it only holds scalars.

4.4 Applications

The benchmarks evaluated in this study are all applications widely used in the high-performance and real-time area. They are drawn from several benchmark suites. The loop nest fragments and complete programs considered are from SPECfp95, Perfect Suite, Livermore kernels, Linpack, Lapack, NAS and BIHAR libraries. Regarding the real-time kernels, this collection includes both kernels operating on arrays and scalar programs, such as SQRT and FIBONACCI. We have also used FFT to show the feasibility of our approach for typical DSP codes.

[1] It is basically the length of the pipeline.

An overview of the five kernels that have been used to evaluate our analysis and optimization techniques can be seen in Table 4.3. For all of them, we have studied a set of different sizes that are reasoned through the thesis. The source code of all of them can be found in Appendix A and Appendix B.

Name	Description
MM	Multiply two 100x100 Int matrices
CNT	Count and sum values in a 100x100 Int matrix
ST	Calc Sum, Mean, Var (2 arrays of 1000 doubles)
SQRT	Computes square root of 1384
FIB	Computes the first 30 Fibonacci numbers
SRT	Bubblesort of 1000 double array
NDES	Encrypts and decrypts 64 bits
FFT	Fast Fourier transformation of 512 complex numbers

Table 4.4: Real-time benchmarks used.

Table 4.4 gives an overview of the 8 real-time programs considered. They are all written in C, drawn from different real-time papers that analyze data cache behavior [5, 87, 162].

4.5 Environment

To check the accuracy of our method, we compare our results against a simulator. We have used a locally trace-driven simulator [153], which has been validated over the years against the well-known Dinero III [69]. We have modified it to handle locking caches and traces from different tasks. We have validated the new features with micro-benchmark simulation, running small kernels and comparing the results with the expected ones. Traces only contain load/store of data and lock/unlock instructions, and are tagged with a task ID.

All programs are compiled and linked using the gnu tools (gcc and g77) with the optimization level set to "-O3". The base addresses and access order of the references are obtained from the binaries. Hence, the miss ratio of a program is sensitive to the particular binary being used. However, for the sake of fairness, both the simulated and the analyzed codes have the same accesses executed in the same order. Figure 4.1 details the steps used to compare the simulator and the analytical methods.

The actual number of misses when executing programs on the Pentium-4 platform are obtained by means of the performance counters. We have measured the

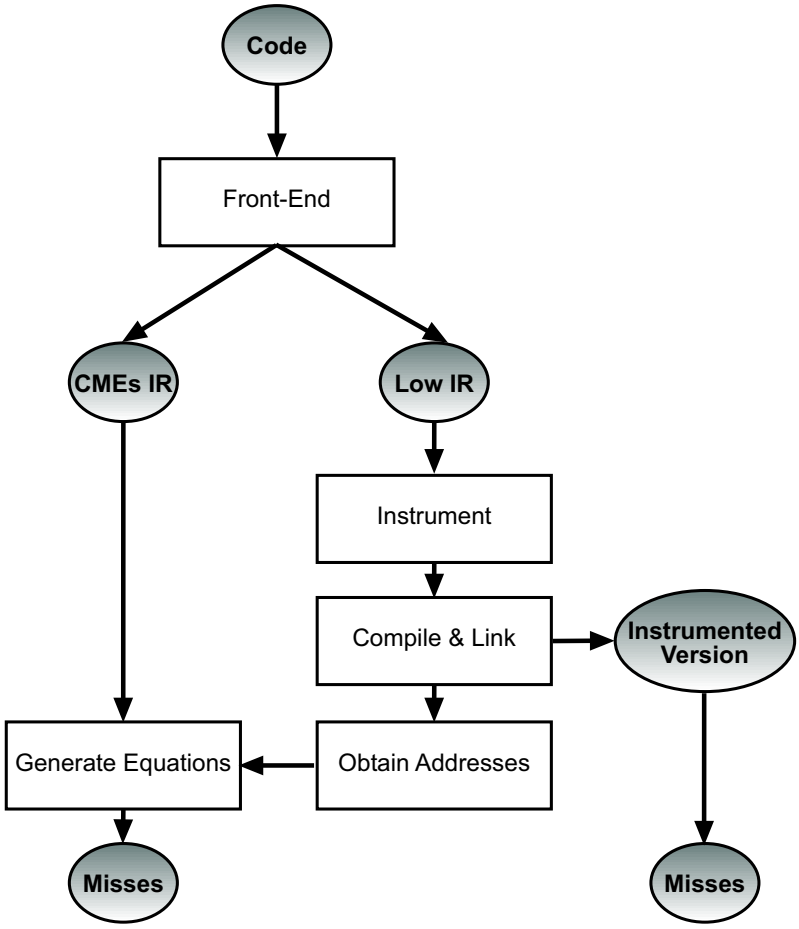


Fig. 4.1: Instrumentation framework.

events *L1 load misses retired* and *L2 load misses retired*.

For an evaluation of the ability of our model to predict actual cache miss ratios, we refer the interested reader to [155]. In that work (which is not part of this thesis), we have run different kernels on a modern out-of-order processor and quantified the error between predicted and actual cache miss ratios.

5

Whole-Program Analysis

CHAPTER 5

WHOLE-PROGRAM ANALYSIS

In order to model cache behavior we must be able to identify the following information: (i) the first time a memory line is brought to cache, (ii) two consecutive accesses to a memory line, and (iii) all intervening accesses between such a pair of consecutive accesses. We first introduce our compile-time framework for achieving these three goals, which consists of abstract call inlining, iteration vectors and reuse analysis. *Abstract inlining* allows having a representation of all memory references which is not context sensitive. We use the *iteration vectors* to represent exactly the address trace of the program, whereas the *reuse analysis* describes the potential reuses among all references. Then, we present the techniques used to obtain the cache behavior based on the new reuse description. A method to generate and solve the equations that represent cache behavior for whole programs is also described.

5.1 Abstract Inlining

In an attempt to analyze exactly a program containing call statements, we perform an *abstract inlining* for a call whenever possible. We do not actually generate the inlined code. We only need to obtain the information required for analyzing the inlined code. Each subroutine is associated with an *abstract function* consisting of the information about the memory accesses to the runtime stack, its code body (i.e., its loop nests with references), and local variable and formal parameter declarations.

The two issues addressed by the abstract inlining are discussed below.

- **Modeling of Accesses to the Runtime Stack.** The calling conventions are compiler- and architecture-dependent. What is shown in Figure 5.1 is one such a convention for a 32-bit machine, in which all actual arguments are passed via the runtime stack. *Stack* denotes the runtime stack modeled as a 1-D array. If BP (base pointer to the stack) is 0 initially, its value is known at compile time at every call site due to the absence of recursion (see our program model

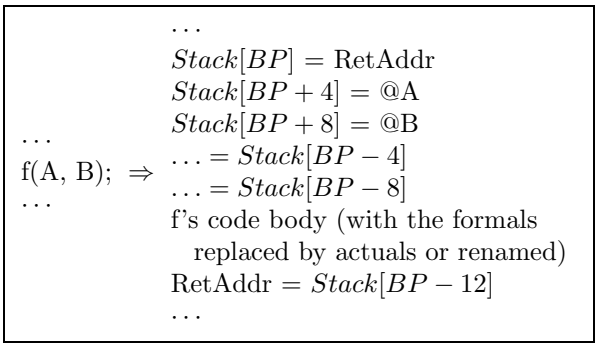


Fig. 5.1: Abstract inlining of a subroutine call.

in Section 3.2). The base address of *Stack*, if unknown at compile time, has to be obtained at run time. Then *Stack* is treated just like an ordinary array.

- **Transformation of Array References in the Callee.** References to the dummy arrays are transformed so that the subscript expressions of the matching actuals are incorporated into the transformed references. The references to the non-dummy arrays remain unchanged. The inlined code may not compile. Hence, the term abstract inlining.

The inlined code is statically analyzable if the original program is. The inlined code contains the same memory accesses executed in the same order as in the original program; the abstract inlining has not modified anything in the original program.

In our current implementation, system calls (to I/O subroutines and intrinsic functions) are not inlined. The memory accesses inside are not accounted for. These calls can be inlined if their abstract functions are known. Without loss of generality, from now on we assume that all parameters are passed by reference.

5.1.1 Transforming Array References

Not every call can be inlined according to Table 5.1. In order to analyze a call exactly, our method needs to know at compile time the base addresses of all its actual parameters. Let AP be an actual parameter that is either a scalar or an array variable or a subscripted variable with an affine data access expression and FP be its matching formal parameter.

AP is *propagateable* if, after inlining, every reference to FP can be replaced by a reference to AP. This allows the reuse to AP both in the caller and in all the callees

Program	Actual Parameters	Calls
	P-/R-/N-able	$\mathbb{A}/\mathbb{T}$
tomcatv	0/0/0	0/0
swim	0/0/0	5/5
su2cor	503/87/0	150/150
hydro2d	122/0/19	82/82
mgrid	68/0/35	2/23
applu	79/0/0	23/23
apsi	1601/0/210	118/186
fppp	83/0/3	16/17
turb3D	759/0/75	86/111
wave5	591/2/110	127/171

(a) SPECfp95

Program	Actual Parameters	Calls
	P-/R-/N-able	$\mathbb{A}/\mathbb{T}$
CSS	2489/0/8	965/965
LWSI	140/0/19	18/28
MTSI	186/0/2	63/63
NASI	236/0/237	41/75
OCSI	620/0/48	209/244
SDSI	189/18/49	103/129
SMSI	321/0/41	38/53
SRSI	242/0/176	13/50
TFSI	137/0/91	13/44
WSSI	836/127/7	179/185

(b) Perfect

Overall	P-/R-/N-able	$\mathbb{A}/\mathbb{T}$
TOTAL	9202/234/1130	2251/2604
%	87.09/2.21/10.89	86.44/100

Table 5.1: Statistics for the actual parameters and calls in SPECfp95 and Perfect benchmarks. $\mathbb{T}$ stands for the total number of calls, whereas $\mathbb{A}$ stands for all analyzable calls.

to be potentially exploited. In column "P-able", we consider AP as propagateable if FP is a scalar, or a one-dimensional array or if both AP and FP are arrays of the same dimensionality with matching sizes in all but the first[1] dimension.

Example 7 *Let {a[N][M], a[I1][I2]} be the actual parameter (a is an array whose dimensions are N and M), and {s[N][M], s[I3][I4+1]} the formal parameter. We are interested in the memory address accessed by the formal parameter. Let* `SOF` *be the size of the elements of the array. The memory address is given by the expression:*

$$@s + I3 * \texttt{SOF} * M + (I4 + 1) * \texttt{SOF}$$

Since parameters are passed by reference, @s is the memory address of the actual parameter. This is, $@a + I1 * \texttt{SOF} * M + I2 * \texttt{SOF}$*. Substituting, we obtain:*

$$@a + I1 * \texttt{SOF} * M + I2 * \texttt{SOF} + I3 * \texttt{SOF} * M + (I4 + 1) * \texttt{SOF}$$

Finally, grouping, we obtain that it is propagateable:

$$@a + (I1 + I3) * \texttt{SOF} * M + (I2 + I4 + 1) * \texttt{SOF} = a[I1 + I3][I2 + I4 + 1]$$

[1]For FORTRAN77 codes it would be the last dimension.

AP is *renameable* if, after inlining, every reference to FP can be replaced by a reference to AP' such that AP and AP' have the same base address (i.e., $@AP = @AP'$). Notice that by definition, propagateable actuals are also renameable. In column "R-able" we account for actuals that are classified as renameable and not as propagateable. We consider AP as renameable if the sizes of all but the first dimension for AP and FP are known statically. This still allows the reuse between the references to FP in the same subroutine to be exploited (but not the reuse between the caller and the callee.)

Example 8 *Let {a[N][M], a[I1][I2]} be the actual parameter (a is an array whose dimensions are N and M), and {s[N'][M'][P]), s[I3][I4][2])} the formal parameter. We are interested in the memory address accessed by the formal parameter. Let* SOF *be the size of the elements of the array. The memory address is given by the expression:*

$$@s + I3 * \texttt{SOF} * M' * P + I4 * \texttt{SOF} * P + 2 * \texttt{SOF}$$

Since we assume parameters are passed by reference, @s is the memory address of the actual parameter. This is, $@a + I1 * \texttt{SOF} * M + I2 * \texttt{SOF}$. *Substituting, we obtain:*

$$@a + I1 * \texttt{SOF} * M + I2 * \texttt{SOF} + I3 * \texttt{SOF} * M' * P + I4 * \texttt{SOF} * P + 2 * \texttt{SOF}$$

Finally, grouping and renaming S by S1, we obtain:

$$@a + I3 * \texttt{SOF} * M' * P + I4 * \texttt{SOF} * P + (2 + I2 + I1 * M) * \texttt{SOF} =$$

$$= s1[I3][I4][I2 + 2 + I1 * M]$$

Column "N-able" accounts for the actuals that are neither propagateable nor renameable, named *non-analyzable*. The propagateable and renameable actuals are potentially *analyzable* since all references to FP can be analyzable if affine.

A call can be abstractly inlined, i.e., is potentially *analyzable*, if all its actuals are analyzable. Table 5.1 shows that we can inline 86.44% of calls from SPECfp95 and Perfect benchmarks. These statistics are obtained at compile time by examining only a call and its callee.

Figure 5.2 serves to illustrate the inlining of a code segment (which may have out of array bound accesses if loop bounds are not chosen properly). The inlined code does not compile (dimensions of the arrays declared in the main program should be statically known) but can be analyzed by our method. Hence, the name abstract inlining.

```
double x, a[10][10], b[20][20];
for(I1 = ...) {
  for(I2 = ...) {
    a[I1][I2] = ···;
    f(x, a, b, b[I1][I2]);
    g(a[I1][I2], a[1][I2], b);

void f(y, c, d, s) {
double y, c[10][10], d[400], *s[10][10];
for(I3 = ...) {
  for(I4 = ...) {
    c[I3][I4 − 1] = y + d[I3 − 1 + 20 ∗ (I4 − 1)];
    s[I3][I4][2] = ···;

void g(e, f, t)
double E[10][10], F[10], T[100, 4];
for(I3 = ...) {
  for(I4 = ...) {
    e[I3][I4] = f[I4] − t[I3][I4];
```

⇓

```
double x, a[10][10], b[20][20];
//THE FOLLOWING LINE DOES NOT COMPILE
double b1[*][10][10], b2[100][4];
for(I1 = ...) {
  for(I2 = ...) {
    a[I1][I2] = ···;
    for(I3 = ...) {
      for(I4 = ...) {
        a[I3][I4 − 1] = x + b[I3 − 1 + 20 ∗ (I4 − 1)];
        b1[I3][I4][I2 + 2 + I1 ∗ 20] = ···;
    for(I3 = ...) {
      for(I4 = ...) {
        a[I1 + I3][I2 + I4] = a[I4 + 1][I2] − b2[I3][I4];
```

Fig. 5.2: Propagation and renaming of actual parameters. All actuals but the last are propagated. The last actuals in both calls are renamed to $b1$ and $b2$, respectively. After inlining, $@b = @b1 = @b2$.

5.2 Iteration Vectors

The inlined program has a flat structure consisting of multiple loop nests without any calls. The iteration vectors are defined to introduce a total order among all the memory accesses from both within a nest and across nests in the abstractly inlined program. They define precisely the address trace of the program at compile time.

After applying loop normalization (see Section 3.3.2), we have statements distributed among innermost loops. The access of a reference in the program can be uniquely identified by (a) the loop nest in which the reference is contained, (b) the iteration of the nest at which the reference is accessed, and (c) the access order of the reference in the nest.

A particular instance of a statement S (known as an *iteration* or *iteration point*, see Section 2.2.1) of the enclosing loop nest is identified by a $2n$-dimensional *iteration vector* of the form $\vec{\imath} = (\ell_1, I_1, \ell_2, I_2, \ldots, \ell_n, I_n)$, where

- $\vec{L} = (\ell_1, \ell_2, \cdots, \ell_n)$ is the *loop label* (*vector*) for the innermost loop containing S, and
- $\vec{I} = (I_1, I_2, \ldots, I_n)$ is the *index vector* consisting of the indices of the n loops enclosing S.

L_1:	**for**$(I_1 = 0; I_1 < N; I_1 + +)$ {	Iteration Vector
L_2:	**for**$(I_2 = 1; I_2 < N; I_2 + +)$ {	
	if$(I_2 > 2)$	
S_1:	$b[I_1 - 1][I_2] = \cdots;$	$(1, I_1, 1, I_2)$
L_3:	**for**$(I_2 = \ldots)$ {	
S_2:	$\cdots = b[I_1][I_2];$	$(1, I_1, 2, I_2)$
L_4:	**for**$(I_1 = \cdots)$ {	
L_5:	**for**$(I_2 = \cdots)$ {	
S_3:	$b[I_2][I_1] = \cdots;$	$(2, I_1, 1, I_2)$

Fig. 5.3: Iteration vectors for statements.

Figure 5.3 lists the iteration vector for each statement in the example. It is not difficult to see how the iteration vectors are derived in general.

Let $\vec{\ell}_L$ be the loop label for loop nest L. Since we have applied loop normalization, all the loop labels are n-dimensional, where n is the depth of the deepest loop nest. The i-th entry of the loop label, if defined, is the order of the loop nest in the i-

depth.[2] Otherwise, we use the undefined value $\perp$. Using this formulation, we obtain the following loop labels for the example in Figure 5.3:

L_1:	$\vec{\ell}_{L_1}$=(1,$\perp$)
L_2:	$\vec{\ell}_{L_2}$=(1,1)
L_3:	$\vec{\ell}_{L_3}$=(1,2)
L_4:	$\vec{\ell}_{L_4}$=(2,$\perp$)
L_5:	$\vec{\ell}_{L_5}$=(2,1)

As usual, the set of all iterations for a particular loop nest is called the *iteration space* of that nest (see Section 2.2.1).

Definition 5.1 **(Access Order of References)** Let S_L be the set of all references in a loop nest L. We define $\alpha : S_L \rightarrow \mathbb{Z}^+$ such that $\alpha(R) = k$, meaning that R is the k-th accessed reference (with any guarding IF conditional ignored) in a common iteration of L.

In a sequential execution, all iteration points are executed in lexicographical order. The usual lexicographic order operators $\prec$, $\preceq$, $\succ$ and $\succeq$ are used later.

5.3 Reference Iteration Spaces

The *reference iteration space* of a reference R is defined as the set of iteration points where the reference is accessed.

Definition 5.2 **(Reference Iteration Spaces)** The reference iteration space (*RIS*) of a reference R, denoted $RIS_R \subset \mathbb{Z}^n$, is the n-dimensional polytope defined by the bounds of its n enclosing loops and its guiding if conditional, if any.

If a reference is not guarded by a conditional, its *RIS* is the entire iteration space of the enclosing loop nest. Otherwise, the *RIS* can be a subspace of that iteration space.

If a reference is guarded by affine conditionals (containing possibly and, or, equality or not operators), the corresponding *RIS* can always be expressed as a finite union of convex polytopes in $\mathbb{Z}^n$. Such a *RIS* can be manipulated by the Omega library [127] and its volume computed using methods [34, 61, 127] for various purposes. Figure 5.4 depicts three commonly occurring cases.

[2] This can be seen as the numbering of sections and subsections in a paper.

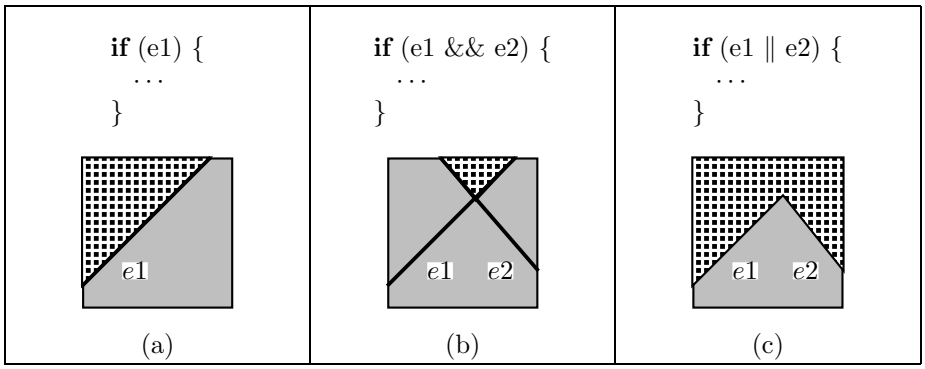


Fig. 5.4: Some commonly occurring *RISs* (in dotted areas).

Example 9 *Consider the statement S_1 in Figure 5.3. Applying the definition of RIS, we obtain:*

$$RIS_{(b[I_1-1][I_2])} = \{(i_1, i_2) \mid 0 \leq i_1 < N, 2 < i_2 < N\}$$

The set of all accesses of a reference R contained in the nest $(\ell_1, \ldots, \ell_n)$ is defined by:

$$\mathcal{M}_R \;=\; \{(\ell_1, i_1, \ldots, \ell_n, i_n, \alpha_{(l_1,\ldots,l_n)}(R)) \mid (i_1, \ldots, i_n) \in RIS_R\}$$

If *RefSet* is the set of all references, then the set of all accesses in the program is given by:

$$\mathcal{M} \;=\; \bigcup_{R \in RefSet} \mathcal{M}_R$$

The accesses in $\mathcal{M}$ ordered by $\prec$ give rise to precisely the address trace of the program at compile time.

Our analytical method can deal with conditionals that can be analyzed at compile time without relying on any runtime information about the conditionals involved. However, data-dependent conditional expressions such as *a[i][j]==0*, which are supposed to be unknown until run time, are beyond any static cache analysis. In Section 7 we show how these statements introduce unpredictability, and discuss our approach to eliminating them in order to compute the worst-case scenario.

5.4 Reuse Analysis

Wolf and Lam [168] discuss how to compute reuse vectors for perfect loop nests with straight-line assignments, assuming all *RISs* are the entire iteration space (see

Section 2.2.5). Later, Xue and Huang [174] describe an extension to allow non-elementary reuse vectors to be represented exactly.

We generalize Wolf and Lam's reuse framework [168, 174] to calculate reuse vectors across different *RISs*, including multiple nests and conditionals. We also add additional spatial reuse vectors to capture the reuse spanning two adjacent rows[3] of an array. Finally, we provide some discussions on the accuracy of our approach.

5.4.1 Parametric Reuse Analysis

Reuse analysis is concerned with identifying two consecutive accesses to a memory line. When we use this reuse to describe cache behavior, we are concerned with finding two consecutive memory accesses to a memory line, with the earlier being the most recent previous access (MRPA). Next, we summarize our previous work [175], where we give a precise and insightful definition for the problem addressed in reuse analysis.

We assume the existence of an undefined access vector $\perp \notin \mathcal{M}$ that is always executed before any other access: $\forall\, \vec{a} \in \mathcal{M} : \perp \prec \vec{a}$. Let S be a set of integer vectors. The notation $\max_{\prec} S$ denotes the *lexicographic maximum* of S. By convention, $\max_{\prec} \emptyset = \perp$.

Let R and R' be two arbitrary references, which are not necessarily different. We define:

$$\mathcal{P}_{R,R'}(\vec{a}) = \{\vec{b} \in \mathcal{M}_{R'} \mid \vec{b} \prec \vec{a}, ml_R(\vec{a}) = ml_{R'}(\vec{b})\} \tag{5.1}$$

which consists of all accesses of R' preceding $\vec{a}$ and mapped to the same line as $\vec{a}$. Let

$$ipred_{R,R'} : \mathcal{M}_R \to \mathcal{M}_{R'} \cup \{\perp\}, \quad ipred_{R,R'}(\vec{a}) = \max_{\prec} \mathcal{P}_{R,R'}(\vec{a}) \tag{5.2}$$

Then $ipred_{R,R'}(\vec{a})$ is the most recent previous access, i.e., the one that *immediately precedes* $\vec{a}$ in $\mathcal{P}_{R,R'}(\vec{a})$.

By composing the $ipred_{R,R'}$ functions for a fixed R but varying R', we get:

$$ipred_R : \mathcal{M}_R \to \mathcal{M} \cup \{\perp\}, \quad ipred_R(\vec{a}) = \max_{\prec}\{ipred_{R,R'}(\vec{a}) \mid R' \in RefSet\} \tag{5.3}$$

Any reuse analysis is concerned with finding a good approximation of the function $ipred_R$ for every reference in the program.

[3] For FORTRAN77 codes it would be columns.

From the definition of the function $ipred_{R,R'}$, it is not difficult to recognize that we can build the function exactly using parametric integer programming (PIP) [43, 110]. However, it is unnecessary and can be expensive to solve this problem as a form of PIP for a large number of reference pairs. In numerical codes, the patterns of accesses are regular [114]. We can capture most of this regularity by considering only the accesses from uniformly generated references [50, 168, 160, 174] that are generalized from single (see Section 2.2.3) to multiple nests.

We have developed a practical algorithm, named *FindMRPA* [175] for computing $mrpa_{R,R'}$ when R and R' are uniformly generated. This algorithm is exact in commonly occurring cases, and according to our experiments, only in 2% of the cases requires the application of integer programming.

Shortest Reuse Vectors Given $\vec{a} \in \mathcal{M}_R$ and $\vec{b} \in \bigcup_{R' \in RefSet} \mathcal{P}_{R,R'}(\vec{a})$, $\vec{a} - \vec{b}$ represents a generalization of Wolf and Lam's reuse vectors from single nests to multiple nests [168]. Among all these reuse vectors of the access $\vec{a}$, the *shortest reuse vector* (SRV) in the entire program is:

$$\begin{aligned} SRV_R(\vec{a}) &= \vec{a} - ipred_R(\vec{a}) \\ SRV_R(\vec{a}) &\succ \vec{0} \end{aligned} \tag{5.4}$$

The concepts of MRPAs and shortest reuse vector are equivalent but the latter represents a roundabout way of providing the same information. $ipred_{R,R'}$ depends on the following factors: (a) the subscript expressions of R and R', (b) the base addresses of the arrays accessed by R and R', (c) the array sizes in all but the first dimension, (d) the line size L, (e) the shape of RIS_R and (f) the shape of $RIS_{R'}$. Wolf and Lam's reuse framework for single nests [168], while sufficient for their optimization purposes, does not provide the notion of shortest reuse vector required in behavior analysis. This is because they compute reuse vectors without considering the factors (b) – (f) above.

Next, we explain the implications of this theory when computing the shortest reuse vectors for our analysis.

5.4.2 Group Reuse Among Different RISs

Let R_p and R_c be two uniformly generated references. Let R_p be the producer $R_p[H\vec{I} + \vec{c_p}]$ nested inside the innermost loop labeled by $(\ell_1^p, \ell_2^p, \cdots, \ell_n^p)$ (see Section 5.2) and R_c be the consumer $R_c[H\vec{I} + \vec{c_c}]$ nested inside the innermost loop labeled by $(\ell_1^c, \ell_2^c, \cdots, \ell_n^c)$, where $\vec{I} = (I_1, I_2, \ldots, I_n)$.

Consider temporal reuse between R_p and R_c. Iterations $\vec{\imath}_1$ of R_p and $\vec{\imath}_2$ of R_c reference the same data whenever $H\vec{\imath}_1 + \vec{c_p} = H\vec{\imath}_2 + \vec{c_c}$, that is, when $M(\vec{\imath}_1 - \vec{\imath}_2) = \vec{c_p} - \vec{c_c}$.

Let $\vec{x} = (x_1, x_2, \ldots, x_n)$ be a solution to:

$$H\vec{x} \;=\; \vec{c_p} - \vec{c_c} \tag{5.5}$$

and

$$\vec{r_t} = (\ell_1^c - \ell_1^p, x_1, \ell_2^c - \ell_2^p, x_2, \ldots, \ell_n^c - \ell_n^p, x_n)$$

such that $\vec{r_t} \succeq 0$. Then $\vec{r_t}$ is a *temporal reuse vector* from R_p to R_c.

We assume a C data layout, where all arrays are row-major. Let $\vec{y} = (y_1, y_2, \ldots, y_n)$ be a solution to:

$$\begin{array}{l} H'\vec{y} = \vec{c_p'} - \vec{c_c'} \\ |H_1\vec{y}| < \mathtt{L} \end{array} \tag{5.6}$$

but not a solution to (5.5), where H_1 is the second row of H, and every primed term is obtained from its corresponding term in (5.5) with its second row or entry removed. Let

$$\vec{r_s} = (\ell_1^c - \ell_1^p, y_1, \ell_2^c - \ell_2^p, y_2, \ldots, \ell_n^c - \ell_n^p, y_n)$$

such that $\vec{r_s} \succeq 0$. Then $\vec{r_s}$ is a *spatial reuse vector* from R_p to R_c. Note that equations 5.5 and 5.6 are equivalent to those for perfectly nested loops (2.4 and 2.3) respectively.

If a memory line spans two adjacent rows of an array, we will add spatial reuse vectors to capture such reuse. The spatial reuse vectors of this second kind are added individually depending on the iteration space shapes and cache parameters used.

Let us derive reuse vectors for the first two references to b in Figure 5.3. Let R_p be $b[I_1 - 1][I_2]$ nested in the inner loop labeled by $\vec{L_p} = (1, 1)$ and R_c be $b[I_1][I_2]$ nested inside the inner loop labeled by $\vec{L_c} = (1, 2)$. The subscript expressions for both references are affine:

$$\begin{array}{rcl} H\vec{I} + \vec{c_p} & = & \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right] \left[\begin{smallmatrix} I_1 \\ I_2 \end{smallmatrix}\right] + \left[\begin{smallmatrix} -1 \\ 0 \end{smallmatrix}\right] \\ H\vec{I} + \vec{c_c} & = & \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right] \left[\begin{smallmatrix} I_1 \\ I_2 \end{smallmatrix}\right] + \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right] \end{array}$$

To find the spatial reuse vectors spanning a single column of b, we solve:

$$\begin{array}{rcl} \left[\begin{smallmatrix} 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right] & = & 0 \\ \left[\begin{smallmatrix} 0 & 1 \end{smallmatrix}\right] \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right] & < & \mathtt{L} \end{array}$$

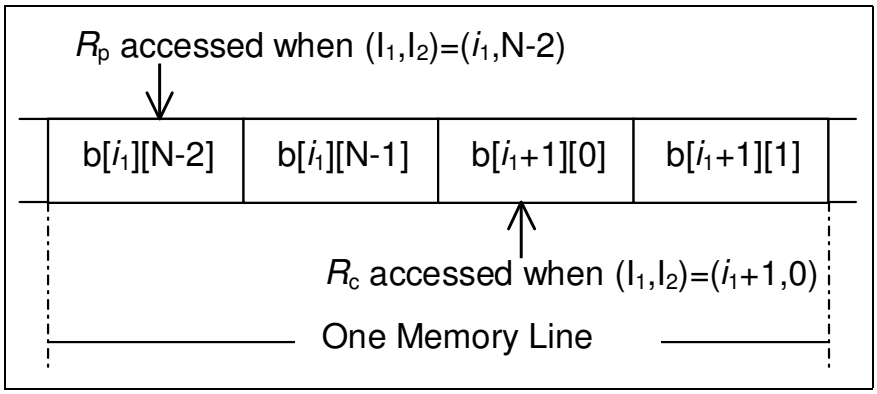


Fig. 5.5: Spatial reuse across array rows (L=4).

which is the instance of the equation (5.6) for this case. Thus, all these vectors have the form $(0, y_2)$. We would add the following spatial reuse vectors:

$$(0, 0, 1, -2), (0, 0, 1, -3), \ldots, (0, 0, 1, -(\mathtt{L} - 1))$$

Finally, we would need reuse vectors $(0, 1, 0, 1 - N)$ to capture the reuse for the elements at the end of one array row and the beginning of the next row. This is illustrated in Figure 5.5.

5.4.3 Discussion

If a reference is guarded by an if conditional, its *RIS* may not be the entire iteration space of the enclosing loop nest. This causes complications only in the derivation of group-temporal reuse vectors. The self-temporal and spatial reuse vectors for a reference are defined and derived without a need to refer to its *RIS*. As for the group-reuse vectors from R_p to R_c, a possible way to handle irregular *RISs* is to generate all potential ones conservatively. In the case of group-temporal reuse, there can be infinitely many reuse vectors from some facets of RIS_{R_p} to some facets of RIS_{R_c}.

Consider an extreme example illustrated in Figure 5.6. R_2 (the reusing reference) at every point (I_1, I_2) on the left boundary of its *RIS* may reuse R_1 (the reused reference) at the point $(30, I_2)$ on the right boundary of R_1's *RIS* along the symbolic group-reuse vector $(I_1 - 30, 0)$. If we ignore the two conditionals to analyze the reuse between the two references, the group-reuse vector $\vec{r} = (0, 0)$ will describe correctly the reuse from R_1 to R_2. When the miss equations for R_2 are formulated, the two conditionals must be taken into account. Then this reuse vector will be ignored

```
for(I1 = 0; I1 < 400; I1++) {
  for(I2 = 0; I2 < 140; I2++) {
    if(I1 <= 30)
      R1: a[I2])
    if(I1 + I2 >= 200)
      R2: a[I2]
  }
}
```

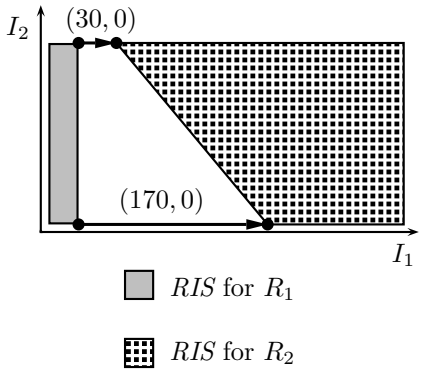


Fig. 5.6: Derivation of group-reuse vectors.

since the two *RISs* do not overlap. As a result, the number of cache misses for R_2 on the left boundary of its *RIS* may be over-estimated. For practical applications, such an over-estimation is negligible because (a) the over-estimation occurs only on a facet of a *RIS* (e.g., the left boundary of R_2's *RIS*) and (b) the underlying reference may reuse on the facet via other reuse vectors. In the example, R_2 may reuse from itself along the self-spatial reuse vector $(1, -1)$. Thus, only a small fraction of these boundary points are miss-predicted.

This situation arises at most in 2% of the cases [175]. Our extensive validation confirms that an overestimation of cache misses caused by ignoring this reuse is negligible since (a) we overestimate only on some facets of RIS_{R_c} and (b) R_c may reuse on the facets by other reuse vectors (usually self-reuse vectors). However, when we need an exact description of the reuse, we apply integer programming techniques [175] to guarantee that we obtain the shortest reuse vector.

5.5 Cache Behavior Modeling

Following the same classification as the CMEs, we divide our miss equations into two groups : *compulsory or cold (miss) equations* and *replacement (miss) equations*. *Compulsory* misses represent the first time a memory line is touched while *replacement* misses are those accesses that result in misses because the cache lines that would have been reused were evicted from the cache before they get reused. Note that replacement equations represent both capacity and interference misses.

In this section, we present the miss equations as a specification of the cache misses in a loop nest. We then discuss two algorithms for finding cache misses from these equations. In particular, our replacement miss equations are formulated and solved differently from those in the CMEs [52] since the involved *RISs* can be

different. We also describe an algorithm for computing efficiently the volume of a *RIS* for sampling purposes.

5.5.1 Forming Equations

Let $\vec{r}$ be a reuse vector from the producer reference R_p to the consumer reference R_c. We want to find out if R_c at iteration $\vec{i}$ can reuse the cache line accessed by R_p at $\vec{i}-\vec{r}$. Let R_i be an *intervening* reference such that the access of R_i at some iteration point $\vec{j}$ between $\vec{i}-\vec{r}$ and $\vec{i}$ [4] may be mapped to the same cache set as the access of R_p at $\vec{i}-\vec{r}$. If that happens, a *set contention* occurs between the access of R_p at $\vec{i}-\vec{r}$ and the access of R_i at $\vec{j}$. In a `k`-way set associative cache with LRU replacement policy, it takes `k` distinct set contentions to evict the cache line touched by the access of R_p at $\vec{i}-\vec{r}$.

We give below the miss equations that can be further analyzed to determine if the access of R_c at $\vec{i}$ is a miss or hit, assuming the single reuse vector $\vec{r}$ from R_p to R_c and the single intervening reference R_i.

Compulsory Equations

The compulsory equations for R_c along $\vec{r}$ represent the iteration points where the memory lines are brought to the cache for the first time:

$$\begin{gathered}\vec{i} \in RIS_{R_c} \\ \text{and} \\ \big(\vec{i}-\vec{r} \notin RIS_{R_p} \\ \text{or} \\ ml_{R_c}(\vec{i}) \neq ml_{R_p}(\vec{i}-\vec{r})\big)\end{gathered}$$

If $\vec{r}$ is temporal, the inequality is false and thus redundant.

Replacement Equations

The *replacement equations* for R_c along $\vec{r}$ are to investigate if R_c at iteration $\vec{i}$ can reuse the cache line that R_p accessed at iteration $\vec{i}-\vec{r}$ subject to the set contentions caused by the memory accesses from R_i. These equations only describe cache set contentions, and we rely on the solver to check whether they result in a miss or not. R_i may cause a cache set contention at all intervening points executed between $\vec{i}-\vec{r}$

[4] In lexicographic order.

INPUT
P = a program
S = sample
OUTPUT
$MissAnalyzer(P,S) = \frac{\sum_R |RIS_R| \times Miss\_Ratio(R)}{\sum_R |RIS_R|}$
ALGORITHM
for each reference R
  $H_R = RM_R = \emptyset$ // *Hits/Replacement misses for R*
  $CM_R = \mathcal{S}(R)$ // *Compulsory misses for R*
  for each reuse vector in $\prec$ order
    for each $\vec{\imath} \in CM_R$
      if $\vec{\imath}$ is "compulsory" miss
        if $\vec{\imath}$ is a "replacement" hit
          $H_R = H_R \cup \{\vec{\imath}\}$; $CM_R = CM_R - \vec{\imath}$
        else
          $RM_R = RM_R \cup \{\vec{\imath}\}$; $CM_R = CM_R - \vec{\imath}$
  $Miss\_Ratio(R) = \frac{|CM_R|+|RM_R|}{|\mathcal{S}(R)|}$

Fig. 5.7: An algorithm for estimating cache misses.

and $\vec{\imath}$:

$$\begin{gathered} ml_{R_c}(\vec{\imath}) = ml_{R_p}(\vec{\imath} - \vec{r}) \\ \vec{\imath} \in RIS_{R_c} \\ \vec{\imath} - \vec{r} \in RIS_{R_p} \\ cs_{R_c}(\vec{\imath}) = cs_{R_i}(\vec{\jmath}) \\ \vec{\jmath} \in J_{R_i} \end{gathered}$$

where J_{R_i} denotes the set of all these intervening iteration points, called the *interference set* for R_c along $\vec{r}$, and is specified precisely by:

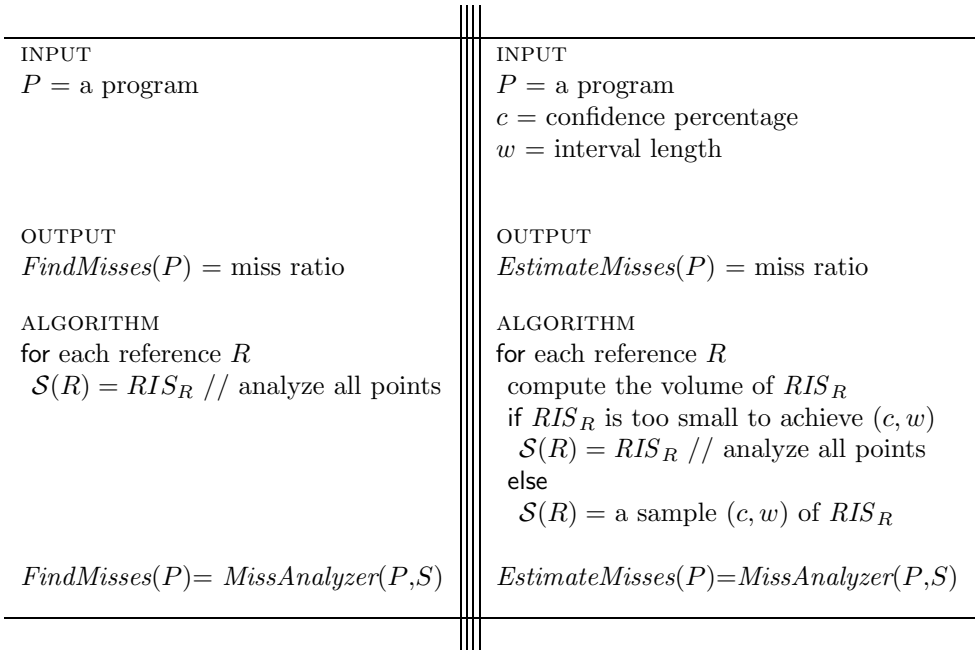
$$J_{R_i} = \{\vec{\jmath} \in RIS_{R_i} \mid \vec{\jmath} \in \ll \vec{\imath} - \vec{r}, \vec{\imath} \gg\}$$

where '$\ll$' is '[' if R_i is lexically after R_p and '(' otherwise and '$\gg$' is ']' if R_i is lexically before R_c and ')' otherwise. A reference is neither lexically before nor lexically after itself.

5.5.2 FindMisses and EstimateMisses

Figure 5.7 gives an algorithm for obtaining the cache misses from a whole program, consisting of multiple references and reuse vectors in multiple loop nests. It analyzes

each reference by going through its reuse vectors in lexicographical order $\prec$ (see Section 2.3.3). If an iteration point is a solution to the cold equations along the current reuse vector $\vec{r}$, its behavior is indeterminate and will be examined further using the other reuse vectors later in the list. Otherwise, the iteration point is classified either as a hit or a miss using the replacement equations along $\vec{r}$. After all reuse vectors have been checked, the remaining indeterminate iteration points are cold misses.



(a) FindMisses (b) EstimateMisses

Fig. 5.8: Algorithms to compute cache misses with different trade-offs between accuracy and execution time.

We present our two practical algorithms in Figure 5.8. *FindMisses* analyzes all iteration points in a *RIS* and is practical only for programs of small sizes [52]. *EstimateMisses* analyzes a sample of a *RIS* and is capable of analyzing programs significantly more efficiently with a controlled degree of accuracy. *EstimateMisses* requires the user to enter values to the two parameters: the *confidence percentage* c and the *confidence width* w, *where* $0\% < c \leqslant 100\%$ *and* $0 < w < 1$ (see Section 3.4). The two input values determine the size of the sample taken from RIS_R and also impose a lower bound on $|RIS_R|$. If a *RIS* is too small to achieve (c, w), we analyze

all points in RIS_R. The meanings of c and w are such that if we run *EstimateMisses* many times, the real miss ratio for each R obtained in c of these runs will lie in the interval $[Miss\_Ratio(R) - w/2, Miss\_Ratio(R) + w/2]$. However, this interpretation does not apply to the miss ratio for the entire loop nest. In all our experiments, real and estimated miss ratios are close (see Section 5.8).

Thus, the statistical sampling technique used requires the size of every *RIS* to be calculated. Our algorithm for computing the volume of a *RIS* is described as follows. If the IF conditions guarding a reference form a union of convex polyhedra, then the corresponding *RIS* is a union of convex polyhedra because the iteration space is convex. The number of points contained in such a *RIS* is calculated by slicing the *RIS* recursively into regions of lower and lower dimensions until eventually every region is either empty or a (one-dimensional) union of line segments so that the points in the region can be counted easily. This algorithm, while exponential in terms of the dimension of the iteration space, is very efficient for practical programs with simple loop bounds and affine conditionals. Other methods for computing the volume of a convex polytope also exist [34, 127].

If a reference R is guarded by some non-affine conditionals, then RIS_R can be arbitrarily complex. There is not any general method for computing the volume of RIS_R. In our implementation, we compute the volume of such a *RIS* by proceeding as before with all non-affine conditionals ignored and then counting only those points that satisfy all non-affine conditionals. This simple extension has not been used in our experiments since we have not found any data-independent conditional that is not affine in any programs analyzed.

5.6 Locking Caches

When we extend our analysis to caches with locking features, we have to treat in a different manner references within a locked region compared to those within an unlocked region. Regarding the reuse vectors, it is enough to ignore reuse vectors whose producer reference is in a locked region. As those accesses within a locked region do not bring data to the cache, they cannot affect the result of future accesses. Furthermore, they cannot affect the decision of the LRU replacement policy since they do not create a recent use of a memory line. When analyzing potential cache set contentions, references within a locked region should be ignored. Again, since they do not either bring data or modify the LRU state, they will not generate any set contention.

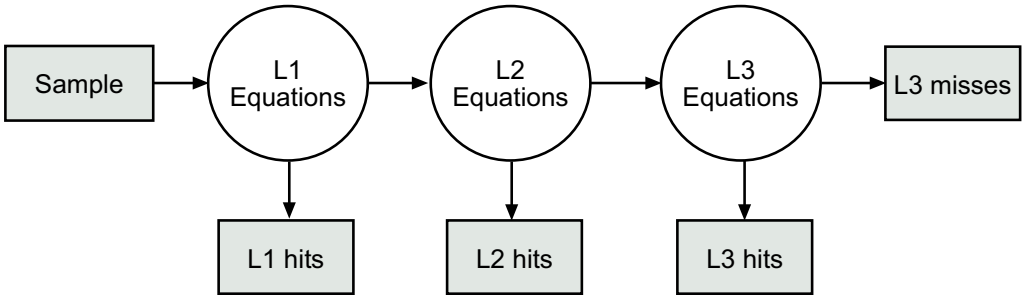


Fig. 5.9: Our approach for analyzing multi-level caches.

5.7 Multi-level Caches

The increasing performance mismatch between memory and processor speeds has required an increased number of cache levels. For instance, Itanium [135] has three levels of cache. Thus, an accurate model for predicting cache behavior must give a quantitative measurement of cache misses for all levels of cache. Therefore, we have extended the analysis to describe cache behavior for modern architectures.

Given a memory reference, the equations are to investigate whether the reuse described by its reuse vectors is realized or not. Thus, for these architectures, we have to analyze differently memory references depending on the cache level they are accessing. For that purpose, a set of equations that describes precisely the relationship among the iteration space, array sizes and cache parameters is set up for each of the cache levels.

Figure 5.9 shows our approach for a 3-level cache memory hierarchy. When analyzing potential cache set contentions, only memory accesses that miss in lower cache levels are considered. Thus, we can see the equations for each level as filters, where only those memory accesses that miss are analyzed in further levels.

Our approach only considers data caches. If the higher levels of cache are unified, we would ignore the accesses due to misses to the instruction cache. In the general case, the number of data accesses is much larger than the number of accesses to higher levels of cache due to instructions misses. Thus, only a small number of data accesses will be miss-predicted.

Kernel	k	(8KB,64B)	(16KB,32B)	(32KB,32B)
HYDRO_K	1	0.19	0.26	0.26
	2	0.18	0.26	0.25
	4	0.18	0.25	0.24
MGRID_K	1	0.35	0.30	0.20
	2	0.37	0.17	0.14
	4	0.30	0.14	0.14
MMT	1	0.19	0.24	0.43
	2	0.14	0.16	0.28
	4	0.09	0.16	0.21
MM	1	0.32	0.37	0.30
	2	0.30	0.29	0.27
	4	0.30	0.29	0.22
LWSI_K	1	0.99	0.63	0.57
	2	0.76	0.62	0.51
	4	0.61	0.65	0.53

Table 5.2: Average absolute errors when compared against simulation for the experiments from Figures 5.10, 5.11 and 5.12.

5.8 Validation

We now present results for different kernels, isolating if conditionals and multiple loop nests in different tests. Then, we put everything together and we show the accuracy and feasibility of our approach for analyzing whole programs. For results when the cache is locked at run time, see Section 7.6.

We analyze the L1 caches of the four high-performance processors considered in this thesis (see Table 4.1 in Section 4.3). All execution times are obtained on a 933MHz Pentium III PC with 512MB of RAM. In all experiments, we ran *Estimate-Misses* with $c = 95\%$ and $w = 0.05$ as the input. The time elapsed on analyzing a program includes the costs of generating and solving the equations, and does not include the parsing time.

5.8.1 Loop Kernels

We first discuss the experimental results for the five kernels given in Table 4.3 and that can be seen in Appendix A. Figure 5.10 compares the predicted miss ratios against that from simulation for a fixed (8KB, 64B) cache configuration but with three different k choices. In each graph, the miss ratios are plotted for one single problem size parameter (with the others, if any, fixed). Figures 5.11 and 5.12 show

Kernel	E.M.(secs)			Sim.(secs)		
	Min.	Max.	Mean	Min.	Max.	Mean
HYDRO_K	0.03	0.27	0.19	0.04	14.49	5.10
MGRID_K	0.17	0.19	0.18	0.50	35.00	11.82
MMT	0.71	15.54	4.71	1054.64	1938.65	1295.66
MM	0.07	0.89	0.33	0.13	1126.93	171.75
LWSI_K	0.06	0.28	0.17	0.21	18.33	8.20

Table 5.3: Execution times for (32KB,32B,2) illustrated in Figure 5.12.

Program	#lines	#subrs	#calls	#refs
TOMCATV	190	1	0	79
SWIM	429	6	6	52
APPLU	3868	16	27	2565

Table 5.4: Three whole programs.

the same plot for (16KB, 32B) and (32KB,32B) cache configurations. Table 5.2 gives the average absolute errors. Table 5.3 shows the times taken to evaluate all the kernels for a single cache configuration. In all experiments, the predicted miss ratios are close to the simulated ones and are obtained in times that are at least two orders of magnitude faster than simulation. In Figures 5.10, 5.11 and 5.12, the seemingly big differences between the predicted and simulated miss ratios for MGRID_K are due to the short ranges used for the miss ratios. We have used a smaller number of samples for MMT because the tile size parameters KN and JN are required to divide N.

5.8.2 Whole Programs

We evaluate *EstimateMisses* against a simulator using three programs from SPECfp95 detailed in Table 5.4. In each case, we have succeeded in abstractly inlining all the calls and obtained one loop nest for the program. In addition, all actual parameters are propagateable, meaning that the references to every actual can be potentially exploited across calls. Each program is analyzed using the reference input data. Thus, the variables in all READ statements are initialized from the reference data and then treated as compile-time constants.

The three programs are further discussed below.

TOMCATV from SPECfp95. This example is used to demonstrate the capability of

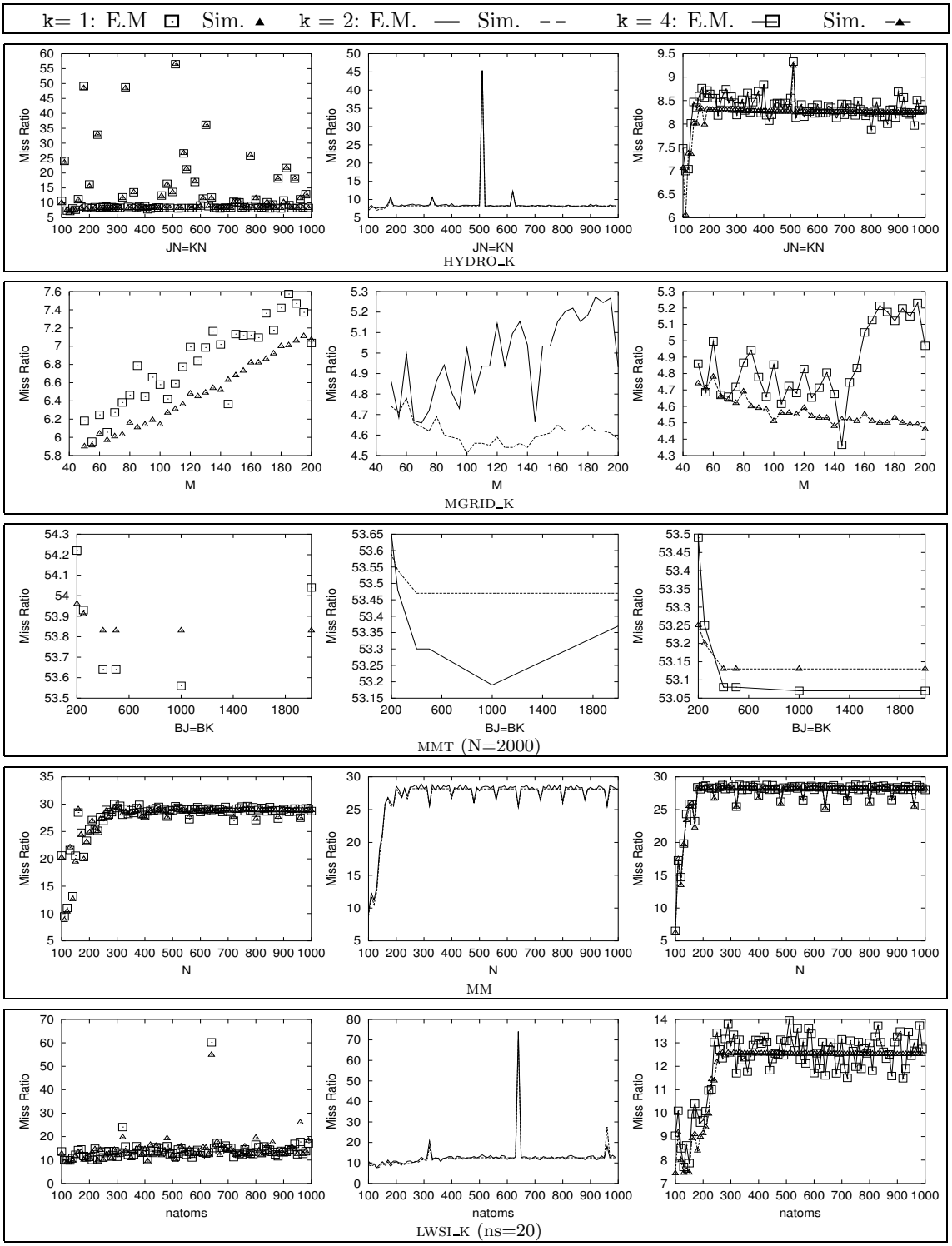


Fig. 5.10: Predicted and simulated miss ratios for (C,L)=(8KB, 64B) with three different k.

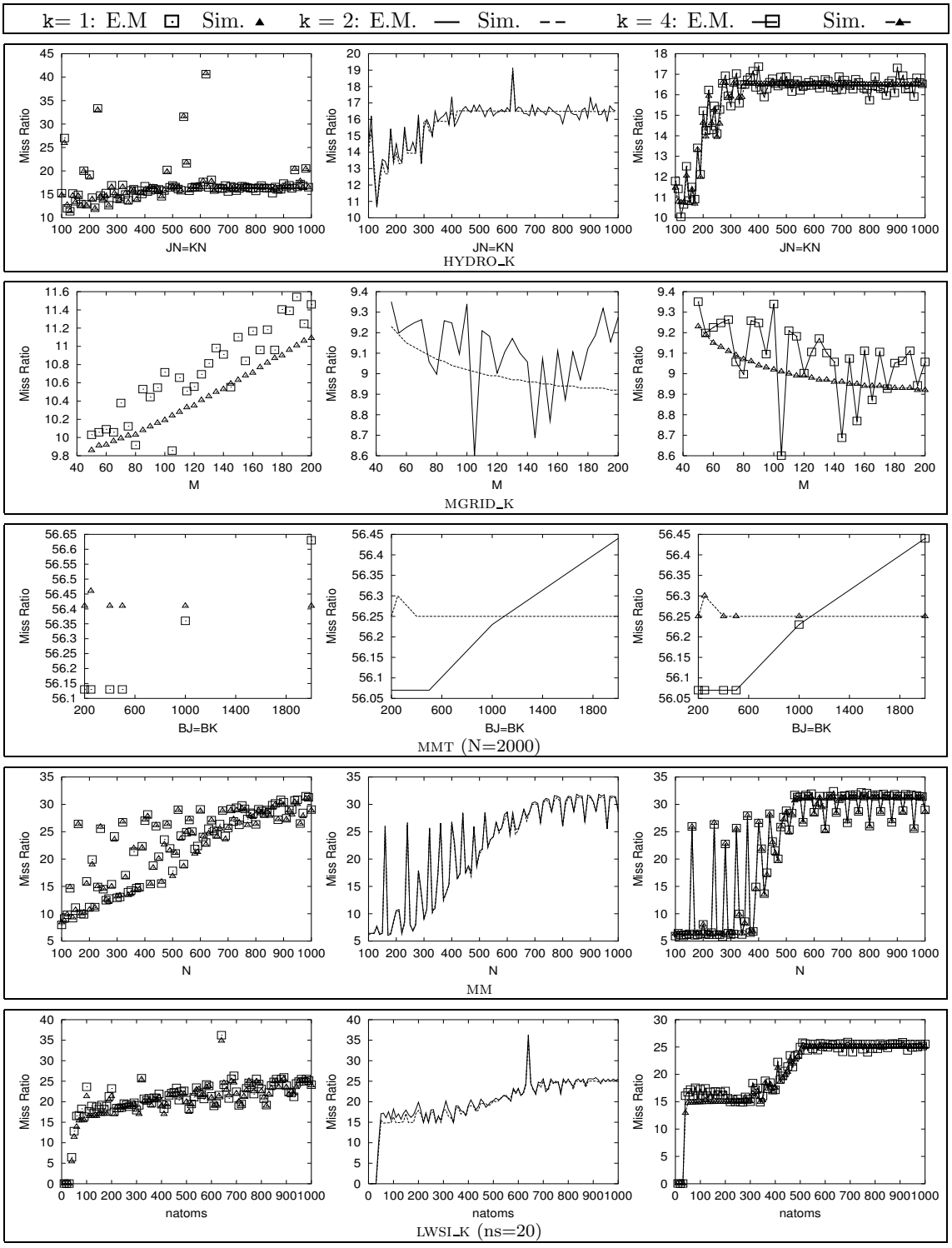


Fig. 5.11: Predicted and simulated miss ratios for (C,L)=(16KB, 32B) with three different k.

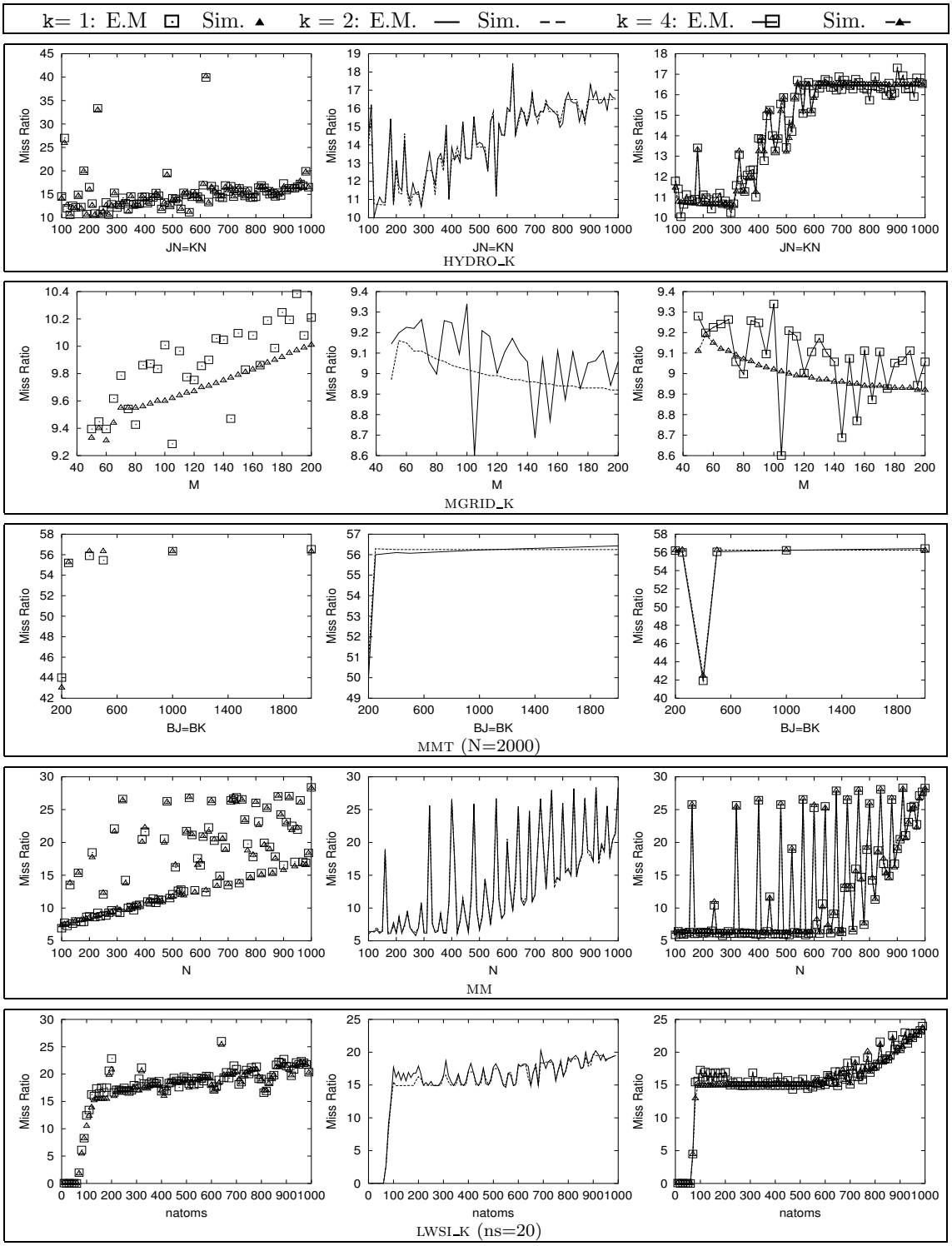
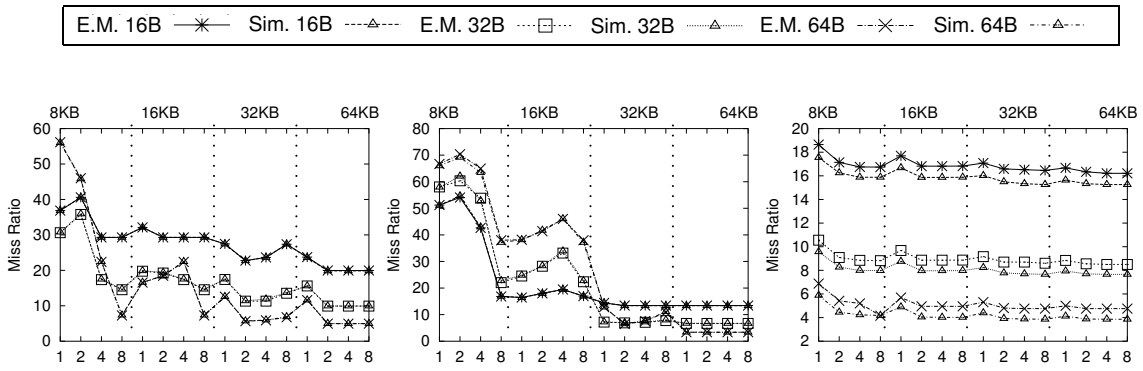


Fig. 5.12: Predicted and simulated miss ratios for (C,L)=(32KB, 32B) with three different k.



(a) TOMCATV (b) SWIM (c) APPLU

Fig. 5.13: Predicted and simulated miss ratios for $\#\mathtt{C} \times \#\mathtt{L} \times \#\mathtt{k} = 96$ cache configurations.

our method in analyzing real codes. The number of iterations of the outermost loop is data-dependent. For the reference input data used, the outermost loop runs for 750 iterations. The only data-dependent if conditional in the program is always false. The memory accesses contained in this conditional are included in our analysis.

SWIM from SPECfp95. This example demonstrates that we can analyze codes consisting of call statements. All calls are parameterless. The outermost loop is an if-goto construct, which has been converted into a for statement.

APPLU from SPECfp95. This shows that our method is capable of analyzing this scale of programs efficiently with a good degree of accuracy. All actual parameters are propagateable. In subroutine SSOR, there are some data-dependent constructs. All but one are guarded by an if branch that is false at compile time and are thus ignored. The remaining data-dependent if construct is a WRITE statement for a register-allocated scalar. The memory accesses in this if conditional are included in our analysis.

Figure 5.13 compares the predicted miss ratios against that from simulation for $\#\mathtt{C} \times \#\mathtt{L} \times \#\mathtt{k} = 4 \times 3 \times 8 = 96$ cache configurations. The total analysis times required by TOMCATV, SWIM and APPLU for all the configurations lumped together are about 14 secs, 2.8 mins and 3.2 hours, respectively. These numbers are in sharp contrast with the respective total simulation times consumed: 55 hours, 110 hours and 230 hours. In addition to being efficient, our model is accurate (in

Program	k	Miss Ratio			Exe.T (secs)	
		E.M.	Sim.	Abs.Err.	E.M.	Sim.
TOMCATV	1	56.16	56.00	0.16	0.31	4780.48
	2	46.01	45.83	0.18	0.49	4897.58
	4	22.38	22.46	0.08	0.60	4878.56
	8	7.27	7.36	0.09	0.63	4769.87
SWIM	1	66.71	65.98	0.73	2.88	8831.84
	2	70.15	69.18	0.97	3.55	9030.47
	4	63.95	63.46	0.49	3.79	9353.51
	8	37.99	37.15	0.84	5.93	9401.60
APPLU	1	6.88	5.89	0.99	129.3	16784.15
	2	5.42	4.45	0.97	129.7	17646.31
	4	5.20	4.23	0.97	129.8	18848.32
	8	4.16	4.18	0.92	129.6	21104.30

Table 5.5: Absolute errors and execution times compared against simulation for (8KB, 64B) from Figure 5.13.

terms of its prediction errors) and consistent (in terms of the trend exhibited by the errors). To understand these points further, Table 5.5 gives the absolute errors and the times to evaluate the three programs for (8KB, 64B). For the programs of the scale such as APPLU, *EstimateMisses* yields close to actual miss ratio in about 130 seconds for each associativity while the cache simulation runs for about 5 hours. This translates into a three orders of magnitude speedup over the cache simulator used. In terms of memory requirement, our model requires about 1.8MB, 7.1MB and 60.3MB to analyze TOMCATV, SWIM and APPLU for the cache configuration (8KB, 64B), respectively.

5.9 Related Work

We review in detail the five *general-purpose analytical* cache models developed primarily for guiding data cache optimizations [29, 48, 52, 63, 143]. These models are all restricted to data-independent language constructs, and consequently, can obtain predictions of miss ratios at compile time without relying on address traces.

An analytical model consists of three components: *reuse analysis*, *cache miss specification* and *cache miss computation*. In some cases, some or all of the three are combined. Reuse analysis applies a reuse metric to obtain quantitative measurement of data reuse in the program. Based on this analysis, some mathematical formulas for specifying cache misses are set up. In the case of numerical programs, these formulas typically describe the relationships among loop variables, array sizes, base addresses and cache parameters. Finally, the cache miss information is generated

from the specification by some means.

Temam *et al* [143] estimate the cache misses of individual perfect nests with rectangular iteration spaces for direct-mapped caches. They consider a subset of uniformly generated references so that all temporal reuse vectors are essentially basis vectors. To find the misses for a particular iteration of a reuse-carrying loop (corresponding to a basis reuse vector), they compute the footprint (i.e., the set of cache lines) accessed by each reference and solve their formulas expressed in terms of these footprints. Despite that the footprints are generally approximated, they obtain good estimates for several kernel examples. Recently, Harper *et al* [63] give an extension to set-associative caches but still for the same class of nests. They obtain good estimates for the four example kernels: MM as given in Appendix A, a 2-D SOR nest, a 2-D Jacobi nest and a blocked matrix multiplication.

Ghosh *et al* [52] introduce the well-known CMEs, a set of equalities and inequalities, to specify the cache misses of a single perfect nest with straight-line assignments for set-associative caches. Their reuse metric is Wolf and Lam's reuse vectors, which they obtain from uniformly generated references approximately by relying on Wolf and Lam's reuse framework and some ad hoc techniques. They show that the CMEs can help an optimizing compiler choose tile and pad sizes without requiring the CMEs to be solved explicitly.

Fraguela *et al* [48] rely on a probabilistic model to provide a fast estimation of cache misses for set-associative caches. They use the so-called area vectors as a reuse metric to represent probabilistically the amount of reuse along those directions and solve their recursive cache miss equations for cache misses. When analyzing imperfect nests, they exploit only the reuse between references contained in a common nest. These references differ by constants in their matching dimensions, forming a subset of uniformly generated references considered in the CMEs. They validate the accuracy of their model using three kernel examples. The two perfect nests can also be analyzed by the CMEs, whereas the third one is a 3-D blocked imperfect nest for computing AB^T (named MMT and given in Appendix A). Table 5.6 compares their method with ours for this particular kernel. Our *EstimateMisses* produces better results in all cases. The two largest relative errors occur since the total number of misses is small.

Chatterjee *et al* [29] present a cost model for exactly analyzing the cache behavior of loop nests for set-associative caches. They use Presburger formulas (as a reuse metric) to specify a program's cache misses, the Omega Calculator [126] to simplify the formulas, PolyLib [163] to obtain an indiscriminating union of polytopes, and finally, Ehrhart polynomials to count the integer points (i.e., misses) in each polytope [34]. They can formulate Presburger formulas for a looping structure

N	BJ	BK	C	L	k	Δ_P	Δ_E
200	100	100	16	8	2	6.23	0.1
200	100	100	256	16	2	2.73	0.5
200	200	100	32	8	1	6.88	0.06
200	200	100	128	8	2	2.86	0.05
200	200	100	128	32	2	44.25	16
200	50	200	16	4	1	4.62	0.05
200	100	200	32	8	2	12.51	0.1
200	100	200	64	16	1	3.31	0.4
400	100	100	16	8	2	4.48	0.03
400	100	100	256	16	2	4.26	0.5
400	200	100	32	8	1	2.65	0.4
400	200	100	128	8	2	5.82	0.05
400	200	100	128	32	2	44.68	16
400	50	200	16	4	1	2.02	0.05
400	100	200	32	8	2	5.55	0.06
400	100	200	64	16	1	7.12	0.3

Table 5.6: Comparison with Fraguela et al's probabilistic method using MMT. Δ_p denotes the relative error between the estimated and real miss ratios for the probabilistic method and Δ_E for our *EstimateMisses*.

consisting of imperfect nests, IF statements, references with arbitrary affine accesses and non-linear data layouts. However, their current implementation as a SUIF [111] pass "does not yet have enough performance to be practical." [29, p. 295]. Two examples are discussed: matrix multiplication — it takes 1 to 10 seconds on a 300MHz Sparc Ultra 60 to analyze the 21×21 instance; matrix-vector product — they have derived the Presburger formulas for $N = 100$ but did not solve them.

Note that all these cache modeling techniques may make use of our *abstract inlining*, since it works as an enabler. While we inline in order to analyze cache misses exactly, existing techniques such as [3, 27, 137] inline primarily to optimize program performance.

5.10 Chapter Summary

We have described a compile-time framework for obtaining quantitative measurement of data reuse in whole programs. This framework consists of abstract inlining, generalized iteration vectors and generalized reuse vectors. The abstract inlining enables a program with calls to be analyzed statically; it may be useful to other cache modeling techniques. The generalized iteration vectors define the execution

order among the iterations from a common nest or distinct nests, allowing the whole program to be represented in the polyhedral model. A generalized reuse vector represents precisely the direction and distance of reuse across the entire program.

To the best of our knowledge, this is the first work that demonstrates the feasibility of analyzing statically the cache behavior of whole programs with regular and compile-time predictable memory accesses. We have described, implemented and validated an analytical model for programs where the bulk of computations are expressed in loop nests operating on arrays and scalars. Our experimental results using kernels and complete programs indicate accurate cache miss estimates in substantially shorter amount of time than simulation. Our model can obtain predictions of miss ratios for program regions ranging from a single reference to the entire program.

There are many benefits to static analytical modeling. In the rest of this thesis, we show how we use our analysis to optimize programs' performance. In addition, we explain how we use the mathematical formulas developed for characterizing cache misses to tighten the bound of the WCET of a program.

6

Data Cache Optimization

CHAPTER 6

DATA CACHE OPTIMIZATION

The performance of the memory hierarchy can be improved by means of data and loop transformations. Tiling is a loop transformation that aims at reducing capacity misses by shortening the reuse distance. Padding is a data layout transformation targeted to reduce conflict misses.

In this chapter we present an accurate cost model that describes misses across different hierarchy levels and considers the effects of other hardware components such as branch predictors. The cost model drives the application of tiling and padding transformations. We combine the cost model with a genetic algorithm to compute tile and pad factors that enhance the program performance.

Our results show that this scheme is useful to optimize programs' performance. When compared to previous approaches, we observe that with a reasonable compile-time overhead, our approach gives significant performance improvements for all studied kernels on all architectures.

6.1 Compiler Cache Transformations

A fast and accurate assessment of a program's cache behavior at compile time is needed to make an appropriate choice of transformation parameters. Unfortunately, cache memory behavior is very hard to predict. Thus, current approaches are based on simple models (heuristics) for estimating locality [25, 35, 93, 129, 131]. However, modern architectures have a very complex internal organization, with different levels of cache, branch predictors, etc. Such models provide very rough performance estimates, and in practice, are too simplistic to statically select the best optimizations.

Tiling has been shown to be useful for many algorithms in linear algebra. By restructuring the loop and changing the order in which memory references are executed, it reuses data in the faster levels of the hierarchy; thus it reduces the average latency. Nevertheless, finding the optimal tile sizes is a very complex task. The solution space is huge, and exploring all possible solutions is infeasible.

```
double a[N][N];
double b[N][N];
double c[N][N];

for(ii=0;i<N;ii+=T1)
  for(jj=0;jj<N;jj+=T2)
    for(k=0;k<N;k++)
      for(i=ii;i< min(ii+T1-1, N);i++)
        for(j=jj;j< min(jj+T2-1, N);j++)
          a[i][j]=a[i][j]+b[i][k]*c[k][j]
```

(a) Tiled Matrix Multiply.

```
double a[N+P_Dim00][N+P_Dim01]
double b[N+P_Dim10][N+P_Dim11]
double c[N+P_Dim20][N+P_Dim21]

for(ii=0;i<N;ii+=T1)
  for(jj=0;jj<N;jj+=T2)
    for(k=0;k<N;k++)
      for(i=ii;i< min(ii+T1-1, N);i++)
        for(j=jj;j< min(jj+T2-1, N);j++)
          a[i][j]=a[i][j]+b[i][k]*c[k][j]
```

(b) Tiled and Padded Matrix Multiply.

Fig. 6.1: Matrix multiply algorithm after applying tiling and padding.

Padding has a significant potential to remove conflict misses. In fact, it can remove most conflict misses by changing the addresses of conflicting data, and some compulsory misses by aligning data with cache lines. However, choosing the optimal data layout is an NP-complete problem [123]. A number of algorithms have been proposed which are based on simple cost models that only consider the first level cache [129].

In comparison, the proposed method makes use of an accurate cost model combined with a genetic algorithm. In particular, we improve the order of memory accesses via tiling, whereas conflict misses that tiling cannot eliminate are removed via padding. Moreover, it chooses the best tile and pad factors at the same time.

Next, we review the transformations implemented in our experimental compiler.

6.1.1 Tiling Overview

Loop tiling combines strip-mining [9] with loop interchange [170] for increasing the effectiveness of memory hierarchy. Recall Figure 2.4, where we show the code for the matrix multiplication (of `NxN` arrays) kernel. Loop tiling basically consists of two steps [168]. The first one consists of restructuring the code to enable tiling those loops that carry reuse. The second one is to select the tile factors that maximize locality. It is the latter step that is sensitive to the characteristics of the cache memory considered. Due to hardware constraints, caches have limited associativity, which may cause cache lines to be flushed out of the cache before they are reused despite sufficient capacity in the overall cache.

We present the tiled version, with tile sizes `T1` and `T2`, in Figure 6.1(a).

Implementing Tiling

The iteration space obtained after tiling n dimensions can be expressed as the union of 2^n convex regions. We illustrate this situation in Figure 6.2. Figure 6.2(a) shows how a 1-dimensional iteration space becomes a two-convex region iteration space (see Figure 6.2(b)) after tiling (`T=3`). The shaded regions correspond to the different convex regions before and after tiling.

A naïve way to overcome this problem is to use only one convex region that approximates the actual non-convex region. This convex region can be the smallest parallelepiped that includes all other convex regions (see Figure 6.2(c)) or alternatively, the region which does not include the last iteration of every tiled loop when the tile size does not divide the upper bound (see Figure 6.2(d)). Nevertheless, neither option is accurate. The first option includes points outside the iteration space, whereas the second option excludes points belonging to the iteration space.

In our aim of having an accurate model, we decided to implement the exact solution. By transforming the min expression into an if conditional, we formulate the equations making use of the *RIS* concept. This transformation is only for analysis purpose. Hence, we do not incorporate any overhead due to the extra conditional.

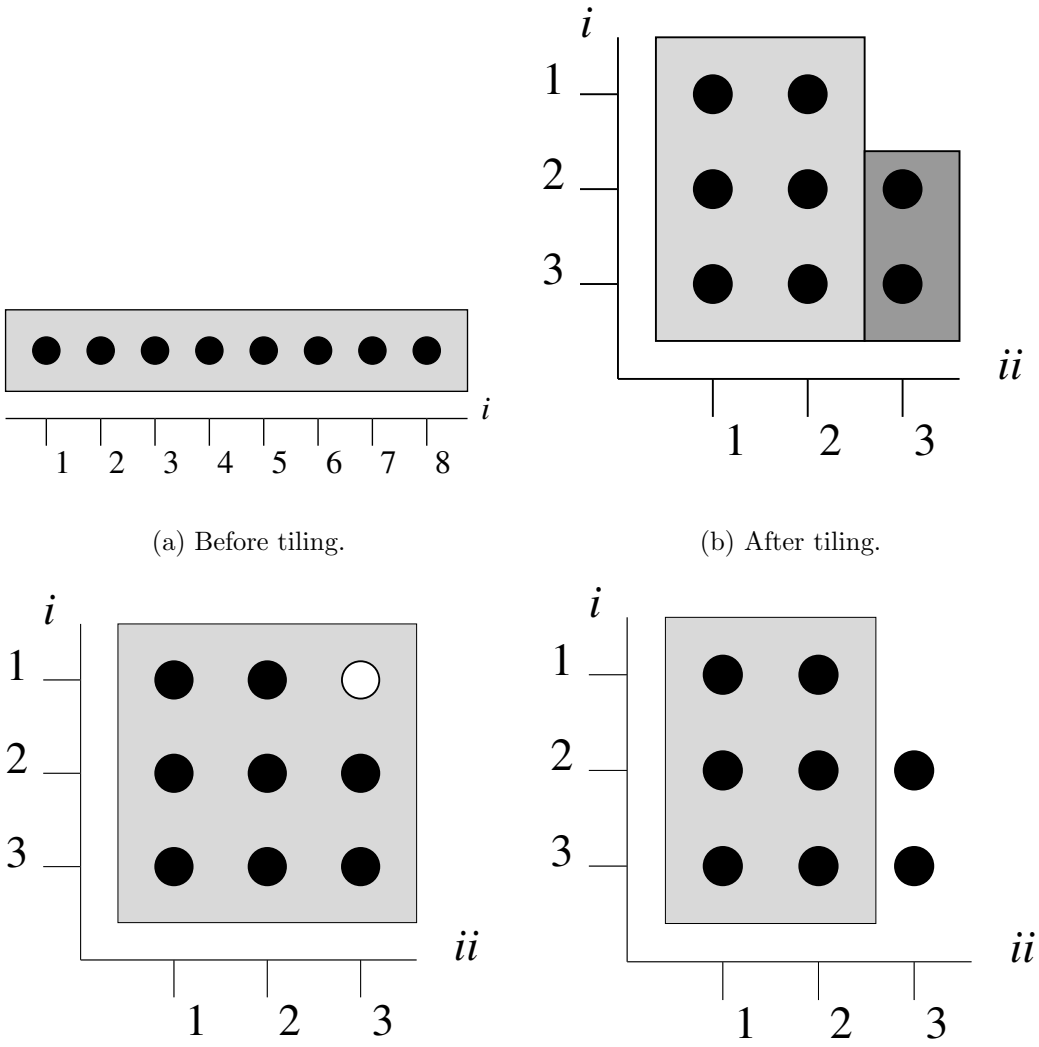
Example 10 *Consider the loop "***for***(j=jj;j< min(jj+*`T2`*-1, N);j++)" in Figure 6.1(a). We would consider the RIS equivalent to the following code:*

```
for(j=jj;j< jj+T2-1;j++)
{ if (j<N)
...
```

6.1.2 Padding Overview

Unlike loop tiling, padding modifies the data layout to eliminate conflict misses. It changes the data layout in two different ways. Inter-padding modifies the base addresses of the arrays, whereas intra-padding changes the size of array dimensions.

As usual, we refer to the L1 (primary) cache size as `C`. mem_i is the original base address of variable number i (Var_i) and P_Base_i stands for the inter-variable padding between Var_i and Var_{i-1}. dim_{ij} stands for the size of the dimension j of Var_i (D_i is the number of dimensions) and S_i is its size. P_Dim_{ij} is the intra-variable padding applied to dim_{ij}, and P_S_i is the size of Var_i after padding (see Figure 6.3). We define Δ_i as $P\_S_i - S_i$. Notice that all variables are natural numbers.



(a) Before tiling.

(b) After tiling.

(c) Using a bigger region.

(d) Using a smaller region.

Fig. 6.2: Example of tiled iteration space.

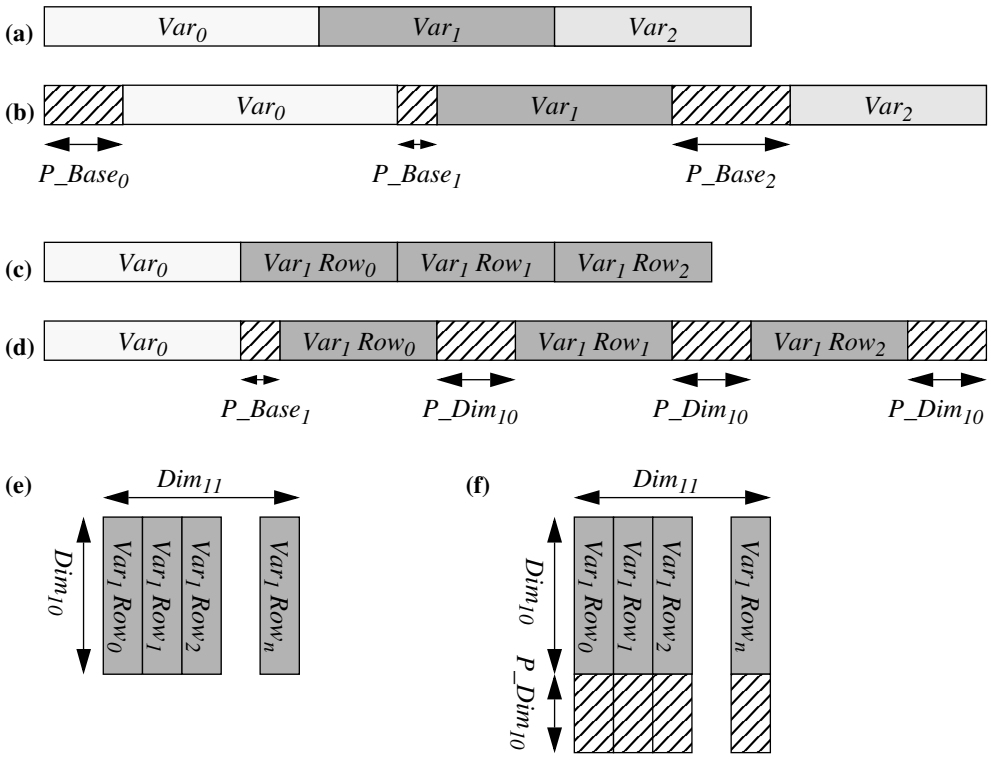


Fig. 6.3: Data layout: (a) before inter-variable padding, (b) after inter-variable padding (c) before padding, (d) after padding, (e) 2-D array, (f) 2-D array after intra-variable padding

Inter-variable Padding

When inter-variable padding is applied only the base addresses of the variables are changed. Thus, padding is performed in a simple way. Memory variable base addresses are initially defined using the values given by the compiler. Then, we define for each memory variable Var_i, a variable $P\_Base_i,\ 0 \leq i < \#vars$:

$$0 \leq P\_Base_i < \mathtt{C}$$

Note that padding a variable results in modifying the initial addresses of the other variables (see Figure 6.3). Thus, after padding, the memory variable base

addresses are computed as follows:

$$BaseAddr(Var_i) = mem_i + \sum_{k=0}^{k \leq i} P\_Base_k$$

Adding Intra-Variable Padding

The result of applying both inter- and intra-variable padding is that all base addresses and sizes of every dimension of each memory variable may change. They are initially set according to the values given by the compiler. For each memory variable $Var_i, 0 \leq i < \#vars$, we define a set of variables $\{P\_Base_i, P\_Dim_{ij}\}$, $0 \leq j < D_i$

$$0 \leq P\_Base_i, P\_Dim_{ij} < \texttt{C}$$

After padding, memory variable base addresses are computed in the following way (see Figure 6.3):

$$BaseAddr(Var_i) = mem_i + \sum_{k=0}^{k<i} (P\_Base_k + \Delta_k) + P\_Base_i$$

and the size of the dimensions are (see Figure 6.3):

$$Dim_j(Var_i) = dim_{ij} + P\_Dim_{ij}$$

Figure 6.1(b) shows our running example after tiling and intra-padding all array dimensions.

6.2 Performance Modeling

In this section, we introduce our cost model. We first describe how we model loop tiling, padding and branch predictor behavior. Then, we describe our cost function to estimate performance. Finally, we discuss the use of a GA to traverse the solution space.

6.2.1 Tiling and Padding Model

We want to improve data locality through loop tiling and padding. We focus on removing capacity misses by means of loop tiling, whereas we use padding to eliminate those conflict misses that loop tiling cannot remove.

We present a compiler strategy that combines both optimizations at the same time by implementing our static analyzer (see Section 5.5) in a parameterized way. We assume normalized loop nests[1]. Thus, each tile factor will range between 1 and the upper bound of the corresponding loop. For the pad factors, there is no need to consider large domains. Usually, if two references do not conflict on a cache of size S, they may not conflict on a cache of size nS (larger by a factor of n). Therefore, we use the cache size of the smallest cache in the hierarchy (which in practice is L1) as the domain. Results show than even smaller domains would be enough to achieve important speedups, since pad factors are generally very small.

Our measure of locality is the number of read and write misses for each cache level. More formally stated, given a loop nest L with n normalized enclosing loops $L = \{l_1, \ldots, l_n\}$, a set of tile and pad factors F, and a memory hierarchy with u levels, we define a function $MCost(L, F)$:

$$
\begin{array}{rcl}
MCost : Loops \times Factors & \longrightarrow & (\mathbb{N}, \mathbb{N}, \ldots, \mathbb{N}, \mathbb{N}) \\
(L, F) & \longmapsto & (rm_{ML1}, wm_{ML1}, \ldots, rm_{MLu}, wm_{MLu}) \\
 & where & \\
F & = & Tile\ Factors \times Pad\ Factors \\
Tile\ Factors & = & (T_i, \ldots, T_k),\ 1 \leq i, k \leq n \\
Pad\ Factors & = & (P\_Base_i, P\_Dim_{ij}), 0 \leq i < \#vars, 0 \leq j < D_i
\end{array}
$$

where rm_{Lk} (wm_{Lk}) stands for the number of read (write) misses on the k-th level cache.

Example 11 *Let us recall the optimized version of our running example shown in Figure 6.1(b), where we tile two loops and only consider intra-padding. We call the following instance of MCost to describe its locality:*

$$
\begin{array}{rcl}
MCost(L, F) & = & MCost(\{l_1, l_2, l_3\}, F) \\
 & where & \\
F & = & Tile\ Factors \times Pad\ Factors \\
Tile\ Factors & = & (T_1, T_2),\ 1 \leq T_1 \leq N,\ 1 \leq T_2 \leq N \\
Pad\ Factors & = & (P\_Dim_{ij}),\ 0 \leq i < 3,\ 0 \leq j < 2 \\
 & & 0 \leq P\_Dim_{ij} < \mathtt{C}
\end{array}
$$

[1]The loop normalization consists in transforming all the loops into a normal form. In this normal form, the lower bound and the increment are equal to one

6.2.2 Branch Model

Tiling must be applied carefully because it may increase the overhead due to the tiled code complexity. Besides, the extra levels of loops may lead to larger number of miss-predicted branches. In order to avoid a large performance degradation due to branch prediction misses, we incorporate the number of possible miss-predicted branches into our model. Notice that the same scheme can be applied to model iterations overhead.

Let L be a loop nest with n normalized enclosing loops $L = \{l_1, \ldots, l_n\}$, with upper bounds $U = \{U_1, \ldots, U_n\}$ respectively. Since current branch predictors may miss-speculate when loops finish their execution, the number of expected miss-predicted branches is:

$$MissPred(L) = \sum_{j=1}^{j \leq n} \prod_{i=1}^{i<j} U_i$$

Thus, we are modeling a *Backward Taken and Forward not Taken* scheme.

Example 12 *Let us consider our running example when* {`N=100, T1=20, T2=20`}*. For this particular example, the values of the pad factors are irrelevant since they do not affect the number of miss-predicted branches. The number of miss-speculated branches for the non-tiled version (see Figure 2.4) is:*

$$\sum_{j=1}^{j \leq 3} \prod_{i=1}^{i<j} 100 = 1 + 100 + 10^4 = 10101$$

The number of miss-predictions for the tiled version (see Figure 6.1(a)) will be:

$$\begin{aligned} \sum_{j=1}^{j \leq 5} \prod_{i=1}^{i<j} U_i &= 1 + \left\lceil \frac{100}{20} \right\rceil + \left\lceil \frac{100}{20} \right\rceil \times \left\lceil \frac{100}{20} \right\rceil + \left\lceil \frac{100}{20} \right\rceil \times \left\lceil \frac{100}{20} \right\rceil \times 100 + \\ &\quad + \left\lceil \frac{100}{20} \right\rceil \times \left\lceil \frac{100}{20} \right\rceil \times 100 \times 20 \\ &= 1 + 5 + 25 + 2500 + 5 \times 10^4 = 52531 \end{aligned}$$

which is over five times as many as the original program.

The last example shows that in order to have a tiled code that runs faster than the original one, we must have an important reduction in number of misses to compensate this overhead.

INPUT

L = $(l_1, \ldots, l_n)$ a loop nest

ML = $(ML_1, \ldots, ML_u)$ a memory hierarchy with u levels

F = $(T_1, \ldots, T_n, P_1, \ldots, P_t)$ a set of tile and pad factors

OUTPUT

LoopCost(L,M,F) = number of estimated cycles

ALGORITHM

$$
\begin{aligned}
< rm_{L1}, wm_{L1}, \ldots, rm_{Lu}, wm_{Lu} > &= MCost(L, F) \\
\text{miss\_predictions} &= MissPred(L) \\
\textit{LoopCost}(L,ML,F) &= \textstyle\sum_{l=1}^{l \le u} (\mu_{Rl} * rm_{MLl} + \mu_{Wl} * wm_{MLl}) \\
&+ \mu_{MP} * \text{miss\_predictions}
\end{aligned}
$$

WHERE

μ_{Rl} = cost of a read miss in level l

μ_{Wl} = cost of a write miss in level l

μ_{MP} = cost of a miss-predicted branch

Fig. 6.4: LoopCost algorithm.

6.2.3 Cost Model

Once we account for misses across all different levels and loop tiling overhead due to miss-predicted branches, we can calculate the loop cost. In Figure 6.4, we give a detailed description of our cost model function, *LoopCost*.

MCost calculates the locality for the loop nest L given the set of tile and pad factors, i.e., the number of read and write misses for each cache level. *MissPred* estimates the number of miss-predicted branches that the branch predictor may incur in when executing loop nest L. Then, *LoopCost* calculates the total cost of executing L. It simply adds up all different misses and the number of miss-predictions, weighting each value by its relative cost.

Values of different μ weights depend on the considered architecture. Most current processors have out-of-order execution and non-blocking caches, which makes the different penalties vary at execution time. In Table 4.1 (see Section 4.3), we give the relative costs used to model the architectures we used in our study for optimizing execution time. In order to have a more accurate model, the vendors may calculate the average penalties empirically, which may translate to better transformed codes.

6.2.4 Compiler Strategy

The main objective of a compiler strategy is to determine which transformation to apply. In our case, our main concern is to decide which tile and pad factors yield

the best results. In this subsection, we explain how we choose the parameter values guided by our cost model.

We want to find a set of tile and pad factors that minimize the *LoopCost* of a loop nest L of depth n. More formally stated:

$$MIN\ LoopCost(L, ML, F)$$

$$1 \leq T_k \leq U_k,\ 1 \leq k \leq n$$

$$0 \leq P\_Base_i, P\_Dim_{ij} < \mathtt{C}$$

$$0 \leq i < \#vars,\ 0 \leq j < D_i$$

where *LoopCost* is called the *objective function*.

6.2.5 Choosing Tile and Pad Factors

In this subsection, we use results from the original CMEs framework and compiler theory to reason about the complexity of achieving optimal tile and pad factors.

Complexity: CMEs

CMEs [52] describe cache behavior by means of Diophantine equations, and so does our model. Each set of equations defines a bounded convex polyhedron. Obtaining the number of misses is equivalent to counting the number of integer points inside those polyhedra. However, counting integer points in a general polyhedron has the same complexity as deciding whether a solution exists to a system of equalities and inequalities, which is NP-Complete [11]. So is the problem of computing the volume (i.e., number of integer points) of a polyhedron [38].

The number of integer points inside a polyhedron can be enumerated by means of Ehrhart Polynomials [34], which describe pseudo-polynomials whose coefficients in the general case are NP-hard to compute. Thus, the parameterized equations used to describe cache misses are a pseudo-polynomial function. Summing up, *LoopCost* is a pseudo-polynomial function, and hence the relationship between tiling, padding and the number of misses is nonlinear.

Tile and pad factors can only take integer values, thus, the problem of optimizing *LoopCost* can be seen as a nonlinear integer optimization (NLP) one, which again, in the general case, is NP-hard [72, 152].

Complexity: Some Results from Compiler Theory

It is common knowledge that choosing the best tile sizes is very hard, and it is considered to be a very difficult problem. However, no proof has been published that it is an NP-hard problem.

Many researchers have spent much effort looking for the best data layout. Recently it has been proved that *"Unless P==NP there is no efficient optimal algorithm for data placement that minimizes the number of misses"* [123]. That is, choosing the best pad factors is an NP-complete problem.

Notice that setting the different tile factors T_k to 1 is equivalent to solving the *"select best pad factors"* problem. On the other end, if we set all the pad factors to 0, the resulting problem is the original *"select best tile factors"* problem.

6.2.6 How to Solve Non-Linear Integer Problems

One of the challenges in NLP is that some problems exhibit local minima. Algorithms proposed to overcome this problem are named *Global Optimization*. Real functions have been studied deeply [54, 72, 147]. Unfortunately, integer functions are hard to optimize.

There are some studies based on {0,1} valued integer functions [62], but in general, this is a hard and time-consuming problem. Hence, the use of heuristics to traverse the solution space is necessary. Tabu search [55] obtains promising theoretical results, but only partial implementations have been reported so far. On the other hand, simulated annealing [89] and genetic algorithms [57, 70] have been used for years with very good results for many problems.

Why a Genetic Algorithm?

The majority of research in optimization via high-level restructuring has relied on smart heuristics and very simple models [24, 35, 93, 129, 142, 168], managing to improve program performance significantly. Current results in compiler theory [123] point out two important practical issues: (i) the use of heuristics is a must, and (ii) the preservation of information is critical to find a good solution.

Our proposal is based on the use of a very accurate cost model, thus reducing the loss of information. Then, we use a heuristic, in this case a genetic algorithm, to optimize the *LoopCost* function. We highlight the list of items why we chose a GA to solve our problem:

- GAs are especially appropriate when the relationships among variables are

not very well understood. This is the case for our *LoopCost*, which is highly non-linear.

- GAs can be used for searching in high-dimensional spaces (i.e., it can cope with lots of variables).

- In contrast with other approaches like neural networks, GAs' solutions are readable.

According to Petrank and Rawitz [123], the only efficient way to evaluate the potential of our method is comparing it with previous ones. Our experimental results show that with a small and reasonable compile-time overhead, our method outperforms all previous approaches, for all benchmarks running on a variety of modern architectures.

6.3 Implementing a Genetic Algorithm

Algorithms for function optimization are generally limited to convex-regular functions. However, there are lots of functions that are not continuous, non differentiable or multi-modal. It is common to solve these problems by means of stochastic sampling. Whereas traditional search techniques use characteristics of the problem to determine the next sampling point (e.g., Gradient), stochastic methods use non-deterministic decision rules [41].

Genetic algorithms are a particular type of stochastic methods that have been used to solve hard problems with objective functions that do not meet the properties needed to be solved by traditional methods [57]. These algorithms search in the solution space of a function simulating the nature-based process of evolution, that is, the survival of the fittest. Usually, the fittest individuals tend to reproduce more than the inferior individuals, and they survive to the next generation propagating the best genes.

GAs simulate the evolution of a population. Figure 6.5(a) shows the simplest GA. It starts from a random generated population. Then, it evolves the population by means of basic genetic operators (selection, mutation and crossover) [57] applied to individuals of the current population to produce an improved next generation.

Next, we explain how we implemented the different genetic operators and our representation of the tile and pad factors.

```
ALGORITHM:
finish := false
iters:=0
Supply a population P_0
while (not finish)
  if (not converge(P_iters))
    iters:=iters + 1
    P_iters=Selection(P_iters-1)
    P_iters=Reproduce(P_iters)
  else finish := true
  endif
endwhile
```

(a) Simple Genetic algorithm

```
ALGORITHM:
finish := false
iters := 0
Supply a population P_0
while (not finish)
  if (iters<15)
    iters := iters + 1
    P_iters:=Selection(P_iters-1)
    P_iters=Reproduce(P_iters)
  else if (iters>=15 and iters<25)
    if (not converge(P_iters-1))
      iters := iters + 1
      P_iters:=Selection(P_iters-1)
      P_iters=Reproduce(P_iters)
    else finish := true
    endif
  else finish := true
  endif
endwhile
```

(b) Our Genetic algorithm

Fig. 6.5: Different implementations of a Genetic algorithm.

6.3.1 Genetic Algorithm Parameters

The use of GAs requires the determination of the following issues: chromosome representation, selection function, genetic operators, the creation of the initial population, and the termination criterion.

Each individual is made up of a set of chromosomes which represents the variables. In our work, each individual is one configuration of tiling/padding (identified by all tile factors and the inter- and intra-variable pad factors). Each chromosome represents one single factor, either a tile or a pad factor. The fitness of those individuals is computed using the objective function (in our case, *LoopCost* in Figure 6.4). The fittest individual is the one that has a set of tile and pad factors that results in the smallest cost according to our cost function *LoopCost*.

A chromosome representation is needed to represent each individual in the population. Genetic algorithms require the natural parameter set of the optimization problem to be coded as a finite-length string over some finite alphabet such as {0,1}. Thus, each chromosome is made up of a sequence of genes from a certain alphabet.

Loop considered l, with $U = 11$

$\Downarrow$ (1) We compute k $\Downarrow$
$k = \lceil log_2 11 \rceil$ $(+1\ if\ odd) = 4$

$\Downarrow$ (2) We set up function g $\Downarrow$
$g : [0 \ldots 15] \longrightarrow [1 \ldots 11]$

$\Downarrow$ (3) We enumerate function g $\Downarrow$

$g(0)$=1	$g(1)$=1	$g(2)$=2	$g(3)$=3
$g(4)$=3	$g(5)$=4	$g(6)$=5	$g(7)$=5
$g(8)$=6	$g(9)$=7	$g(10)$=7	$g(11)$=8
$g(12)$=9	$g(13)$=9	$g(14)$=10	$g(15)$=11

Fig. 6.6: Example of mapping between representation values and tile factors.

Representing Tile Factors

It has been shown that using large alphabets gives better results [115]. For representing the tile factors, we have experimentally observed that using the alphabet {00, 01, 10, 11} produces good results.

The function to transform the chromosome values into tile sizes is not the identity function. Tile factor T_i can take any value in the range $[1 \ldots U_i]$. On the other hand a chromosome is represented by a sequence of genes encoded in a binary representation. Thus, each chromosome will be represented by a value in the range $[0 \ldots 2^k\text{-}1]$ where k is $\lceil log_2 U_i \rceil$. If k is an odd number, k is increased by 1 due to the alphabet we have used to represent genes. Thus, there are more values in the representation range for a chromosome than possible tile size values. Therefore, we need a function to map values from the domain $[0 \ldots 2^k\text{-}1]$ to the range $[1 \ldots U_i]$.

Let g be the function that represents the tile factor for each possible value of a chromosome. We define it as follows:

$$\begin{aligned} g : [0 \ldots 2^k - 1] &\longrightarrow [1 \ldots U_i] \\ x &\longmapsto \left\lfloor \frac{x * (U_i - 1)}{2^k - 1} \right\rfloor + 1 \\ where \qquad & k = \lceil log_2 U_i \rceil \ (+1\ if\ odd) \end{aligned}$$

Figure 6.6 illustrates an example of how this function works. It can be seen that every possible tile factor has at least one representation.

Example 13 *Let us codify two tile factors $\{T_1, T_2\}$ for two nested loops with upper bounds $\{U_1 = 40, U_2 = 100\}$. Each tile factor is represented by one chromosome. Thus, the first chromosome is represented by 3 genes (and function g_1), and the second one by 4 genes (and the corresponding function g_2). Thus, the value 27 (011011) and 74 (01001010) correspond to the tile factors 17 (g_1(27)=17) and 29 (g_2(74)=29) respectively, and are represented by the following genes:*

$$\underbrace{\underbrace{01}_{gene_{1_0}=1}\ \underbrace{10}_{gene_{1_1}=2}\ \underbrace{11}_{gene_{1_2}=3}}_{chromosome_1}\ \underbrace{\underbrace{01}_{gene_{2_0}=1}\ \underbrace{00}_{gene_{2_1}=0}\ \underbrace{10}_{gene_{2_2}=2}\ \underbrace{10}_{gene_{2_3}=2}}_{chromosome_2}$$

Representing Pad Factors

Representing pad factors is easier than representing tile factors due to the fact that each pad factor belongs to the range $[0, \ldots \mathtt{C}-1]$.[2] Thus, we have used the alphabet $\{0, \ldots, 2^t-1\}$, where t is the greatest divisor of the $\lfloor log_2\ \mathtt{C} \rfloor$ that is lower than $log_2\mathtt{C}$. This is, the largest value of t that guarantees that a single pad factor consists of at least two genes for every cache size. This is not a restriction because the compilers know the cache size. Thus, this computation can be done automatically.

Example 14 *Let us assume a 32KB cache. Thus, $log_2(32 \times 2^{10}) = 15$. The set of divisors is $\{1, 3, 5, 15\}$. Hence, the greatest divisor less than 15 is 5, and we will use the alphabet $\{0, \ldots, 31\}$, representing each single pad factor with 3 genes. For instance, a pad factor of 10017 is represented by the following three genes:*

$$\underbrace{\underbrace{01001}_{gene_0=9}\ \underbrace{11001}_{gene_1=25}\ \underbrace{00001}_{gene_2=1}}_{chromosome}$$

[2]If the cache size is not a power of 2, we consider the largest cache size which is power of 2 and smaller than the considered cache.

Genetic Operators

Genetic operators provide the basic search mechanism of the GAs, creating new solutions based on the solutions that already exist. The *selection* of individuals to produce successive generations plays an extremely important role. A common selection approach assigns probability of selection to each individual depending on its fitness. Individuals with higher fitness have a higher probability of contributing one or more offsprings to the next generation. Then, individuals are selected depending on this probability. Let us have a population of size N (i.e., a population with N individuals). A selection scheme consists of choosing N individuals from the N individuals of the previous generation. We have adopted one of the selection schemes that gives better results, which is known as *remainder stochastic selection without replacement* [57]. This selection scheme allows an individual to be chosen more than once, so that best individuals contribute with more offsprings.

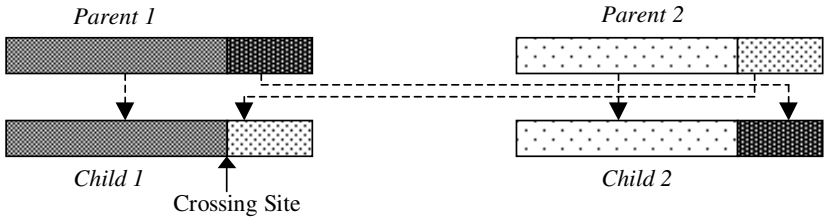


Fig. 6.7: Schematic of simple crossover.

The next step consists of pairing the chosen individuals and applying *crossover*. *Crossover* takes two individuals and produces two new individuals with a given probability, merging the genetic material in a random point (named cross site). In the case they do not crossover, both individuals are added to the new population (see Figure 6.7). Finally, *mutation* changes one individual to produce a new one by flipping some of its genes. Both crossover probability and mutation probability have to be determined empirically, and are related to the size of the population.

Convergence Criterion

The GA must be provided with an initial population (see Figure 6.5(a)), that is created randomly. GAs move from generation to generation, and even though other criteria can be used [57], the usual termination criterion is the number of generations.

Our experiments have shown that an initial population of size equal to 30, with a crossover probability of 0.9 and a mutation probability of 0.005, gives near-optimal results in most of the cases after 15 generations. However, in some other cases the near-optimal results are obtained after a number of generations between 15 and 25. Figure 6.5(b) shows our particular GA, where *converge()* is a function which decides when the population is homogeneous enough. We consider that a population converges when the best individual has a cost smaller than 2% with respect to the average of its generation. We have observed that for the evaluated loops, this convergence criterion is only achieved if the population is close to the optimal.

6.4 Example of a Genetic Algorithm

In this section we show an example of an iteration of the genetic algorithm. Let us recall our running example in its tiled and padded version (see Figure 6.1(b)). We consider the problem of finding the best tile and pad sizes for a 4KB cache when `N=100`.

The steps to set up a genetic algorithm are as follows:

1. Identify variables.
2. Identify domains of those variables.
3. Decide representation.

We review these three steps explained in Section 6.3 by constructing a GA for the problem stated above. Then, we will show how the GA works by illustrating some iterations.

6.4.1 Setting Up the GA

According to the code shown in Figure 6.1(b), our problem depends on 8 different variables. There are two tile factors, `T1` and `T2`, and 6 intra-pad factors, `P_Dim00`, `P_Dim01`, `P_Dim10`, `P_Dim11`, `P_Dim20` and `P_Dim21`.

First, we determine the domains according to Section 6.2.1:

$1 \leq \texttt{T1} \leq 100$ $\quad$ $1 \leq \texttt{T2} \leq 100$
$0 \leq \texttt{P\_Dim00} < 4096$ $\quad$ $0 \leq \texttt{P\_Dim01} < 4096$
$0 \leq \texttt{P\_Dim10} < 4096$ $\quad$ $0 \leq \texttt{P\_Dim11} < 4096$
$0 \leq \texttt{P\_Dim20} < 4096$ $\quad$ $0 \leq \texttt{P\_Dim21} < 4096$

Once the domains are determined, the next step consists on deciding the representation that is used for each parameter. In order to do that, we follow the steps explained in Section 6.3.1.

For representing the tile factors, we will need 8 bits. We obtain this value as follows:

$$\lceil log_2 100 \rceil = \lceil 6.64 \rceil = 7$$

since 7 is odd, we add 1 and choose 8. Thus, each tile factor will be represented by 4 genes (each gene $\in \{00, 01, 10, 11\}$).

Pad factors are somehow easier to represent. We start considering the cache, which is 4KB. Thus, $log_2 4^{10} = 12$, which means that each chromosome will consist of 12 bits. The set of divisors is $\{1, 2, 4, 6, 12\}$, therefore we choose to represent each gene with 6 bits.

Summing up, each individual (that represents a set of tile and pad factors), will be made up of 88 bits. This is, 20 genes or 8 chromosomes:

$$\underbrace{\underbrace{xx}\ \underbrace{xx}\ \underbrace{xx}\ \underbrace{xx}}_{\texttt{T1}} \cdots \underbrace{\underbrace{xxxxxx}\ \underbrace{xxxxxx}}_{\texttt{P\_Dim00}} \cdots \underbrace{\underbrace{xxxxxx}\ \underbrace{xxxxxx}}_{\texttt{P\_Dim21}} \qquad x \in \{0, 1\}$$

6.4.2 Iterating the GA

Now, we show in detail how the genetic operators are applied in order to improve the population. Let us consider that the selection method has chosen two individuals, I_1 and I_2 such that:

$$I_1 = \underbrace{\underbrace{xx}\ \underbrace{xx}\ \underbrace{xx}\ \underbrace{xx}}_{\texttt{T1}} \cdots \underbrace{\underbrace{xxxxxx}\ \underbrace{xxxxxx}}_{\texttt{P\_Dim00}} \cdots \underbrace{\underbrace{xxxxxx}\ \underbrace{xxxxxx}}_{\texttt{P\_Dim21}} \qquad x \in \{0, 1\}$$

$$I_2 = \underbrace{\underbrace{yy}\ \underbrace{yy}\ \underbrace{yy}\ \underbrace{yy}}_{\texttt{T1}} \cdots \underbrace{\underbrace{yyyyyy}\ \underbrace{yyyyyy}}_{\texttt{P\_Dim00}} \cdots \underbrace{\underbrace{yyyyyy}\ \underbrace{yyyyyy}}_{\texttt{P\_Dim21}} \qquad y \in \{0, 1\}$$

According to Section 6.3.1, we first apply crossover, and then we mutate the new individuals.

Crossover is NOT Applied

Let us consider first that crossover is not applied. Thus, we obtain two new individuals which are exactly the same as before, $I'_1 = I_1,\ I'_2 = I_2$. The next step consists

of mutating some bits of these individuals. After mutating, we obtain these two new individuals which are added to the new population:

$$I_1'' = \underbrace{\overline{x}x\ xx\ xx\ xx}_{\texttt{T1}} \cdots \underbrace{\overline{x}xxxxx\,xxxxxx}_{\texttt{P\_Dim00}} \cdots \underbrace{xxxxxx\,xxx\overline{x}xx}_{\texttt{P\_Dim21}} \qquad x \in \{0,1\}$$

$$I_2'' = \underbrace{yy\ y\overline{y}\ yy\ yy}_{\texttt{T1}} \cdots \underbrace{yyyyyy\,yyy\overline{y}yy}_{\texttt{P\_Dim00}} \cdots \underbrace{yy\overline{y}yyy\,yyyyyy}_{\texttt{P\_Dim21}} \qquad y \in \{0,1\}$$

Crossover IS Applied

When crossover is applied, we obtain two new individual which are obtained merging the genes from both original individuals. In our case, the crossing site is at the 5th bit, which gives rise to these two new individuals:

$$I_1' = \underbrace{xx\ xx\ xy\ yy}_{\texttt{T1}} \cdots \underbrace{yyyyyy\,yyyyyy}_{\texttt{P\_Dim00}} \cdots \underbrace{yyyyyy\,yyyyyy}_{\texttt{P\_Dim21}} \qquad x,y \in \{0,1\}$$

$$I_2' = \underbrace{yy\ yy\ yx\ xx}_{\texttt{T1}} \cdots \underbrace{xxxxxx\,xxxxxx}_{\texttt{P\_Dim00}} \cdots \underbrace{xxxxxx\,xxxxxx}_{\texttt{P\_Dim21}} \qquad x,y \in \{0,1\}$$

Finally, mutation is applied, which results in the two individuals that are eventually added to the new population:

$$I_1'' = \underbrace{x\overline{x}\ xx\ xy\ yy}_{\texttt{T1}} \cdots \underbrace{yyyyyy\,yy\overline{y}yyy}_{\texttt{P\_Dim00}} \cdots \underbrace{yyyyyy\,yyyyyy}_{\texttt{P\_Dim21}} \qquad x,y \in \{0,1\}$$

$$I_2'' = \underbrace{yy\ yy\ yx\ xx}_{\texttt{T1}} \cdots \underbrace{xxxxxx\,xx\overline{x}xx\overline{x}}_{\texttt{P\_Dim00}} \cdots \underbrace{xx\overline{x}xxx\,x\overline{x}xxxx}_{\texttt{P\_Dim21}} \qquad x,y \in \{0,1\}$$

6.5 Experiments Setup

To evaluate our method, we have implemented our algorithms, executed the original and transformed versions and collected execution times and number of misses for different cache levels. The actual number of misses when executing programs on the Pentium-4 platform are obtained by means of the performance counters. Otherwise they are obtained by means of our *EstimateMisses* (see Section 5.5.2).

In order to evaluate our ability to improve data locality, we start studying padding and tiling separately. Then, we combine both of them and report results for a set of kernels.

6.5.1 Padding

Since the objective of padding is removing conflict misses, we optimized those programs from SPECfp95 that present a high number of conflict misses, which are TOMCATV and SWIM [46]. In addition, their miss ratio is highly affected by cache size. We have chosen the most time-consuming loop nests from each program that in total represent between the 90-100% of the whole execution time, using the reference input data. The codes can be found in Appendix B.

A fully-associative cache has been evaluated as a reference point to estimate the amount of conflict misses that are not removed by the padding technique. In order to measure our ability to improve locality, we compare our padding algorithm with Rivera and Tseng's state-of-the-art technique [129, 130].

6.5.2 Tiling

An overview of the five kernels that have been evaluated can be seen in Table 4.3 in Section 4.4. For all of them, we have studied a set of different sizes that are explained in the different experiments. We chose these kernels because they exhibit high number of capacity misses.

We also determined the effectiveness of our method by comparing it with other methods which represent the state-of-the-art:

- **lrw**: Lam *et al.* [93] choose the largest non-conflicting square tile.
- **tss**: Rivera and Tseng [131] extend Coleman and McKinley's [35] Euclidean GCD algorithm.

6.6 Evaluation

In this section, we evaluate our approach. We first present results for a set of experiments where padding and tiling are applied in isolation. Then, we analyze the efficiency of our approach, which consists on applying both of them.

6.6.1 Padding

Table 6.1 shows, for the two programs analyzed, the miss ratio of a direct-mapped cache before and after applying inter-variable padding. Since the objective of padding is to eliminate conflict misses, inter-variable padding provides a huge improvement in miss ratio for TOMCATV and SWIM. Note that for both programs, a

Program	Cache Size	NO Padding (%)	Inter-Padding (%)
TOMCATV	32KB	9.6	8.8
	16KB	14.8	11.8
	8KB	46.0	21.6
	4KB	72.1	52.0
SWIM	32KB	8.1	7.1
	16KB	28.8	7.2
	8KB	62.9	7.8
	4KB	77.9	8.2

Table 6.1: Average miss ratios for TOMCATV and SWIM for a set of direct-mapped caches. Cache line is 32B.

small improvement is obtained for a 32KB cache. This is caused by the fact that almost no conflicts arise for 32KB caches or bigger for these programs due to the relatively small working set of the SPECfp95 applications. However, the smaller the cache the bigger the miss ratio and the bigger the improvement that inter-variable padding obtains.

For the SWIM program, the miss ratio grows from 8.1% to 28.8%, 62.9%, and 77.9% when the cache is reduced from 32KB to 16KB, 8KB, and 4KB respectively. However, when we apply inter-variable padding, the miss ratio is kept almost constant (7.1%, 7.2%, 7.8% and 8.2% respectively). This is because most of the misses of this program are caused by conflicts between different data structures (inter-variable conflict misses) and the algorithm practically obtains the optimal padding among them.

For the TOMCATV program, the miss ratio also grows significantly when the cache size is reduced (9.6%, 14.8%, 46.0%, and 72.1% respectively for the different cache sizes). In this program, we also obtain a considerable improvement when applying inter-variable padding for caches smaller than 32KB. However, the miss ratio after inter-variable padding varies significantly with the cache size (8.8%, 11.8%, 21.6%, and 52%). This variation is caused by intra-variable conflict misses (e.g., conflicts among distinct rows and columns of the same array) whose frequency also grows when the cache size is reduced. Inter-variable padding does not remove the latter type of conflicts, which are the target of intra-variable padding.

Figure 6.8 details the miss ratio for the main loop nests of TOMCATV and SWIM (note the different scales for the different cache sizes). The figure shows the miss ratio for each loop before and after applying inter-variable padding. It also shows the miss ratio for a fully-associative cache.

For the SWIM program loop nest 1 has practically no improvement due to inter-

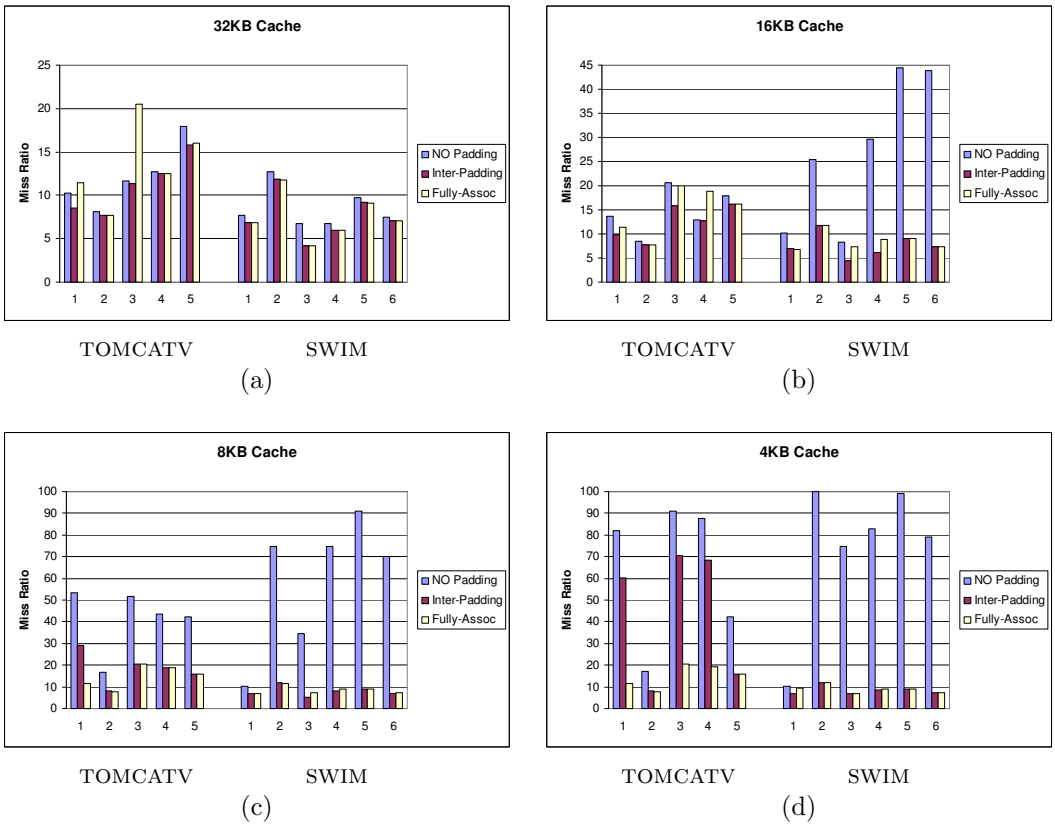


Fig. 6.8: Miss ratio before and after inter-variable padding for a set of direct-mapped caches.

variable padding (excepting a slight improvement due to alignment) because it has no conflict misses. Note also that this loop nest has almost the same miss ratio regardless of the cache size. On the other hand, loop nests 2 to 6 have an extremely large miss ratio. As an extreme case, loop nest 2 has a miss ratio close to 100% for a 4KB cache, which after inter-variable padding is reduced to 11.8%. Note that inter-variable padding removes all the conflict misses for all SWIM loops since the miss ratio after inter-variable padding and the fully-associative miss ratio are practically identical.

The TOMCATV program has several loop nests that deserve special comments. For the 32KB and 16KB, the proposed inter-variable padding technique practically removes all conflict misses. For the 8KB cache, inter-variable padding removes all conflict misses from all loop nests except for loop 1. In this case, inter-variable padding reduces the miss ratio from 53.6% to 29.2% but not all conflict misses are removed since the fully-associative cache miss ratio is 11.4%. An analysis of this loop shows that there are also intra-conflict misses.

In the case of a 4KB cache, inter-variable padding achieves about the same miss ratio as a fully-associative cache for loop nests 2 and 5. As a noticeable case, the miss ratio of loop 5 has been reduced from 42.3% to 15.8%. For the other loop nests there is a significant improvement but the miss ratio is still far from that of the fully-associative cache. An analysis of these three loop nests revealed that most of the remaining misses are intra-variable conflict misses.

Intra-Variable Padding

The objective of intra-variable padding is to eliminate those intra-variable conflict misses that inter-padding cannot remove.

We have shown that TOMCATV is the only program that has a significant intra-variable conflict miss ratio, in particular for caches of 4KB and 8KB. Figure 6.9 shows the miss ratio for the different loop nests of the TOMCATV program. The figure shows the miss ratio for each loop after applying inter- and intra-variable padding. It also shows the miss ratio before padding and that of a fully-associative cache. As we observed before, inter-variable padding does not remove all conflict misses because there are intra-variable conflict misses. Intra-variable padding achieves about the same miss ratio as the fully-associative cache, which means that the proposed padding algorithm removes practically all conflict misses.

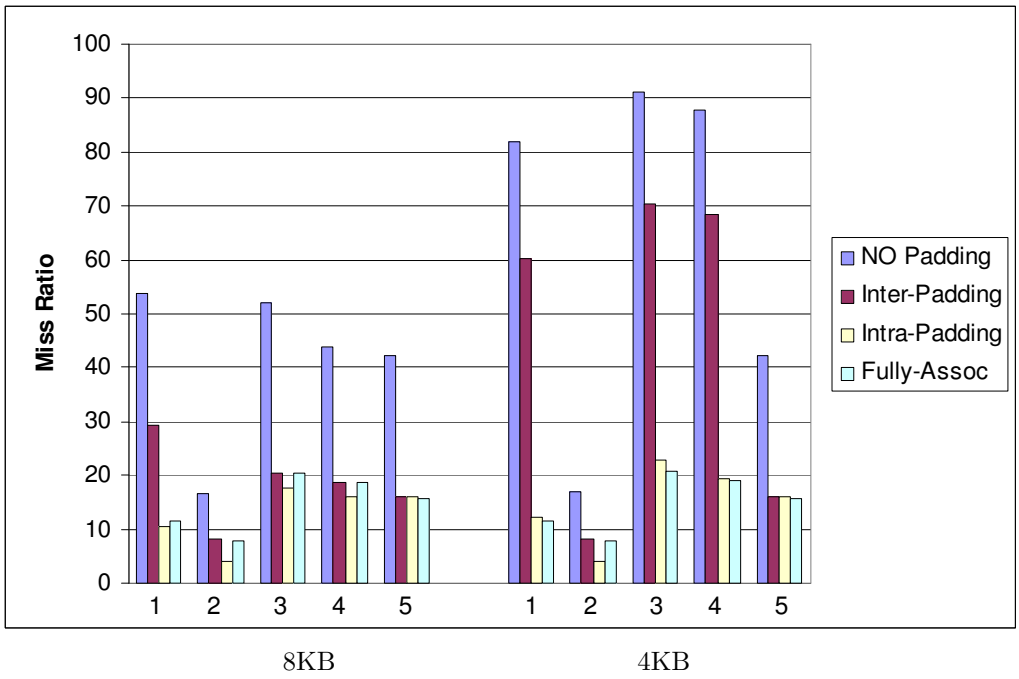


Fig. 6.9: Miss ratio for different TOMCATV loop nests before and after inter- and intra-variable padding.

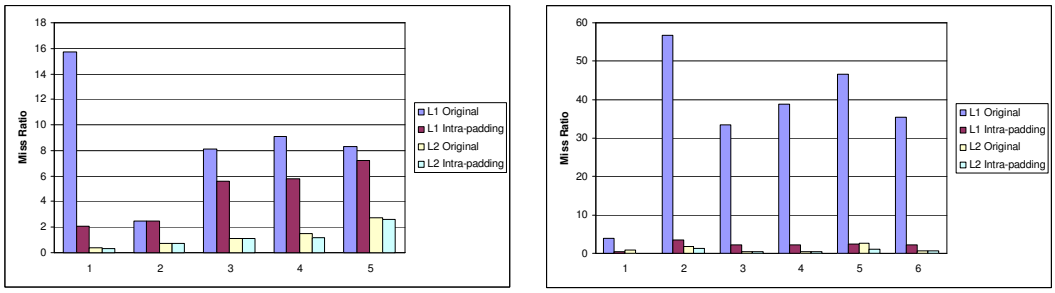
Miss Ratio Results

We have obtained the miss ratios for the most significant loop nests from TOMCATV and SWIM after running them on the Pentium-4 platform. Figure 6.10 shows the miss ratios for both L1 and L2[3] caches before and after applying intra-padding. We use the miss penalties shown in Table 4.1 to calculate the *MCost* (see Section 6.2.3) for each loop nest. We assume that read and write misses are equally expensive. Intra-variable padding reduces the average miss penalty for the TOMCATV program by 12.4%, whereas it reduces the average miss penalty by 140.8% for the SWIM program.

Performance Results

Figures 6.11 and 6.12 show the runtime improvements. We have executed the original and padded versions on the four considered platforms (see Table 4.1). Notice

[3]L2 misses are calculated w.r.t. the total number of memory accesses.



(a) TOMCATV (b) SWIM

Fig. 6.10: L1 and L2 miss ratios before and after intra-padding for the Pentium-4.

Name	Problem Size
MATMUL	100, 200, 500, 1000, 2000
MATVEC	100, 200, 500, 1000, 2000
T2D	100, 200, 500, 1000, 2000
ADI	100, 200, 500, 1000, 2000
VPENTA	128

(a) Problem sizes for evaluating the reduction in number of capacity misses.

Name	Size 1	Size 2
MATMUL	400+50i	1000+50i
MATVEC	500+43i	1000+43i
T2D	2000+53i	4000+53i
ADI	2000+53i	4000+53i
VPENTA	1028+47i	2056+47i

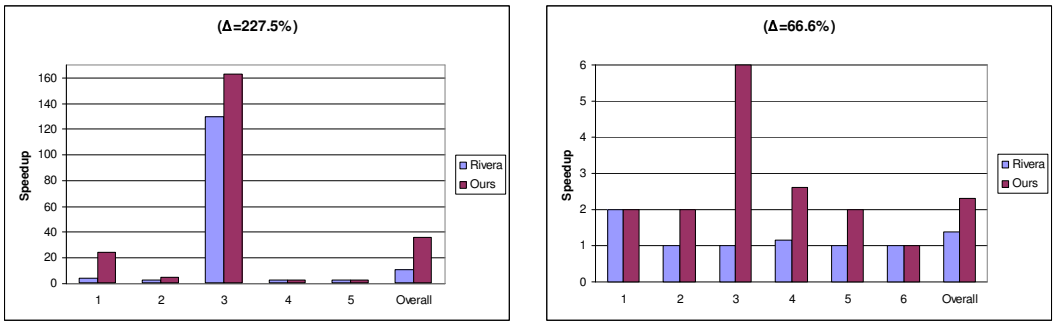
(b) Problem sizes ($i = 0 \ldots 14$) for evaluating execution time.

Table 6.2: Problem sizes for evaluating tiling.

the different scales for each chart. We have also compared our approach to selecting pad sizes with Rivera and Tseng's algorithm [129]. The first column presents the speedups achieved running Rivera and Tseng's method. We use the best result yielded by their two approaches PAD and PADLITE. The second column shows the speedups obtained by our approach. We observe that in all cases our approach performs better, with relative speedups with Rivera and Tseng's ranging between 1% and 227% for the TOMCATV program and 4% and 66% for the SWIM program.

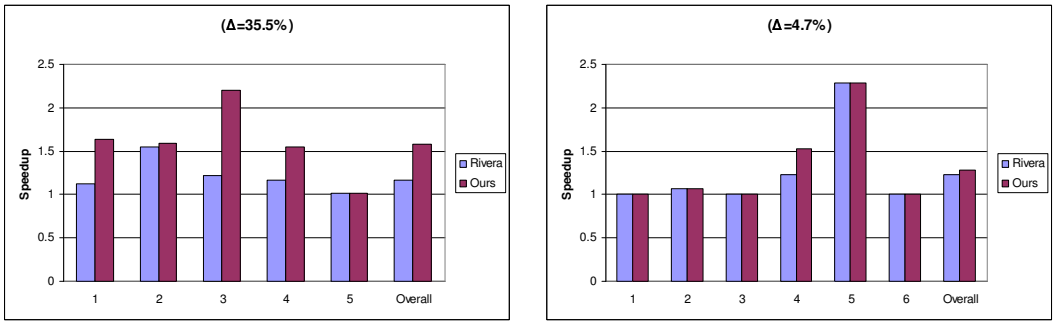
6.6.2 Tiling

We now present results for our evaluation of tiling.



TOMCATV SWIM

(a) Pentium-4



TOMCATV SWIM

(b) Alpha-21264

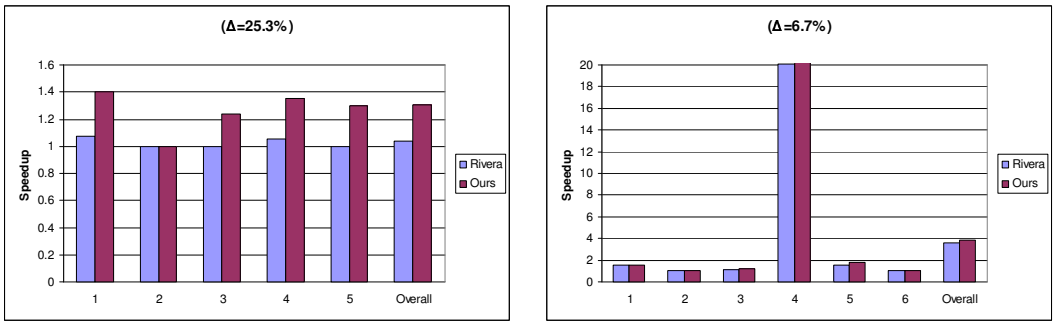
Fig. 6.11: Speedups of the padded versions compared to the original programs. Δ stands for the relative speedup of our method compared to Rivera and Tseng's.

Removing Capacity Misses

We first show the ability of tiling to reduce the number of capacity misses.[4] In order to investigate that, we have evaluated the replacement miss ratio of the studied kernels for a set of problem sizes shown in Table 6.2(a). We do not include the compulsory misses since tiling does not change them. Unlike the SPECfp95, we can change the size of the working sets. Thus, we use bigger caches and bigger working sets according to today's workloads.

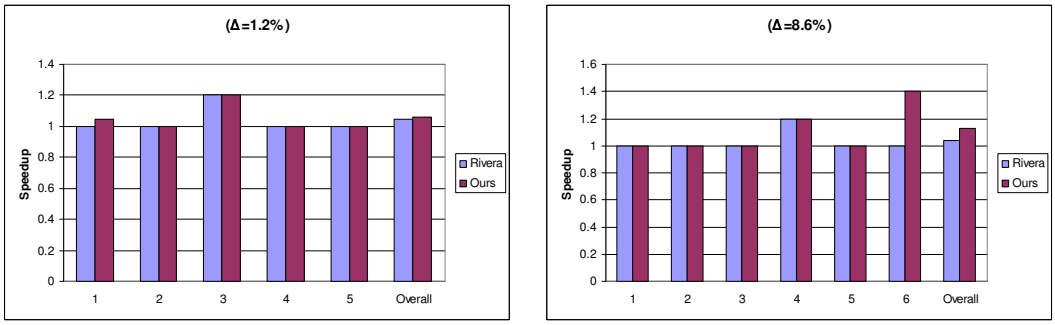
We show the results in Figure 6.13. We observe that tiling practically removes all replacement misses for almost all kernels. However, the replacement miss ratio

[4]CMEs replacement misses account for both capacity and conflict misses.



TOMCATV SWIM

(a) UltraSparc-III



TOMCATV SWIM

(b) Itanium

Fig. 6.12: Speedups of the padded versions compared to the original programs. Δ stands for the relative speedup of our method compared to Rivera and Tseng's.

obtained for VPENTA is still rather high for all cache sizes due to conflict misses. To confirm this intuition, we have applied tiling and padding for this kernel, which removes all replacement misses (i.e., conflict and capacity misses). This case illustrates the need for applying tiling and padding in concert in order to remove both conflict and capacity misses.

Loop Tiling Overhead

We now present the importance of considering the loop tiling overhead. We show that without an accurate estimate of the penalty of miss-predicted branches, the degradation in performance can be severe.

The results of our set of experiments are shown in Figure 6.14. In order to prove

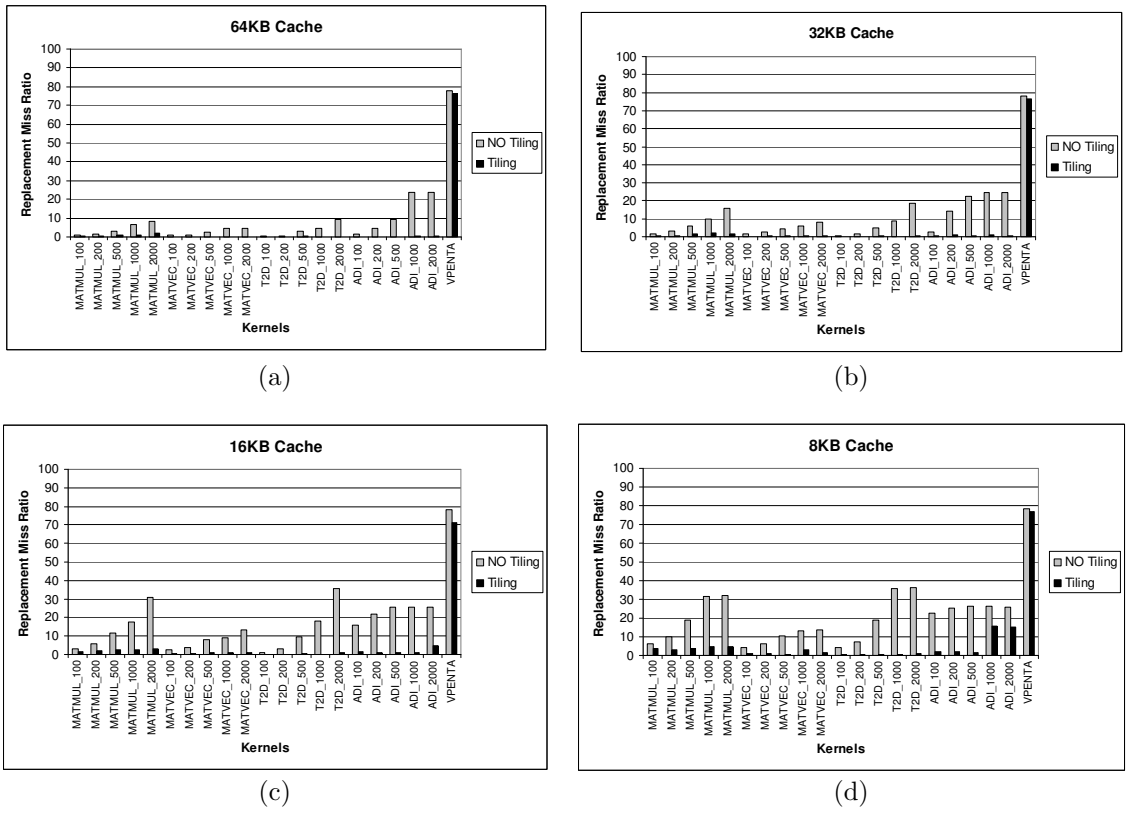


Fig. 6.13: Miss ratio before and after tiling for a set of direct-mapped caches. Cache line is 32B.

the importance of considering branch predictor behavior, we have analyzed different penalty values for the Pentium-4 processor for the problem sizes shown in the second column of Table 6.2(b). For the sake of comparison, we only consider the effects of tiling; we have run our approach for obtaining the best tile sizes considering different values of μ_{MP}, and compared the execution times to that of the selected penalty ($\mu_{MP} = 20$). We present results in terms of slowdowns. We can see that in general, execution time converges smoothly to the estimated penalty, which confirms our intuition. When the penalty is set to small values, the degradation in performance may be very important (up to 34% for MATMUL). This is because we generate tiles that are very small in order to minimize memory penalty, though incurring in a high overhead due to the increased number of miss-predicted branches. On the other hand, if we set large penalty values, we prioritize branches overhead, thus, tiles are bigger but we incur in more misses that can degrade performance.

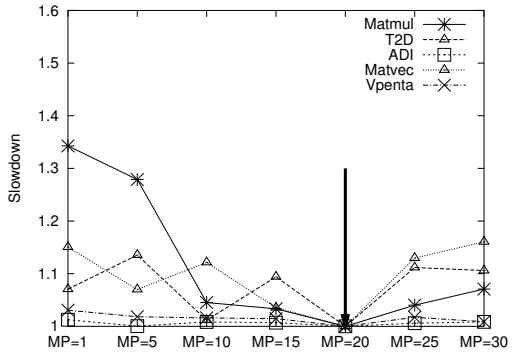
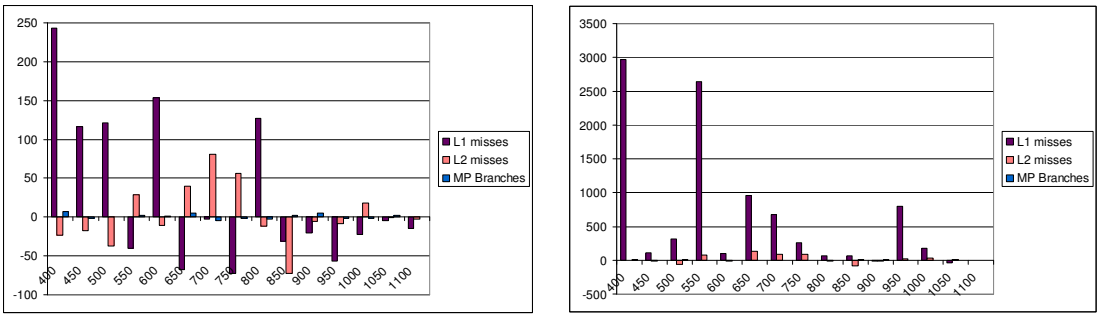


Fig. 6.14: Impact of branch miss-prediction overhead for the Pentium-4 processor. Results are normalized to our estimated penalty, $\mu_{MP} = 20$.

LoopCost Results

We have obtained the miss ratios for the different problem sizes from MATMUL as shown in the second column of Table 6.2(b) after running them on the Pentium-4 platform. Figure 6.15 shows the number of L1 and L2 misses for **lrw** and **tss**



(a) **tss** (b) **lrw**

Fig. 6.15: Run-time information of the three different tiling algorithms for the execution on the Pentium-4 platform.

normalized to those of our approach. We also show the number of modeled miss-predicted branches (*MissPred* in Section 6.2.3) based on the selected tile sizes. Note that the resulting tiled code of our approach is always the fastest.

Since our approach tries to optimize overall performance, it does not focus on removing L1 cache misses. Instead, it takes into account all different factors. Thus,

we can observe that the other approaches may yield less misses on the L1 or L2 cache level, or have less miss-predicted branches. However, our approach considers all possible factors, and the resulting program runs faster.

Performance Results

While **lrw** and **tss** can be applied to any loop nest, they were originally thought for programs involving matrix operations, and especially, for tiling matrix multiplication.

In this subsection we compare our tile selection approach with them for the matrix multiplication kernel. We present results for 5 possible loop orders, IKJ, JIK, JKI, KIJ and KJI (the remaining IJK order is used in the next subsection). We use 15 different matrix sizes:

$$\mathtt{N} = 1000 + 53i,\ 0 \leq i \leq 14$$

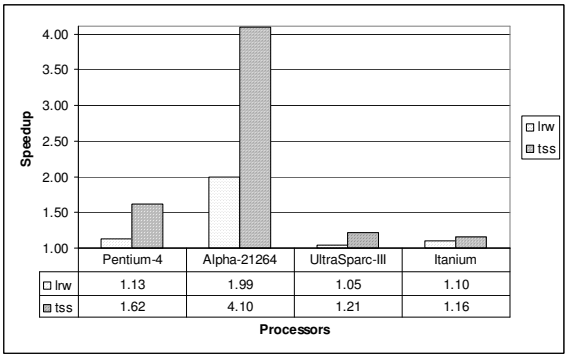
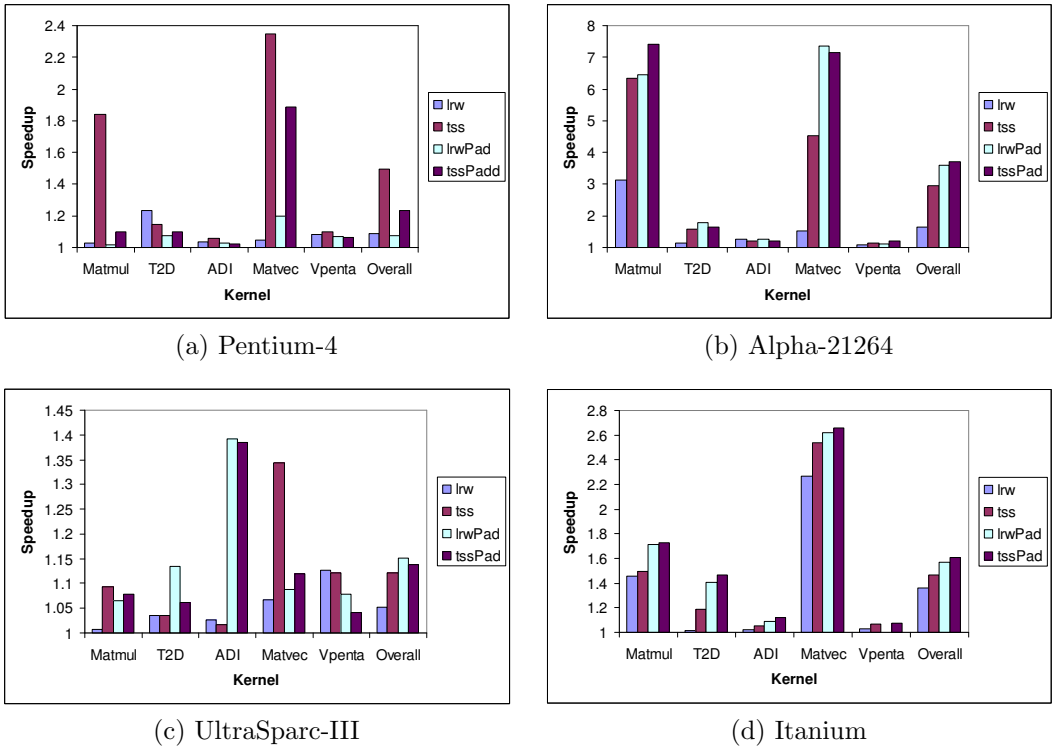


Fig. 6.16: Speedups for 5 different loop orders of the MATMUL kernel.

Figure 6.16 shows the average speedups of our method (only tiling is applied) compared to **lrw** and **tss**. For obtaining these results, we have run all different approaches to select the best tile sizes for each platform. Then, we have executed the tiled version measuring the actual execution time. The results show that our approach outperforms these two techniques significantly, up to 310% for **tss** on the Alpha processor. We also show that our approach is better than both techniques for all platforms, with improvements ranging between 5% and 310%.

6.6.3 Tiling and Padding



(a) Pentium-4 (b) Alpha-21264

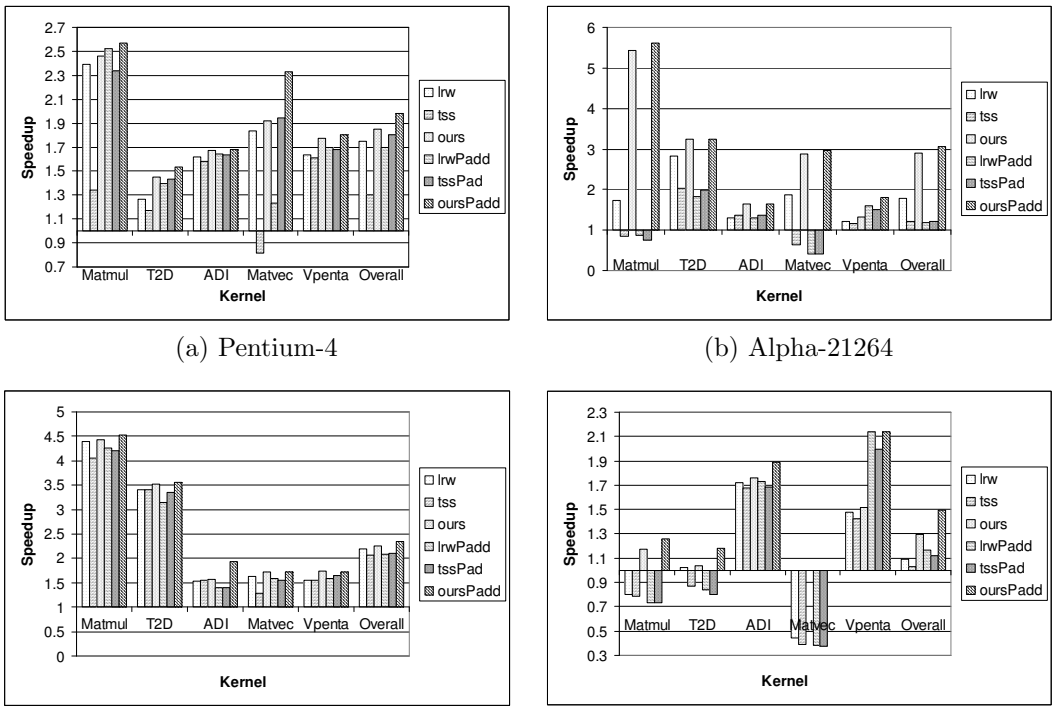
(c) UltraSparc-III (d) Itanium

Fig. 6.17: Speedup obtained by our approach compared with **lrw** and **tss** algorithms.

We now present results for a set of common kernels that may benefit from tiling and padding. We showed the problem sizes in Table 6.2(b). The second column shows the sizes considered for the Pentium-4, Alpha-21264 and UltraSparc-III, whereas the third column shows the sizes used for experimenting on the Itanium machine. We chose different sizes for Itanium in such a way that blocking could be useful to enhance performance.

In order to see the effectiveness of our method, we have compared our approach to selecting tile and pad factors with **lrw** and **tss**. Figure 6.17 shows, for each machine, to what extent our method is better in terms of execution time of the optimized codes.

We first consider the results where only loop tiling is applied. For each program, the first two bars report the speedup compared to **lrw** and **tss** respectively. In all cases our method yields better results than previous approaches. Our ability



(a) Pentium-4 (b) Alpha-21264

(c) UltraSparc-III (d) Itanium

Fig. 6.18: Speedup of all approaches w.r.t. the original program.

to select tile factors results in important runtime improvements; on average, our transformed code runs 8% and 49% faster on a Pentium-4 compared to **lrw** and **tss**. On the Alpha machine results are even more impressive, with average speedups of 63% and 195% respectively.

Now, we consider results where both tiling and padding techniques are applied. The second set of bars reports the speedup compared to **lrwPad** and **tssPad**, enhanced versions of **lrw** and **tss** where padding is allowed [131]. Note that the memory requirement for all methods was roughly the same. We can see that the speedup is smaller on the Pentium-4, where our transformed code runs 7.7% and 26% faster than **lrwPad** and **tssPad** respectively. However, the difference increases on the other three platforms, with the most significant results showing for the Alpha (260% and 271%).

Finally, in order to see to what extent tiling and padding help enhancing program's performance, we show in Figure 6.18 the speedups that the different approaches (with and without padding) obtain w.r.t. the original kernel. The applica-

tion of padding on **lrw** and **tss** does not always translate to a better performance. Padding improves especially **tss** on the Pentium-4, but it yields worse results on the Alpha machine. On the other hand, our approach applies padding selectively. Our accurate cost model guides the selection of tile and pad factors; if padding is not useful, our cost model will predict a performance degradation, so pad factors will be set to 0. Overall, our approach obtains (98%, 204%, 135%, 49%) average speedups on the Pentium-4, Alpha, UltraSparc-III and Itanium respectively. Combining the other methods with padding, **lrwPad** obtains (69%, 19%, 107%, 16%) and **tssPad** (20%, 80%, 109%, 11%). Otherwise, their speedups are (74%, 78%, 119%, 9%) and (30%, 20%, 105%, 3%) for **lrw** and **tss** respectively.

Note that the use of an accurate model allows us to obtain always a version of the code that it is not worse than the original one. For instance, when optimizing MATVEC for the Itanium platform our cost model determines that tiling is not useful, thus we do not apply it. However, the other approaches do not have an accurate model that guides the transformations, which results, some times, in optimized codes that run slower than the original version.

6.6.4 Compile-Time Overhead

Clearly, for our method to be considered a realistic optimization approach, it must be shown that the compilation time required is small enough to be practical. Although a precise cost model combined with a GA can find very good results, the compilation time required for that may be infeasible. In order to investigate this, we have collected the execution time needed to obtain tile and pad factor for all our experiments. We account for 15 problem sizes for each of the 5 kernels.

Processor	MIN	MAX	AVG
Pentium-4	1.8	14.5	**4.6**
Alpha-21264	0.1	11.9	**3.6**
UltraSparc-III	0.87	16.5	**5.8**
Itanium	0.4	17.0	**5.5**

Table 6.3: Compile-time overhead when selecting tile and pad factors on a Pentium-4 running at 1.6GHz.

Table 6.3 shows the average times needed to generate the optimized versions (including both tiling and padding) for each architecture. We see that in the worst case, it takes an average of 5.8 seconds to optimize a code. We believe that this amount of time is reasonable for a static compiler.

6.6.5 Summary

Overall, we have shown the effectiveness of our method to select tile and pad factors. We first have presented results where only padding and tiling are applied. Then, we have reported results that highlight the importance of modeling the branch predictor behavior. Later, we have seen that our approach outperforms state-of-the-art techniques to select tile and pad sizes for all analyzed kernels, for all platforms. We have shown how our cost model selects tile and pad factors in concert, which translates to consistent speedups.

From these results, we conclude that accurate cost models that consider not only cache behavior but other hardware components are necessary. A simple cost model may hinder the compiler's ability to generate good code that improves overall performance. For instance, it is not clear when padding should be combined with tiling for the **lrw** and **tss** algorithms.

6.7 Related Work

Researchers working on locality optimizations have considered re-ordering techniques such as loop interchange [50, 113, 168, 170], loop fission/fusion [113] and loop tiling [24, 35, 93, 131, 168].

The success of loop tiling depends on the tile size and shape selection. Lam *et al.* [93] present an algorithm that chooses the largest non-conflicting square tile, considering caches with low associativity. Coleman and McKinley [35] try to maximize the tile size while minimizing the cross-interferences. Their cost model is based on computing the footprints of the array references. Rivera and Tseng [131] further extend the Euclidean algorithm [35] by computing tile widths using a recurrence. They realize that there may be some pathological problem sizes where tile selection does not work very well. They propose padding the first dimension of all arrays with the same pad to eliminate such cases.

Array padding can help eliminate conflict misses. Rivera and Tseng [129, 130] propose several simple heuristics that are addressed to eliminate conflicts in some particular cases. They mainly focus on conflicts that occur on every loop iteration, addressing only inter-padding for uniformly generated references (so they cannot remove conflict misses for references such as b[i][j]) and c[k][j]). On the other hand, they do not use intra-padding to remove cross-interferences. In the case they cannot remove all the conflicts, no changes are done to the data layout. Besides, they use the padding algorithm devised to avoid conflict misses for direct-mapped caches to remove conflict misses for set-associative caches, without taking into account that

interferences arise in different situations for different cache architectures. A set contention in a set-associative cache does not necessarily mean there is an interference. They presented an extension of this work targeting multi-level caches [132], where they study the effects of optimizing L1 cache on L2 cache behavior.

Ghosh *et al.* [52] use the CMEs to propose a tiling and padding technique. Padding works on direct-mapped caches, optimizing conflicting arrays that have the same column size. Their technique finds the optimal padding if there is a padding such that the total number of replacement misses after padding is zero. However, if such padding does not exist, their technique does not provide any solution. Note that replacement misses include both conflict and capacity misses and one may expect the case where replacement misses cannot be decreased up to zero to be common. Tiling is based on maximizing the tile size for every self-interference equation, obtaining a tile that has no conflicts for the given equation. However, they do not give insights about how to combine the different tile sizes obtained. Furthermore, tiling is not applied to cross-interferences.

Our approach has several advantages over previous research. First, our cost model describes accurately the cache behavior of any memory hierarchy and considers all array accesses within a loop nest. Moreover, we model tiling overhead due to miss-predicted branches. Second, our padding considers different pad factors for each array dimension, increasing the chances of finding a better optimized code. Finally, we perform tiling and padding at the same time, hence considering a global solution.

6.8 Chapter Summary

This chapter presents a new approach to improving execution time of programs by enhancing data locality. It combines tiling and padding to remove both capacity and conflict misses. First, using the cache locality model presented in previous chapters, we presented a very accurate cost model that describes cache locality across different levels of cache. Moreover, this cost model takes into account the possible tiling overhead due to the added miss-predicted branches. We discuss how this model can be tuned to describe accurately the performance cost for different modern architectures.

Second, we introduce the use of genetic algorithms to explore the solution space. We show how our approach can guide compiler optimizations efficiently; with a small compile-time overhead (average of 4.9 seconds per kernel), we obtain significant runtime improvements. Our results show that, compared to the best performing scheme among previous approaches for each particular architecture (which happens not to

be always the same scheme), we have 7.7%, 63.2%, 5.2% and 35.7% average speedup for Pentium-4, Alpha-21264, UltraSparc-III and Itanium processors respectively.

Overall, we contribute a new technique that makes a case for the use of accurate models to guide compilers in order to improve execution time. Moreover, it does not only model cache behavior but hardware components such as branch predictors, which shows the possibility of having complex and accurate models for modern architectures.

7

Timing Cache Behavior

CHAPTER 7

TIMING CACHE BEHAVIOR

A safe and tight worst-case behavior of programs when executed on a system with data caches is difficult to obtain due to intra- and inter-task cache conflicts. This thesis contributes a new technique to obtain predictability in preemptive multitasking systems in the presence of data caches. Our method explores the use of *cache partitioning*, *dynamic cache locking* and *static cache analysis* to provide worst-case performance estimates in a safe and tight way. We use cache partitioning, which divides the cache among tasks to eliminate inter-task cache interferences. We combine static cache analysis and cache locking mechanisms to ensure that all intra-task conflicts, and consequently, memory access times, are exactly predictable. To minimize the performance degradation due to cache partitioning and locking, two strategies are employed. First, the cache is loaded with data likely to be accessed so that their cache utilization is maximized. Second, compiler optimizations such as tiling and padding are applied in order to reduce cache conflict misses.

7.1 Motivation

The presence of data-dependent accesses makes the estimate of whether the reuse is realized or not infeasible. A naïve way of having a safe approach is to assume all future accesses as misses. However, this will result in a large overestimation of the WCET.

Another approach to overcoming this problem in real-time systems is to lock the cache for the whole execution of the program. Unfortunately, this translates to very poor cache utilization when data do not fit the cache. In order to confirm this intuition, we have run a number of different programs (for a detailed description of the benchmarks, see Table 4.4 in Section 4.4) assuming two cache configurations:

(i) Normal behavior with enabled cache (*Unlocked*).

(ii) Loading the cache with most frequently accessed memory lines and locking it

Program	Analysis	Miss Ratios				Cycles
		MIN	MAX	AVG	Increase(%)	Degradation(%)
MM	Unlocked	1.88	33.53	10.01	721.77	599.55
	Locked+Load	59.14	99.36	82.27		
CNT	Unlocked	5.67	8.33	7.94	710.92	565.67
	Locked+Load	18.08	98.72	64.45		
ST	Unlocked	3.57	14.29	7.66	389.87	307.87
	Locked+Load	3.57	96.80	35.87		
SQRT	Unlocked	1.43	1.43	1.43	0.00	0.00
	Locked+Load	1.43	1.43	1.43		
FIB	Unlocked	0.49	0.49	0.49	0.00	0.00
	Locked+Load	0.49	0.49	0.49		
SRT	Unlocked	8.37	16.74	10.93	69.16	58.28
	Locked+Load	8.37	93.73	18.49		
NDES	Unlocked	0.90	1.74	0.96	38.40	12.30
	Locked+Load	0.90	6.56	1.33		
FFT	Unlocked	0.59	56.10	9.18	119.37	97.66
	Locked+Load	0.59	93.64	20.15		

Table 7.1: Comparison of performance between an unlocked cache and a locked cache loaded with the most frequently accessed lines for programs in Table 4.4. *Increase* (*Degradation*) represents the average increase in miss ratios (cycles) of "Locked+Load" over "Unlocked" across all cache architectures.

at start time for the whole execution (*Locked+Load*).

In order to obtain the most frequently accessed lines, we have run the program once and collected statistics for each memory line accessed. Then, we load each cache set with the most frequently accessed memory lines that map to that particular set.

We have analyzed the following cache architectures: 4KB, 8KB, 16KB and 32KB (32B per line) for three different associativities (direct-mapped, 2-way and 4-way). We have also simulated the microSPARC I cache architecture (direct-mapped, 512 bytes, 32B per line). We present results accounting only for load/store instructions: we assume a conservative architecture where a cache hit takes 1 cycle and a cache miss 50 cycles.

Table 7.1 shows the results of our experiments. For each program and configuration, we report the minimum, maximum and average miss ratios across all architectures. The last two columns show to what extent locking the cache degrades performance. We present results for miss ratios and simulated cycles. We can see that performance drops significantly for most programs (in some cases, it degrades more than 500% in cycles). Cache locking performs well only in those cases where

all data fit the cache such as SQRT (it only accesses a few floating point values) or SRT (when the cache is large enough to hold the array being sorted).

7.2 A WCET Tool Overview

A real-time system is a computer-based system where the *timing* of a computation is as important as the actual *value*. In most cases, steady and predictable behavior is the desired property; sometimes *too fast* is as bad as *too slow*.

Hard real-time systems are those where a failure to meet a deadline can be fatal. To guarantee their behavior, the worst case behavior has to be analyzed. It will be used to ensure timely responses from tasks as well as input to scheduling algorithms.

That implies that it is necessary to know the execution time for the tasks in a real-time system. However, a task does not have a unique execution time. There are two sources of execution time variation: (i) the task may have different work loads depending on the input, and (ii) the initial state of the hardware where the task is executed may change for different runs. Since execution time varies, the WCET (i.e., the longest execution time for a program for all possible input data) is used as a safe upper limit.

7.2.1 Estimating WCET

In order to be useful, WCET has to be conservative and tight. On one hand, an underestimated WCET will produce a system that can fail. On the other hand, a very large overestimated WCET may be perfectly useless, since it will imply a very large waste of resources.

A naïve approach to computing the WCET of a task would be to run the program for each possible input. However, this is not possible in practice due to measurement time. Running the program with the input data that cause the WCET would be a solution, but it is usually hard to know such data. A third option is running the code with an estimated very *bad* input data. Then a safety margin is added.

However, the WCET that is obtained in each of these approaches is not safe, since it cannot be proved that it is the actual WCET. Besides, hardware components like caches may give different results for two otherwise identical runs, due to, e.g., different cache states at startup. In particular, cache behavior depends on the initial state of the cache. Therefore, a sound static analysis is needed.

7.2.2 Task Model and Schedulability Analysis

We consider a set of N periodic tasks T_i, $1 \leq i \leq N$. We denote the period and worst-case execution time of task T_i by P_i and C_i, respectively.

We consider two schedulability analyses for periodic tasks, UA (utilization-based analysis) and RTA (response time analysis). For dynamic priority preemptively scheduled systems (e.g., *earliest deadline first*), the utilization condition $U \leq 1$ is necessary and sufficient, where U is defined as follows:

$$U = \sum_{i=1}^{N} \frac{C_i}{P_i} \tag{7.1}$$

For static priority preemptively scheduled systems such as *rate monotonic*, we use response time analyses [78, 145] to obtain a necessary and sufficient condition. For a task T_i, the idea is to consider all preemptions produced by higher priority tasks on an increasing window time. The fixed point of the following recurrence gives the response time R_i of task T_i:

$$\begin{aligned} R_i^0 &= C_i \\ &\vdots \\ R_i^{n+1} &= C_i + \sum_{T_j \in HP(T_i)} \left\lceil \frac{R_i^n}{P_j} \right\rceil \times C_j \end{aligned} \tag{7.2}$$

where $HP(T_i)$ is the set of tasks with higher priority than T_i. In order to check the schedulability of task T_i, one only has to compare the response time R_i with its period P_i. Task T_i is schedulable if and only if $R_i \leq P_i$.

Our approach eliminates cache penalties due to cold-starting the cache after a context switch. Thus, classical non-cache sensitive schedulability analyses should be used rather than their cache-sensitive versions, CUA [12] and CRTA [17].

7.2.3 Extended Program Model

We recall our program model from Section 3.2. Our model applies to programs consisting of subroutines, calls, arbitrarily nested loops, and assignments possibly guided by if conditionals.

The following restrictions define the scope of programs where only one path is analyzed:

- Calls are non-recursive.

- Bounds of all loops are known and affine.
- The if conditionals are analyzable at compile time.

Otherwise, when data-dependent situations arise, we will apply path merging in order to reduce the number of paths being analyzed. We assume that the maximum number of iterations of a loop and the maximum number of recursive calls are known. This can be done either by manual annotations [40] or by automatic approaches [60]. Next, we discuss the path analysis in detail.

7.2.4 Flow Analysis

In order to estimate the WCET statically, we must analyze all possible paths. For each path generated, the execution time will be estimated for a particular architecture.

Program flow analysis determines the possible paths through a program.

Definition 7.1 **(Path)** A path from u to v in the control flow graph of a program is a sequence of directed edges, $n_0, n_1, \ldots, n_k$, such that $n_0 = u$, $n_k = v$ and (n_i, n_{i+1}) is an edge in the graph.

Flow analysis yields information about which functions are called, the number of iterations of a loop, etc. Unfortunately, it is infeasible to analyze all possible paths in a program.

Approximations during computation must be selected so that path explosion is reduced: a simple loop with an if-then-else statement that iterates a hundred times generates 2^{100} possible paths. Whenever it is possible, infeasible paths are removed [40, 103] reducing the number of paths to be analyzed.

In order to further reduce the number of paths analyzed, we use a common technique known as *merging* to make the analysis more efficient. This basically consists of reducing the path explosion by merging paths in those cases where a path enumeration is needed [45, 64, 104]. This includes data-dependent conditionals, loops with multiple paths inside and loops with unknown loop bounds. Figure 7.1 introduces three examples that will be used through this chapter to illustrate path merging.

However, this approximation trades performance for accuracy. At every merge point, the most pessimistic assumptions are made in order to have a safe estimate. In the presence of caches, this generally translates to an unknown state of the cache, since the final state of the cache for each path is also merged.

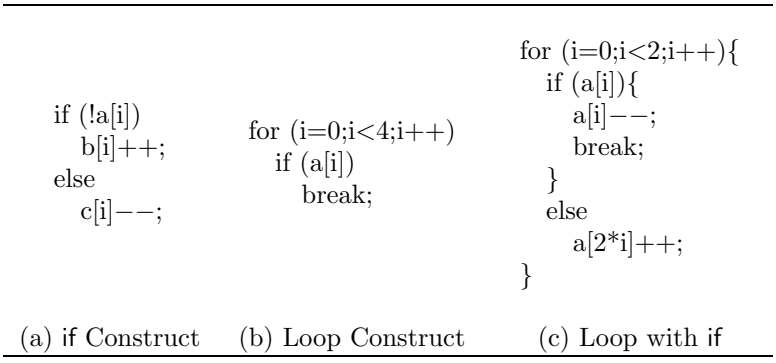


Fig. 7.1: Non-analyzable codes.

7.2.5 Merging Operator

Merge points can be chosen arbitrarily depending on accuracy and execution time desired. We use a strategy where the number of paths to be explored are reduced from n^k to kn for a loop that contains n possible paths and iterates k times. In each iteration, all n paths are analyzed, but at the start of the next iteration all these paths are merged into one. Note that we only consider natural loops, which may have multiple exits. In the other cases, path merging is not applied and thus all paths are analyzed.

Conditionals are also treated in a special way. When a conditional branch whose condition is unknown is found, both paths have to be analyzed. In order to keep the number of paths that are analyzed reasonable, both paths are merged.

Definition 7.2 **(System State)** For each path p, we denote its *system state* as the result of its (partial) execution, and it may include information about the program counter (which instruction is executed), memory locations, registers contents, etc.

Regarding our analysis, we are only concerned about the program counter (PC) and the cache state (CC).

Definition 7.3 **(Set of Cache Lines)** L is the set of cache lines.

Definition 7.4 **(Memory)** S is the set of all memory lines. To represent the absence of any block in a cache line, we introduce a new element $I \notin S$, $S' = S \cup \{I\}$.

INPUT

p_a = a path

p_b = a path

pre-condition:

$\mathsf{p}_a.state.PC = \mathsf{p}_b.state.PC$

OUTPUT

$\mathsf{p}_a \sqcup \mathsf{p}_b$ = a path

ALGORITHM

p_c is a path

$\mathsf{p}_c.state.PC := \mathsf{p}_a.state.PC;$

$\mathsf{p}_c.state.CC := \mathsf{p}_a.state.CC \sqcup_{CC} \mathsf{p}_b.state.CC ;$

$\mathsf{p}_a \sqcup \mathsf{p}_b := \mathsf{p}_c$

Fig. 7.2: Merging operator for paths.

Definition 7.5 **(CC)** The *state of the cache* is a mapping

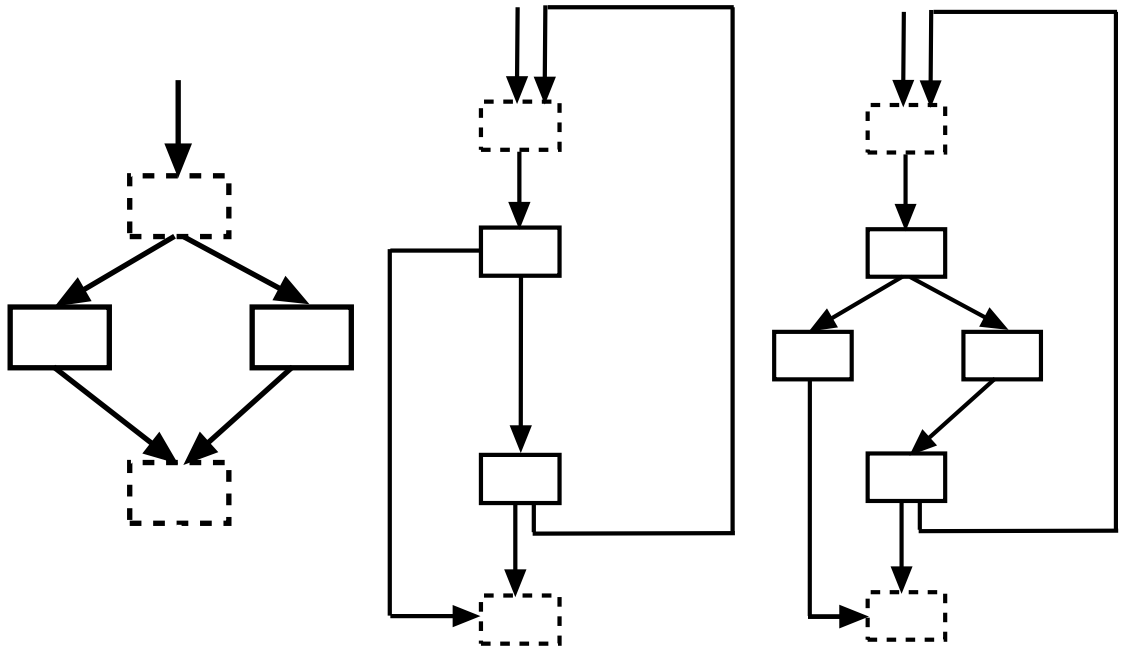
$$CC : L \rightarrow S'$$

When considering set-associative caches, we assume that cache lines are sorted within sets according to the replacement policy.[1]

For each merge operation, one must union the *system states* of all merging paths. The merging operator $\sqcup$ that merges paths is outlined in Figure 7.2. The union operation for cache states is defined as usual [5, 103]:

$$\begin{aligned} a \sqcup_{CC} b &= CC : L \rightarrow S' \\ &\qquad l \mapsto a(l) \sqcup_L b(l) \\ x \sqcup_L y &= \begin{cases} I \; if \; x \neq y \\ x \; otherwise \end{cases} \end{aligned}$$

Note that the program counter of two merging paths at the merge point is the same by definition. However, the cache states do not have to be the same, thus some approximations when merging are taken. Our approach will eliminate this situation: all merging paths have the same cache state at the merging point.



(a) if Construct (b) Loop Construct (c) Loop with if

Fig. 7.3: Control-flow graph for examples in Figure 7.1. Dashed boxes represent entry/exit nodes.

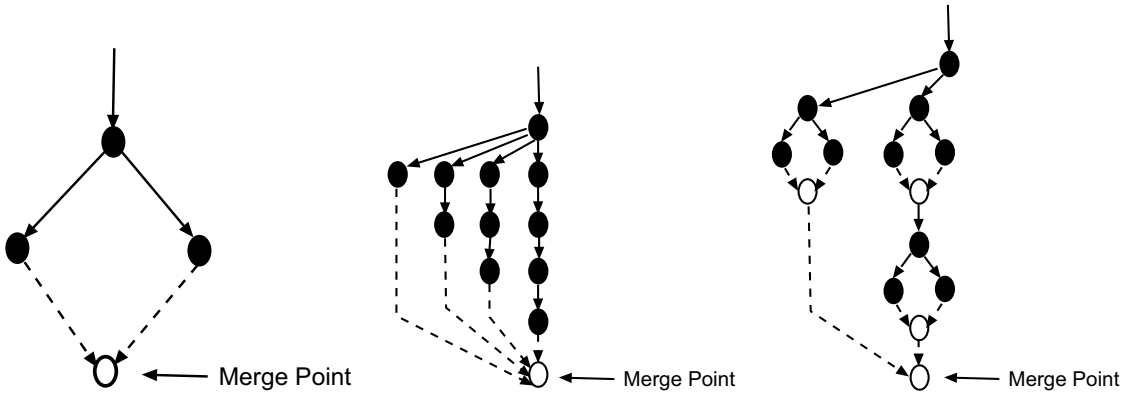
7.2.6 Merging Operator Placement

We describe the situations where the merge operator is applied below, where we use the codes in Figure 7.1. Their associated control-flow graphs are shown in Figure 7.3, where the dashed nodes represent the entry/exit nodes. The unfolded versions are shown in Figure 7.4.

Data-dependent conditionals Figure 7.4(a) shows an example of such a case. At compile time, it is impossible to figure out which branch is going to be executed. The merge point is set in such a way that it merges the outcomes from both branches.

Unknown number of iterations of a loop This situation arises when either the loop bounds are unknown or there is a jump out of the loop. Either way, a

[1]The current implementation only supports LRU replacement policy, which is the one adopted by all studied real-time processors.



(a) if Construct (b) Loop Construct (c) Loop with if

Fig. 7.4: Basic merge situations.

path is created for each possible number of iterations, and all of them merged later when they exit the loop (see Figure 7.4(b)).

Notice that these two situations can be combined. When analyzing a loop with a data-dependent conditional, we may want to merge the branches of each iteration and later, all the iterations (see Figure 7.4(c)).

7.3 Predictable Cache Behavior

When considering cache memories, schedulability analyses should consider the cost of reloading the cache lines that may have been evicted from cache. When a preempted task resumes its execution, it may spend a lot of time reloading those cache lines that have been displaced from cache. Recent studies incorporate some *cache-related* preemption costs into the schedulability analysis [12, 17, 97]. They basically consider that the preempted task will incur a miss for each cache line when resuming execution. However, this approach cannot be used when dynamic cache locking is used, since the cost of preempting a task that is accessing a locked region may be much larger than a cache miss for every cache line. A preempting task may unlock the cache and load it with its own data; when the preempted task resumes its execution, it will not reload the cache since the cache is locked. Thus, there may be more extra misses than one per cache line throughout the locked region.

Our goal is to have a method that allows obtaining an exact (we want to guarantee an exact classification of memory accesses as cache hits or misses) and safe

```
INPUT
  S   =   a set of tasks
  C   =   a cache architecture
OUTPUT
  PredictMultiTask(S, C)   =   <set of tasks, set of partitions>

ALGORITHM
  CP:= CreatePartitions(S, C);                  // set of partitions
  S_aux := ∅;                                   // set of modified tasks
  for each  task T_i ∈ S
     CP_i is T_i's cache partition
     P_aux:= LockAndLoad(T_i);                  // modified task after locking
     P_aux:= CacheOptimize(P_aux, CP_i);
     S_aux.insert(P_aux);

  PredictMultiTask(S, C) := < S_aux, CP >
```

Fig. 7.5: An algorithm for obtaining a predictable set of tasks on a multitasking system.

WCMPs of tasks for multitasking systems with data caches, so that current schedulability analyses can be applied without modifications. We propose the use of cache partitioning for avoiding inter-tasks conflicts. This allows us to compute the WCMP of each task in isolation. We combine it with some compiler cache optimizations (such as tiling and padding introduced in Chapter 6) to reduce the loss of performance due to the use of a smaller cache. When calculating the WCMP of a task, we use our static analysis combined with dynamic cache locking. We first transform the program issuing lock/unlock instructions to ensure a tight WCMP estimate at static time. In order to keep a high performance, load instructions are added only when necessary.

PredictMultiTask given in Figure 7.5 takes as input a set of tasks and a cache architecture, and generates a set of cache partitions and a set of analyzable tasks that have the same semantics as the original tasks. Then, we run our static analyzer which calculates an *exact* WCMP for each transformed task. In the next sections, we explain in detail the different parts of the algorithm. We first discuss the implications of using the cache partitioning technique. Then, we outline how we solve the problem of predictability for data caches. In order to optimize the cache behavior of tasks, we have used the different techniques explained in Chapter 6. Thus, the performance of the tasks is not jeopardized.

7.4 Cache Partitioning (CreatePartitions)

Inter-task interference occurs when memory accesses from different tasks conflict in cache (i.e., different tasks use the same cache lines and thus, a task may evict data that have been brought by another task), which causes unpredictability. Cache partitioning [88] divides the cache into disjoint partitions, which are assigned to tasks in such a way that inter-conflicts are removed.

Let $\{T_1, \ldots, T_n\}$ be a set of tasks. Usually, cache partitioning creates $n + 1$ partitions, one for each real-time task and another one which is shared among non-real-time tasks. Each task is only allowed to access its own partition, thus removing inter-task conflicts. Note that tasks that have the same priority (thus, they are non-preemptively related to each other) can share the same partition, since they are only preempted by tasks that have higher priority, and thus the predictability of cache behavior is not affected. Therefore, it is enough to divide the cache in p partitions, where p is the number of different priorities.

Cache partitioning can be implemented by either software [121, 169] or hardware [88]. Both techniques impose the partition size to be a power of two, so that the pointer transformation to access data structures can be performed in a fast way.[2] The software approach requires compiler and linker support [121], which are responsible for relocating data to provide exclusive mappings on the cache for each task.

When a cache is partitioned, each task will access a smaller fraction of the cache, which may cause capacity misses to increase. Thus, the size of the partitions has an impact on the overall performance. In order to obtain the best data cache partitioning, the decision should be taken based on the priorities and the reuse patterns of tasks. For instance, a task that has a workload of 8KB but only accesses each cache line once only needs one cache line, whereas a task with a workload of 1KB that reuses each cache line one million times would suffer a performance loss with a partition smaller than 1KB.

Our approach works with both hardware and software mechanisms, and it does not depend on the size of the partitions created. From now on we assume that the cache is divided in n equally-sized partitions, one for each task. Yet simple, we show how it is good enough for scheduling real sets of tasks and better utilizing the CPU than other approaches. An algorithm to obtain even better performance, by choosing different cache partition sizes for different tasks, is left as future work.

[2] This restriction does not apply to instruction caches.

7.5 Dynamic Cache Locking (LockAndLoad)

INPUT
P = a program

OUTPUT
$LockAndLoad(P)$ = a predictable program

ALGORITHM
P_aux is a program

$P\_aux := LockMergingPoints(P);$
$P\_aux := LockDataDependent(P\_aux);$
$P\_aux := OptimizeLock(P\_aux);$
$P\_aux := LoadData(P\_aux);$

$LockAndLoad(P) := P\_aux$

Fig. 7.6: An algorithm for obtaining a predictable program.

LockAndLoad given in Figure 7.6 takes as input a general program, and generates an analyzable program with the same semantics. In this section, we explain in detail the different parts of the algorithm to have a predictable program. We first discuss the implications of using the merging technique. Secondly, we explain how we solve the problem of predictability for data caches. Then, we outline an algorithm to selectively load the cache, so that the performance is not jeopardized. Finally, we present how this approach can be used to compute the `WCMP` and WCET of a task.

7.5.1 Path Merging (LockMergingPoints)

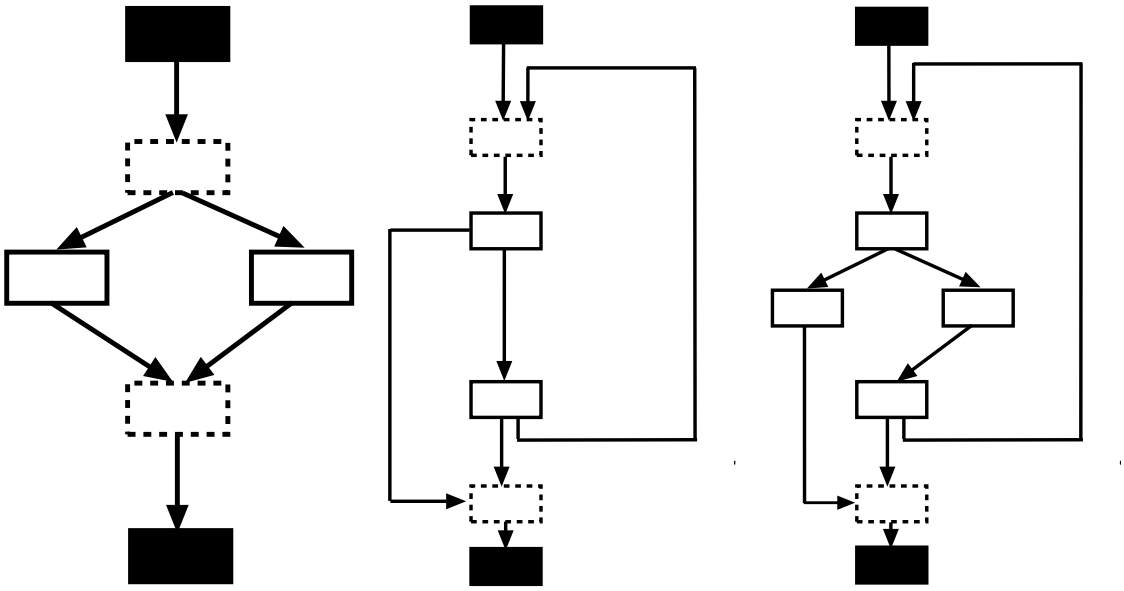
A practical limitation for WCET estimation is that the number of paths to be analyzed can easily be prohibitive, especially when studying loop constructs with multiple paths inside. We have shown in Section 7.2.4 how we manage to bound the number of analyzed paths to just a few by means of path merging. However, merging paths leads to an unknown state of the cache, since a new state of the cache is created based on the state of the cache at the end of each path [5, 104].

When a memory access is analyzed on the pipeline and it cannot be classified as a hit or a miss, both situations should be analyzed to identify the longest path [39]. This translates to larger analysis times. In order to avoid an unknown state of the cache due to merging, we lock the regions where paths are merged.

LockMergingPoints makes use of the control-flow graph of the program. It inserts

lock/unlock instructions (lock/unlock nodes in the control-flow graph) exactly for the situations described in Section 7.2.6:

- The lock instruction is placed at the top of the entry node of the if or loop construct.
- The unlock instruction is placed after the exit node of the if or loop construct. For loops with multiple exits, an unlock instruction is placed for each possible exit. Without loss of generality, from now on we only consider loops with one exit.



(a) if Construct (b) Loop Construct (c) Loop with if

Fig. 7.7: Control-flow graphs with lock/unlock nodes for examples in Figure 7.3. Black boxes represent the lock/unlock nodes with the lock/unlock instructions.

Figure 7.7 shows the resulting control-flow graphs after inserting lock/unlock instructions for the control-flow graphs shown in Figure 7.3. Figure 7.8 shows the codes with the lock instructions for the corresponding codes in Figure 7.1.

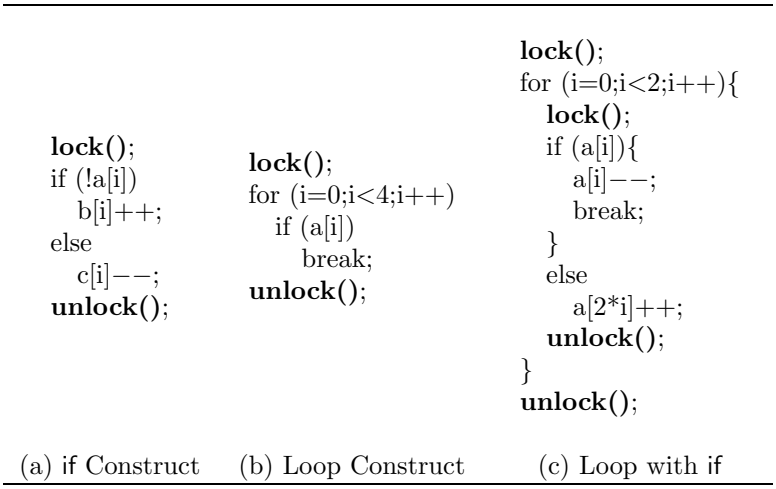


Fig. 7.8: Non-analyzable codes with lock instructions.

7.5.2 Data Cache Locking (LockDataDependent)

In the discussions so far, we have ignored the effects of data-dependent memory accesses.

Initially it may appear that obtaining a reasonable bound on the `WCMP` when the data accessed is unknown is far from being feasible. This includes indirection arrays (e.g., a[b[i]]), variables allocated dynamically (e.g., mallocs) and pointer accesses that cannot be determined statically. We also include non-linear array references that are not handled by our static analyzer (e.g., a[i*j]). However, a tight prediction of the `WCMP` can be achieved by automatically locking and loading the cache in those regions where we find those accesses. We explain how our *LockDataDependent* works for those situations in detail below.

Run-time allocation. Real-time codes are usually free from dynamic memory allocation. Otherwise, as long as the actual calls to the *malloc* routine and the size of the memory allocated are known at static time, it is possible to figure out where the *mallocs* go, just by keeping information about the allocated and deallocated memory in the program. If everything else fails, the only option is to lock the cache when accessing data allocated dynamically.

Pointers. Pointer analysis can be used to determine some pointer values [164]. Besides, programmer annotations may be used to tighten the analysis. When analyzing indirection arrays in a loop, we lock the cache for the loop nest. If the array being accessed fits the cache, we load it. Otherwise, we make sure that it is not

in the cache by invalidating those lines that contain parts of it.[3] This allows us to (i) predict the result of the memory access, and (ii) reduce the variation of the execution time, since we cannot have a hit when we have predicted a miss (and vice versa).

Library and operating system calls. In order to obtain an accurate WCMP of a task with library calls, we would need to analyze the source code of the library to generate annotations that would help our analysis. Otherwise, just to ensure that those calls do not interfere with our analysis, we lock the cache before each call statement and unlock it afterwards. The memory accesses within the library call will not be guaranteed as hits/misses, thus both situations will be analyzed in the pipeline analysis of the WCET calculation.

The lock/unlock instructions are placed following the same principles as we used in Section 7.5.1. Lock instructions are placed at the top of the control-flow node where the memory access is realized (or where the call statement is placed). The unlock instructions are placed at the bottom.

7.5.3 Optimizing Placement of Lock/Unlock Instructions (OptimizeLock)

Automatic placement of lock/unlock instructions may cause an important performance degradation. On one hand, the execution of lock/unlock instructions incurs a run-time overhead. This is especially important for instructions placed within loops, since they will execute several times. On the other hand, locking the cache when it is not necessary usually worsens performance.

Let us consider the code in Figure 7.8(c). At first sight, the lock/unlock instructions within the loop nest are unnecessary, due to the lock/unlock instructions placed at the entry/exit of the loop respectively. If we assume that each lock/unlock executes in 1 cycle, removing the unnecessary instructions will reduce the number of cycles needed to run the code by $2*k$, where k is the number of iterations. For our example, we would save 4 cycles.

OptimizeLock goes through the control-flow graph looking for redundant lock/unlock instructions. It is an algorithm that keeps iterating while some progress is done. We have currently implemented the following optimizations, where ";" represents arcs between nodes in the control-flow graph.

Rule 1. Lock/unlock instructions that lock the whole loop body are placed outside the loop.

[3] We can obtain this information from our static analyzer.

```
lock();
lock();
for (i=0;i<2;i++){
  if (a[i]){
    a[i]−−;
    break;
  }
  else
    a[2*i]++;
}
unlock();
unlock();
```

(a) After 1st iteration

```
lock();
for (i=0;i<2;i++){
  if (a[i]){
    a[i]−−;
    break;
  }
  else
    a[2*i]++;
}
unlock();
```

(b) After 2nd iteration

```
lock();
for (i=0;i<2;i++){
  if (a[i]){
    a[i]−−;
    break;
  }
  else
    a[2*i]++;
}
unlock();
```

(c) Final code

Fig. 7.9: Application of *OptimizeLock* on the code in Figure 7.8(c).

$$\texttt{loop;lock;S;unlock;endloop} \;\Rrightarrow\; \texttt{lock;loop;S;endloop;unlock}$$

Rule 2. Remove nested lock regions.

$$\texttt{lock;lock;S;unlock;unlock} \;\Rrightarrow\; \texttt{lock;S;unlock}$$

Rule 3. Fuse two consecutive locked regions.

$$\texttt{lock;S1;unlock;lock;S2;unlock} \;\Rrightarrow\; \texttt{lock;S1;S2;unlock}$$

Rule 4. Inspecting the memory accesses for the different outcome branches of an IF statement may allow us to detect that the memory accesses are actually the same, thus we do not have to distinguish among them.

$$\textit{if}\ \texttt{S1}.\text{memory\_accesses}=\texttt{S2}.\text{memory\_accesses}\ \textit{then}$$
$$\texttt{lock;if;then S1;else S2;unlock} \;\Rrightarrow\; \texttt{if;then S1;else S2}$$

We show in Figure 7.9 the results of running the code in Figure 7.8(c) through *OptimizeLock*. The first iteration applies Rule 1, whereas in the second iteration it uses Rule 2 to remove the innermost locked region. Finally, it stops at the third iteration since no further changes are done. The final code is showed in Figure 7.9(c).

We define two extra rules to optimize lock/unlock instructions placement:

Rule 5. Move a statement past a lock instruction.

```
S1;lock;S2;unlock  ⇛  lock;S1;S2;unlock
```

Rule 6. Move an unlock instruction past a statement.

```
lock;S1;unlock;S2  ⇛  lock;S1;S2;unlock
```

Whereas Rules 1–4 do not modify cache behavior, these last two rules may not always be beneficial. If data accessed in the newly locked statements is already in cache then these transformations do not hurt performance. However, they may create later opportunities for other optimizations. The automatic placement of lock/unlock instructions to get the best performance remains as future work.[4]

7.5.4 Selecting Data to Lock in the Cache (LoadData)

The benefit of cache locking is clear from the predictability point of view. Locking the cache allows us to analyze data-dependent constructs while not jeopardizing the analysis of the forthcoming code. However, programs do not benefit from locality as we have shown in Table 7.1.

In order to overcome this problem, we can load the cache with data likely to be accessed. Nevertheless, determining accurately which data in the cache gives best performance is too expensive; it would be the same as knowing, before running the program, the most frequently accessed memory lines for each cache set. However, we can use a simple analysis based on reuse analysis to determine which data to load, if any.

Figure 7.10 outlines *LoadData*, the algorithm we use to load the cache selectively. The for in line 3 analyzes all locked regions sequentially. For each locked region, it collects all variables that are accessed within it, classifying the variables depending on whether they have data-dependent (non-analyzable) accesses or not (analyzable).

Loading Analyzable Variables

We first study those variables whose memory accesses are statically analyzable. Since we want to maximize the locality, *IssueInstrForAV* in Figure 7.11 analyzes variables in descending order; the variable that has more memory references is going to be

[4]This is equivalent to optimize the conversion of data between two formats which is known to be NP-hard.

INPUT
P = a program [with locked regions]

OUTPUT
$LoadData(P)$ = program with selective load instructions

ALGORITHM
1 $P\_aux := P$ is a program
2 $Lock(P)$:= locked regions of P;
3 for each locked region $L \in Lock(P)$
4 $NAV(L)$:= non-analyzable variables $\in L$;
5 $AV(L)$:= analyzable variables $\in L$;
6 $P\_aux := IssueInstrForAV(P\_aux, AV(L))$;
7 $P\_aux := IssueInstrForNAV\ (P\_aux, NAV(L))$;
8 $LoadData(P) := P\_aux$

Fig. 7.10: Algorithm for selective loading.

allocated first. In order to decide which memory lines to load, we compute, for each variable, the range of addresses that it accesses. When analyzing array variables, we use the concept of *uniformly generated references (UGR)* [50] to decide which part of the array is accessed within the region. At line 3 we classify all memory references to the studied variable V in uniformly generated classes, that is, it groups references whose array subscript expressions differ at most in the constant term.

We estimate the amount of data that can be reused from outside the locked region using the reuse vectors. The for in line 4 studies all UGR classes, again in descending order, giving priority to those that have more references. Our algorithm is a simple volume analysis based on reuse vectors (line 5). It is a modified version of those proposed previously [134, 168] in order to handle locked regions.

If we detect that some elements will not be in cache when we lock the cache, *AddLoads* includes the necessary load instructions to place them in cache. For large data sets, we may try to load a memory line that maps to a cache set that is already full. In those cases, we do not reload it since it has been loaded by a variable with higher locality.

Loading Non-Analyzable Variables

A precise approach to computing WCET uses global information (such as cache behavior) as input to the local low-level analysis, simulating the result of the cache

```
INPUT
  P   =   a program
  S   =   set of variables
OUTPUT
  IssueInstrForAv(P,V)   =   program with selective
                             load instructions
ALGORITHM
1   P_aux := P is a program
2   for each variable V ∈ S (in desc. order)
3       UGR(V) := classify references ∈ V in UGR classes;
4       for each reference class R ∈ UGR(V) (in desc. order)
5           if (R has no locality)
6               P_aux := AddLoads(P_aux,R);
7   IssueInstrForAv(P,V)  :=  P_aux
```

Fig. 7.11: Algorithm for issuing load instructions for analyzable variables.

miss/hit on the actual execution of instructions in the processor pipeline. However, when the result of a cache access is unknown, both possible results have to be simulated.

Our *IssueInstrForNAV* analyzes those variables that have non-analyzable accesses. Since it is not possible to determine at compile time which part of the array is accessed, we assume that the whole array is accessed. Besides, since we want to avoid those situations where a cache access cannot be classified, we load the whole array if there is space in the cache. Otherwise we remove all elements present in cache, thus ensuring cache misses for all accesses to that array. This algorithm is illustrated in Figure 7.12. It first makes room in the cache invalidating all data from variables that do not fit in cache (lines 2–4). Then, it tries to load into cache those variables that do fit in cache.

Architectural Support

Several processors offer the ability to load and invalidate cache lines selectively, with *cache fill* and *invalidate* instructions respectively. Thus, no special hardware support is necessary to implement our *LoadData* algorithm.

However, both of them could be "simulated" in software if necessary, even though at the cost of worse performance.

```
INPUT
  P  =  a program
  S  =  set of variables
OUTPUT
  IssueInstrForNAV(P,V)  =  program with selective
                            load instructions
ALGORITHM
1  P_aux := P is a program
2  for each variable V ∈ S (in desc. order)
3       if (V does not fit in cache)
4            P_aux := InvalidateArray(P_aux,V);
5  for each variable V ∈ S (in desc. order)
6       if (V fits in cache)
7            P_aux := LoadArray(P_aux,V);
8  IssueInstrForNAV(P,V) := P_aux
```

Fig. 7.12: Algorithm for issuing load instructions for non-analyzable variables.

Remarks

In the discussions so far, we have ignored the effects of possible conflicts with memory accesses coming after the locked region. It may happen that due to the added load instructions, a memory line that otherwise would have been reused later is flushed out from cache, thus worsening cache behavior for that particular access. This would cause, in the worst case, one miss per each cache line. However, keeping those lines could cause a poor performance for the locked region. Achieving the best overall performance (i.e., deciding which memory lines to load taking into account the whole program) is a challenging problem that we plan to address in the future.

7.5.5 Putting It All Together

In this subsection, we will use the code in Figure 7.13(a) to illustrate how *LockDataDependent* and *LoadData* work. We assume, for this example, a 4KB direct-mapped cache with 16B per line.

We start running *LockDataDependent*, which detects those constructs that are not analyzable at compile time and places lock/unlock instructions. After running *OptimizeLock* to avoid unnecessary locks/unlocks at every iteration, we obtain the code shown In Figure 7.13(b).

The next step consists in running *LoadData* to decide which data to load. We

```
int a[100], b[100];
int c[100], k=0;
for (i=0;i<100;i++)
   a[i]=random(i);
for (i=0;i<100;i++)
   c[i]=b[a[i]]+c[i];
for (i=0;i<100;i++)
   if (c[i]>15)
      k++;
   c[i]=0;
```

Non-analyzable constructs:
b[a[i]]
c[i]>15

(a) Original Code

```
int a[100], b[100];
int c[100], k=0;
for (i=0;i<100;i++)
   a[i]=random(i);
lock(); /*region 1*/
for (i=0;i<100;i++)
   c[i]=b[a[i]]+c[i];
unlock();
for (i=0;i<100;i++){
   register int temp=(c[i]>15);
   lock();/*region 2*/
   if (temp)
      k++;
   unlock();
   c[i]=0;
}
```

(b) Code with lock/unlock

Fig. 7.13: A code before and after applying *LockDataDependent* and *OptimizeLock*.

summarize the steps applied when analyzing the first locked region in Figure 7.14.

We start studying the analyzable variables for each region. For the first region, it identifies two analyzable variables, c and a. It first analyzes c since it has two references, and then a, determining that c is not in cache yet. Thus, it issues the corresponding load instructions. Using the reuse vectors, we detect temporal locality between the two occurrences of *a[i]*, and the volume analysis says that neither access will flush the datum accessed out from the cache. A similar analysis is performed for the second region, determining that k is already in cache due to the initialization.

Then, the non-analyzable variables for each locked region are analyzed. Our approach identifies variable b in the first locked region, which is accessed for the first time. Assuming that there are no interferences, there is enough space in the cache to load b.

Eventually, the worst-case memory performance will be computed. Figure 7.15 shows the final transformed code. With the information of when a memory access is to be a miss/hit, we compute that the longest path is the one where *c[i]>15* holds in all instances. It results in 26 misses due to first accesses to k and a, 50 misses due to the loading of b and c and 775 hits. In case that array b did not fit the cache, we would estimate all its accesses as a miss, since we would not know the memory

Anal. Vars	Refs	Locality	Load
c:	*c[i]*, *c[i]*	N/A	YES
a:	*a[i]*	YES	NO

⇓ UGR classes for c ⇓
c[i]

⇓ c has not locality ⇓
Issue loads for *c[i]*, $i = 0 \ldots 99$

⇓ UGR classes for a ⇓
a[i]

⇓ a has locality ⇓
Do not issue loads

Non-Anal. Vars	Refs	Fits
b:	*b[...]*	YES

⇓ b fits the cache ⇓
Issue loads for b, $i = 0 \ldots 99$

Fig. 7.14: Detailed steps for the *LoadData* execution for region 1.

lines being accessed (besides, we would have invalidated array b since our analyzer would not take advantage of it).

7.6 Experimental Results

We have conducted experiments for data caches commonly used in real-time systems. The cache configurations and access times are the ones specified in Table 4.2. Otherwise, we make clear which cache architecture is used. Each instruction to load the cache is treated as a normal memory access. We present results in terms of `WCMP`.

Figure 7.16 depicts the framework used to compute the worst-case performance and study the schedulability of a task set. The paths that are used to obtain the path corresponding to the worst-case scenario are currently manually fed to our system.

The central component is the static analyzer. Since we want to guarantee hits and misses for all memory accesses, we have used our *FindMisses* algorithm (see Section 5.5.2) to obtain precisely which memory accesses result in a miss. We present the performance of our approach for two real task sets. We set up a **large** task set in order to evaluate the efficiency of cache partitioning and compiler optimizations. The **medium** task set is used to show that even for smaller workloads, our approach performs better than static cache locking. An overview of the two task sets can be

```
int a[100], b[100];
int c[100], k=0;

for (i=0;i<100;i++)                lock(); /*region 1*/
   a[i]=99-i;                      for (i=0;i<100;i++)
load(c[0]);                          c[i]=b[a[i]]+c[i];
load(c[4]);                        unlock();
⋮                                  for (i=0;i<100;i++){
load(c[92]);                         register int temp=(c[i]>15);
load(c[96]);                         lock();/*region 2*/
load(b[0]);                          if (temp)
load(b[4]);                             k++;
⋮                                    unlock();
load(b[92]);                         c[i]=0;
load(b[96]);                       }
```

Fig. 7.15: Transformed code with lock/unlock and load instructions.

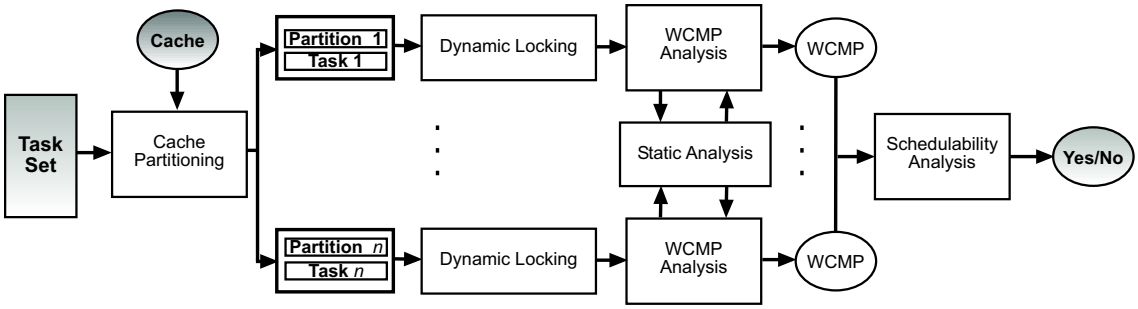


Fig. 7.16: A framework for worst-case performance computation.

seen in Table 7.2. The programs are introduced in Table 4.4. For each task, we present its name, its description, and the WCMP when the data cache is disabled. We give two possible periods. The *normal* periods of tasks have been selected so that the relation between CPU utilization[5] and amount of data is the same for each task set. We chose a CPU utilization of 2.03 for the **large** task set, and 4.69 for the **medium**. For the *HP* (high performance) periods, we chose them so the CPU

[5]In terms of our simple timing model.

Name	Workload (bytes)	WCMP (no cache)	Period (Normal)	Period (HP)
Large Task Set				
MM	120000	153140000	117800000	102093333
SRT	8000	113925998	159496397	227851996
FIB	16	7790	155800000	3895
FFT	8192	1655808	152334336	3311616
Medium Task Set				
CNT	40000	1140000	570000	285000
SQRT	16	5360	241200	2680
ST	16000	532000	266000	266000
NDES	960	220938	331407	110469

Table 7.2: Task Sets used.

utilization is 4.5 and 10.0 for the **large** and **medium** task set respectively, so that tasks have higher throughput. Thus, the task sets are not feasible if a data cache is not used for any of the period configurations.

We start analyzing dynamic cache locking. We first evaluate the accuracy of our static data cache analysis when adding the locking features. Then, we analyze the efficiency of our loading algorithm for reducing the performance degradation due to the lock/unlock instructions. Next, we present our estimated WCMP for the set of benchmarks for different architectures.

As a second step, we discuss the impact of partitioning the cache on the system's throughput. Later, we compare the performance of different methods that ensure predictability when applied to partitioned caches. Finally, we show the worst-case performance when our method is applied, and compare it with static data cache locking [125].

7.6.1 Accuracy of FindMisses

The results of our first set of experiments are shown in Table 7.3. Table 7.3(a) shows the accuracy of our method for those codes where locking was not necessary. All the programs consist of a set of subroutines, some of them containing IF statements. In all the cases, we predict exactly the same results as yielded by the simulator. Moreover, we exactly predict for each memory access the actual behavior.

Table 7.3(b) presents the results for those codes where our method issued lock and unlock instructions. In order to show our capability to statically analyze the cache behavior in this situation, we analyze the same path that is actually executed

Name	MP.	Simulated Cost	Estimated Cost	Est/Sim Ratio
MM	S	7108684	7108684	1.00
	P	8836226	8836226	1.00
	M	8513410	8513410	1.00
	I	8701615	8701615	1.00
CNT	S	75000	75000	1.00
	P	122500	122500	1.00
	M	105000	105000	1.00
	I	105192	105192	1.00
FIB	S	223	223	1.00
	P	279	279	1.00
	M	220	220	1.00
	I	244	244	1.00
ST	S	31784	31784	1.00
	P	32500	32500	1.00
	M	29000	29000	1.00
	I	33500	33500	1.00

(a) Codes where cache is not locked.

Name	MP.	Simulated Cost	Estimated Cost	Est/Sim Ratio
SQRT	S	332	332	1.00
	P	883	883	1.00
	M	458	458	1.00
	I	962	962	1.00
SRT	S	7509	7509	1.00
	P	12287	12287	1.00
	M	10515	10515	1.00
	I	12789	12789	1.00
NDES	S	9040	9040	1.00
	P	9450	9450	1.00
	M	9841	9841	1.00
	I	9514	9514	1.00
FFT	S	233344	233344	1.00
	P	807936	807936	1.00
	M	358400	358400	1.00
	I	849152	849152	1.00

(b) Codes with lock/unlock instructions.

Table 7.3: Dynamic results for data caching. *S* stands for microSPARC-IIep, *P* for PowerPC 604e, *M* for MIPS R4000, *I* for IDT 79RC64574.

in the simulator. Thus, we can isolate the results of our analyzer from those of the `WCMP` computation. *FindMisses* obtains the same results as the simulator in all cases.

Analysis time

We run our approach on a Pentium-4 processor at 1.6GHz. The average execution time needed to analyze each configuration was 0.6 seconds. MM was the program that took the longest time to analyze, with 3 seconds for each cache configuration. The problem size is `N=100`, which means that we have to evaluate around 3 million accesses. We believe this time is reasonable for the kind of analysis performed.

7.6.2 Performance of Data Cache Locking

The goal of using data cache locking is to eliminate unpredictability by locking those regions in the code where a static analyzer cannot be applied. However, cache locking may cause degradation in performance, which we try to avoid by means of loading the cache with data likely to be accessed. To evaluate the effectiveness of this approach, we compare the memory cost of the resulting code with lock/unlock instructions against the same code extended with selective load instructions. For the sake of comparison, we do not consider the additional cycles due to extra loads and locks/unlocks. In order to isolate the results from those of the `WCMP` computation, we consider the actual path that is executed (i.e., we do not use the longest path).

The results of this experiment are shown in Table 7.4. We only analyze programs where lock/unlock and load instructions were issued. We can see that in the general case, locking the cache without loading it leads to a significant performance degradation, in one case as large as over 1000%. SRT presents almost the same performance regardless the presence of lock/unlock instructions because when the lock regions are executed, all data are already in cache. The same happens to NDES; however, when considering a small cache like the one in microSPARC-IIep, it cannot hold all data and thus the performance drops.

When loading the cache, performance degradation is usually eliminated. In those cases where there are conflicts among data accessed in the locked regions, loading the cache reduces the performance degradation, but it cannot eliminate it completely. Finally, the last column presents the number of extra loads issued to load the cache. It shows that the reduction of memory cost can be achieved with few selected loads.

We have evaluated the overall overhead of the resulting code in more detail.

Name	C.	Unlock	Lock	Lock & Load	$\Delta_U(\%)$	$\Delta_L(\%)$	#Loads
SQRT	S	158	330	158	108.8	0.0	1
	P	214	881	214	311.6	0.0	1
	M	185	456	185	146.4	0.0	1
	I	218	960	218	340.3	0.0	1
SRT	S	7507	7507	7507	0.0	0.0	0
	P	12285	12285	12285	0.0	0.0	0
	M	10513	10513	10513	0.0	0.0	0
	I	12787	12787	12787	0.0	0.0	0
NDES	S	6299	6992	6992	11.0	11.0	0
	P	6970	6970	6970	0.0	0.0	0
	M	6641	7361	7361	10.8	10.8	0
	I	7034	7034	7034	0.0	0.0	0
FFT	S	88696	231296	118544	160.7	33.6	256
	P	52736	805888	52736	1428.1	0.0	128
	M	50944	356480	50944	599.7	0.0	256
	I	53248	847104	53248	1490.8	0.0	128

Table 7.4: Memory cost in cycles for the lock & load algorithm. *S* stands for microSPARC-IIep, *P* for PowerPC 604e, *M* for MIPS R4000, *I* for IDT 79RC64574. (Δ_U=loss of performance without loading the cache, Δ_L=loss of performance when loading the cache).

Figure 7.17 contains the results where cycles due to locks/unlocks and extra loads are considered. The memory cost is normalized to the memory cost of the actual execution of the program without lock instructions. We can see that the slowdown ranges from 0% to 43%, mainly because the cache is not big enough to contain all data accessed in the locked regions. For instance, FFT has an overhead of 43% for the microSPARC-IIep architecture. When the cache size is increased, the conflicts disappear and the overhead is minimal.

NDES deserves special comments. The overhead is basically due to the lock/unlock instructions. This happens because a majority of these instructions are nested in loops, and they cannot be removed by applying Rules 1–4 explained in Section 7.5.3. Thus, we have decided to apply Rules 5–6. Figure 7.18 shows the overhead for NDES after optimizing the placement of lock/unlock instructions. We can see that these transformations have worked as *enablers*, i.e., they have enabled Rules 1–4 to achieve a better instruction placement. Due to changes in the lock/unlock instructions placement, our algorithm has now issued 12 load instructions for microSPARC-IIep and MIPS R4000, and 22 for the remaining PowerPC 604e and IDT 79RC64574.

In the following section, we show how this small degradation in performance allows having a fully predictable program. Thus, we can compute the `WCMP` in a

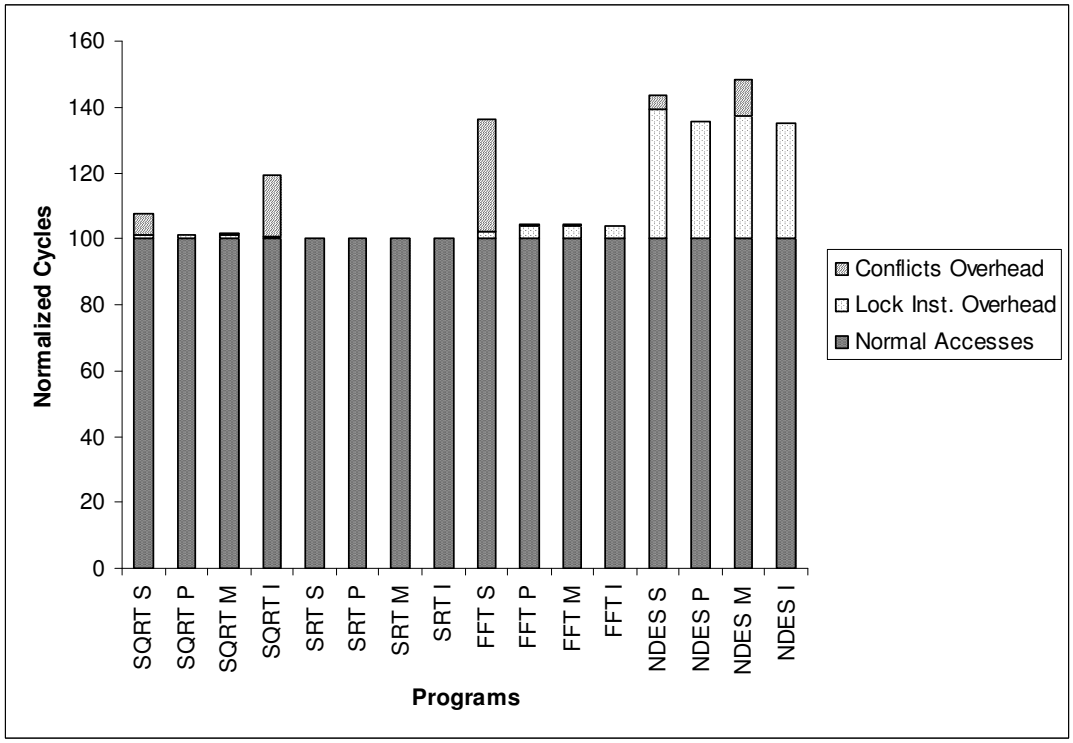


Fig. 7.17: Overall overhead of the cache locking. *S* stands for microSPARC-IIep, *P* for PowerPC 604e, *M* for MIPS R4000, *I* for IDT 79RC64574.

much tighter way than previous approaches. Even though the actual execution time of the task may increase, the WCMP will be smaller, thus we will be able to make better use of resources.

7.6.3 WCMP

Our locking algorithm will be successful if the presence of locked regions allows us to compute a smaller WCMP than before. This is,

$$if\ \texttt{WCMP}(task + lock + load) < \texttt{WCMP}(task)$$

In order to see the effectiveness of our approach, we have compared our method to compute WCMP with two other methods that are currently used:

- Cache disabled (i.e., cache locked all the time).

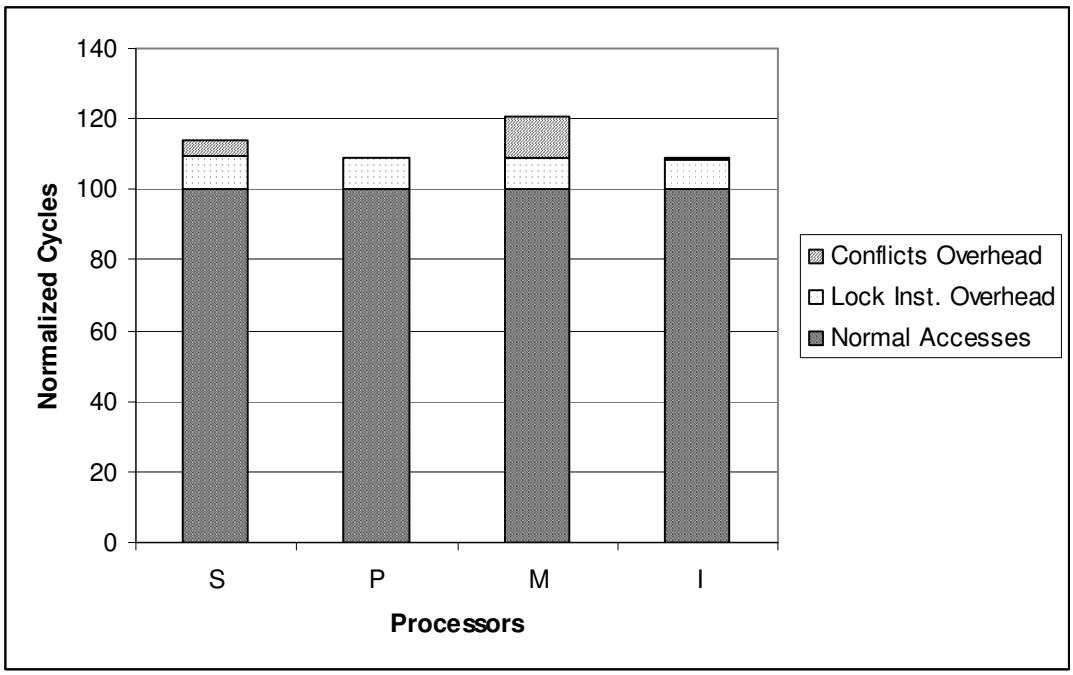


Fig. 7.18: Overhead of cache locking for the NDES program after further optimizing lock/unlock instructions placement. *S* stands for microSPARC-IIep, *P* for PowerPC 604e, *M* for MIPS R4000, *I* for IDT 79RC64574.

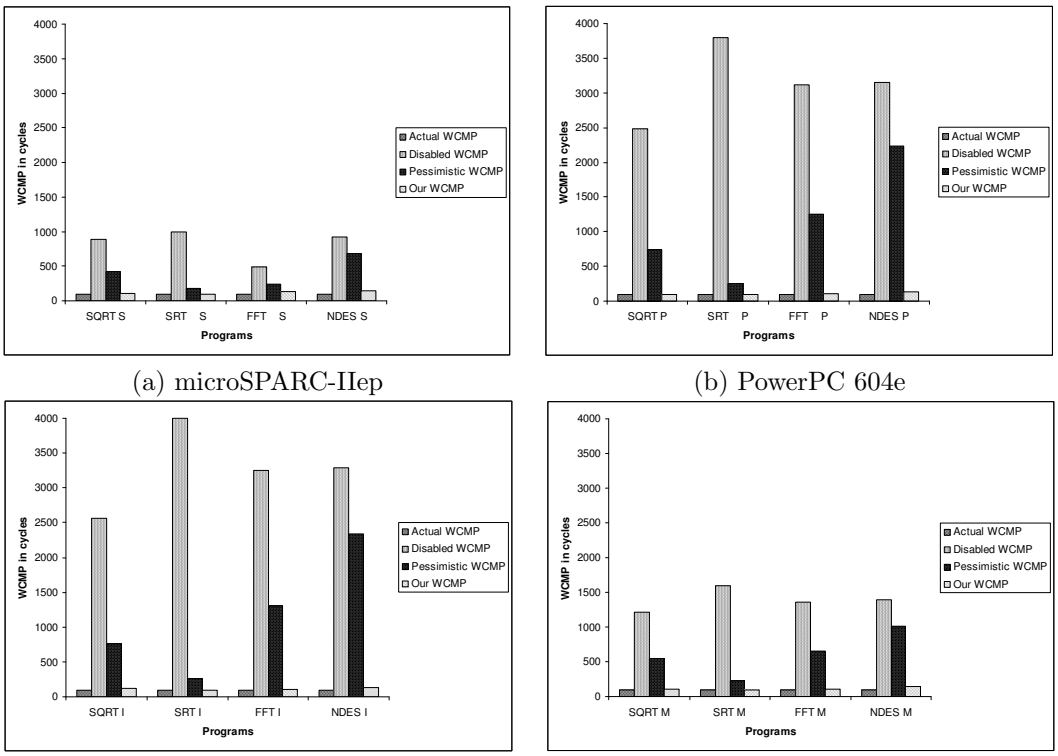
- Cache unlocked, making pessimistic assumptions whenever we do not know what happens. This can be seen as considering an empty cache where we would unlock the cache in our approach.

We use as a reference the actual `WCMP` of the program without lock instructions, which is obtained running the program with the worst-case input data.

Figure 7.19 shows the different estimates for each method. When we consider the cache disabled, all memory accesses are considered as misses, which yields a very large overestimation of the `WCMP`. The values show that the estimated `WCMP` is between 5 and 38 times larger than the actual one.

The pessimistic approach performs better than considering the cache disabled, but it is still far from a tight `WCMP`. The estimated `WCMP` is between 2 and 22 times larger than the actual `WCMP`. Our approach gives an exact `WCMP` of the transformed program (i.e., the program with lock instructions).

Finally, Figure 7.20 shows our estimated `WCMP` for the NEC V850E/ME2 micro-



(a) microSPARC-IIep (b) PowerPC 604e

(c) IDT 79RC64574 (d) MIPS R4000

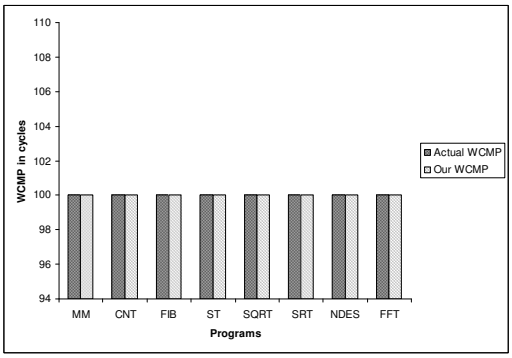
Fig. 7.19: Estimates of the WCMP.

controller. We assume that each cache miss takes 24 cycles. We use as a reference the actual WCMP of the transformed program. We can observe that for all kernels studied, our approach is exact and gives the actual WCMP.

7.6.4 Dynamic Locking: Summary

Overall, we have shown the effectiveness of dynamic locking. Whereas some performance may be lost due to the locking mechanism (in the worst case, the program runs 0.4 times slower), we can achieve a perfect estimate of the WCMP for the benchmarks given. Besides, we have seen that the estimate of the WCMP(task+lock+load) is much smaller than the best estimate of the WCMP(task). For those programs where lock instructions are not issued, our estimate is exact and there is no overhead.

We have presented results that highlight the accuracy of our static approach. Later, we have seen that in all cases, our selective locking technique allows us to



Program	WCMP
MM	10128312
CNT	145000
FIB	228
ST	59448
SQRT	143
SRT	3011521
NDES	8827
FFT	57344

Fig. 7.20: Estimates of the WCMP for the NEC V850E/ME2 high-performance microcontroller.

accurately predict the cache behavior, which translates to an exact computation of the WCMP. We have shown that estimating the WCMP without the help of locking the cache is very hard, and it usually yields very large overestimates. Moreover, the knowledge of the memory behavior will allow us to compute tighter WCET.

7.6.5 Performance of Cache Partitioning

For analyzing the whole system, we have chosen 16KB and 32KB caches with 32B lines (like the ones of PowerPC 604e and IDT 79RC64574). For each cache, we have considered a direct-mapped cache, 2-way and 4-way set associative caches.[6] We chose the hit and miss access times after the PowerPC 604e [119], where each hit takes 1 cycle and each miss 38 cycles. Lock and unlock instructions take 1 cycle each.

The goal of using cache partitioning is to eliminate unpredictability due to inter-task conflicts for multitasking systems that have data caches. However, they trade predictability for performance, which may cause some performance degradation. In order to evaluate the effectiveness of applying cache partitioning, we have compared the following three situations, where cache locking is not used:

- **Fully dynamic execution.** Each task uses the whole cache.
- **Partitioned dynamic execution.** We create equally-sized partitions. Each task runs on its own partition.
- **Cache disabled.** We consider the system without cache.

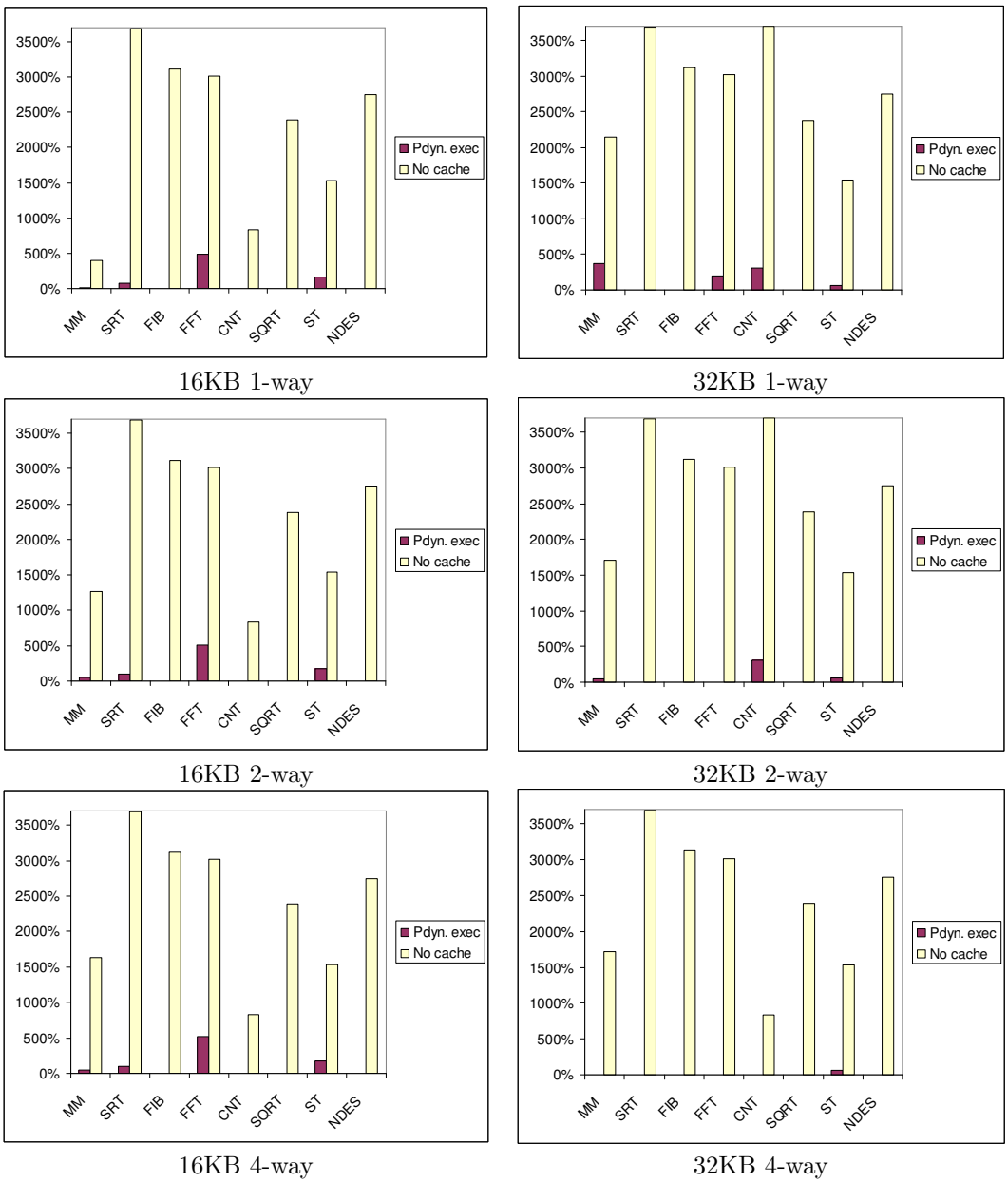


Fig. 7.21: Cache partitioning impact: comparison of performance degradation for a system with a partitioned cache and a system without a cache.

Figure 7.21 shows the results of this experiment. We present results in terms of slowdowns when compared to the memory cost of each task when fully dynamic execution is allowed. We can observe that the average memory cost increases by 79% and 2470% for partitioned dynamic execution and a system without cache respectively. This demonstrates that cache partitioning degrades performance compared to a system where each task uses the whole cache, but it is much better than not having a cache at all. Thus, we are trading performance for predictability.

7.6.6 Optimizations

The use of cache partitioning increases predictability by removing inter-task cache conflicts. However, it may increase intra-task cache conflicts since each task uses a smaller cache. This can be critical for direct-mapped caches, whereas set-associative caches can handle conflicts in a better way. In order to reduce intra-task conflicts, we apply well-known compiler cache optimizations in concert with dynamic cache locking. For the **large** task set, the application of tiling has translated to a 5.6% (1%) `WCMP` reduction for MM on the 16KB direct-mapped (2-way) partitioned cache, and padding has reduced the `WCMP` for FFT by 99.9% on the 32KB direct-mapped partitioned cache. The average memory cost compared to the partitioned dynamic execution scheme drops to 189.12% and 7.23% for the 16KB cache and 32KB cache respectively. This has allowed us to schedule successfully the **large** data set on the 16KB direct-mapped cache, whereas it has reduced the CPU utilization on the other cases.

7.6.7 Worst-Case Performance: Schedulability

In order to see the effectiveness of our approach, we have compared our method (all optimizations are on) to have a predictable multitasking system with data caches when static cache locking [22, 125] is applied. For that purpose, we have loaded the cache with the most frequently accessed memory lines[7] for each task set, and locked it for the whole execution (it is the same as *Lock-MU* in [125]). This is the best worst-case performance that can be obtained with a shared cache using static cache locking; it gives better results than applying static locking for each task independently once the cache is partitioned since tasks that use the cache intensively use more cache lines.

[6]Caches with larger associativity usually use random or FIFO replacement policies.

[7]We assume the worst-case path for each task is known.

	Large Task Set						**Medium Task Set**					
	32KB			16KB			32KB			16KB		
Ways	1	2	4	1	2	4	1	2	4	1	2	4
Lock	0.93	0.93	0.93	**1.19**	**1.19**	**1.19**	**1.51**	**1.75**	**1.74**	**2.16**	**2.19**	**2.18**
Ours	0.29	0.13	0.10	0.81	0.68	0.65	0.43	0.43	0.43	0.57	0.57	0.57

Table 7.5: Performance of static cache locking and our cache analysis.

	Large Task Set						**Medium Task Set**					
	32KB			16KB			32KB			16KB		
Ways	1	2	4	1	2	4	1	2	4	1	2	4
Lock	3.55	3.55	3.55	3.85	3.85	3.85	2.97	3.44	3.44	5.11	4.37	4.37
Ours	0.40	0.21	0.17	0.99	0.85	0.81	0.79	0.79	0.79	0.92	0.93	0.93

Table 7.6: Performance of static cache locking and our cache analysis for a high-performance system.

The worst-case system performance of both task sets is given in Table 7.5. Each cell contains the CPU utilization (if it is smaller than 1, it is schedulable for dynamic priority preemptive schedules by (7.1)). A bold number indicates that the task set is *not* schedulable according to fixed priority schedules by (7.2). We can see that our dynamic cache locking performs better than static cache locking for all cases. Even though our approach only uses a fourth of the whole cache for each task, the combination of dynamic locking and static analysis makes better use of the cache, thus reducing the `WCMP`. Static cache locking is only able to schedule (both dynamic and fixed priority systems) the **large** task set for all 32KB cache configurations. However, our approach schedules all task sets for all cache architectures. Furthermore, the CPU utilization is between 3.2 and 9.8 times smaller for the 32KB architecture, and between 1.5 and 3.8 times smaller for the 16KB cache.

7.6.8 High-Performance Systems

Finally, we show results for a high-performance multitasking system, where throughput is higher and thus the CPU utilization increases. For that purpose, we have chosen the *HP* periods in the last column of Table 7.2. Since the magnitude of the periods is very different among tasks, fixed priority systems do not perform well, and thus we only compute the CPU utilization. We can observe that our approach works better under tight deadlines, and it is able to schedule all task sets. However, static cache locking fails to schedule any of the task sets. In this case, the CPU

utilization of our method is between 3.8 and 20.0 times smaller for a 32KB cache and between 3.8 and 5.5 for the 16KB architecture. This indicates that our method scales better than static cache locking for systems that need high throughput.

7.6.9 Cache Partitioning: Summary

Overall, we have demonstrated the effectiveness of our approach. We have evaluated the impact of applying cache partitioning on a multitasking system. We have seen that even though the performance degrades, partitioning the cache is much better than not having a cache at all. Then, we have evaluated the application of static locking and dynamic locking to ensure predictability once the cache is partitioned. We have also pointed out how the application of compiler cache optimizations can be useful to reduce the performance degradation caused by the use of a small fraction of the cache. Finally, we have compared our approach with static cache locking in which all the tasks share the whole cache. We have shown that our method performs much better, and is capable of scheduling tasks that need a high throughput.

7.7 Related Work

During the last years, the real-time community has intensified the research in the area of predicting WCET of programs in presence of caches. Calculation of a tight WCET bound of a program involves difficulties that come from the very characteristics of data caching. Even though some progress has been done when studying processors with instruction caches [7, 64, 99], few steps have been done towards analyzing data caches.

We summarize below the approaches that can be used for analyzing WCET in the presence of data caches for multitasking hard real-time systems.

1. **Static Cache Analyses**. They attempt to classify statically the different memory accesses as hits or misses. However, the best static cache analyses do not consider preemptive systems and are limited to codes free of data-dependent constructs. In addition, only results for direct-mapped caches have been reported [87, 100, 102, 162].

2. **Cache-Preemption Delays**. When a task resumes its execution, it may spend a long time reloading the cache with previously loaded cache blocks. This increases the execution time of the task, and may invalidate the results of the schedulability analysis. Some studies have addressed the issue of incorporating cache preemption costs into the schedulability analysis [12, 17, 97].

However, preemption changes the cache contents in an unpredictable manner. Thus, a cache-sensitive analysis of a task assumed to run in isolation might be invalid in a context where the task is preempted: the worst-case execution path may not be the same anymore since hits may be turned into misses and vice versa. Adding a penalty by assuming the cache is cold-started might be unsafe on processors with out-of-order instruction scheduling, where a cache hit under some circumstances may be more expensive than a miss [105]. Moreover, this method resorts to a static cache analysis to obtain the WCET.

3. **Cache Locking**. The ability to lock cache contents is available on several commercial processors (PowerPC 604e [119], 405 and 440 families [73], Intel-960, some Intel x86, Motorola MPC7400 and others). Each processor implements cache locking in several ways, allowing in all cases *static locking* (the cache is loaded and locked at system start) and *dynamic locking* (the state of the cache is allowed to change during the system execution). Provided that the cache contents are known, the time required for a memory access is predictable. Cache locking can be applied to each task in isolation or at system startup [125].

4. **Cache Partitioning**. These techniques [18, 88, 101, 121] give reserved portions of the cache to certain tasks to guarantee that data will be in cache despite preemptions, thus eliminating inter-task conflicts. The reduction of the cache size that each task uses may, however, translate to a significant loss of performance.

Now, we describe the most relevant approaches in detail. Alt *et al* [5, 45] provide an estimation of WCET by means of abstract interpretation. As well as the usual drawbacks from abstract analysis (i.e., time consuming and lack of accuracy), they only analyze memory references which are scalar variables. When providing experimental results, they only deal with instruction caches. Lim *et al* [102] present a method that computes the WCET taking into account data caching. However, they only analyze static memory references (i.e., scalars), failing to study real codes with dynamic references (i.e., arrays and pointers). Kim *et al* [87] propose a method that improves the previous method extending the analysis that classifies references as either static or dynamic. However, they deal with neither arrays nor pointers (i.e., only detecting temporal locality). Further, it is limited to basic blocks, without taking into account possible reuse among different subroutines or loop nests. Li *et al.* [100] describe a method which does not merge the cache state but tries to calculate possible cache contents along with the timing of the program. The whole CPU

is modeled by a linear integer programming problem, and a new constraint is added for each element of a calculated reference. This requires a very large computation time, and has problems of scalability with large arrays. Besides, they do not report results for WCET in the presence of data caches.

White *et al* [162] propose a method for direct-mapped caches based on static simulation. They categorize static memory accesses into (i) first miss, (ii) first hit, (iii) always miss and (iv) always hit. Array accesses whose addresses can be computed at compile-time are analyzed, but they fail to describe conflicts which are always classified as misses. For instance, they overestimate the memory cost by 10% and 17% for MM and ST respectively (we estimate the `WCMP` exactly without issuing lock instructions).

Lundqvist and Stenström [104] propose an approach where variables that have non-analyzable references are mapped onto a non-cacheable memory space. They show that the majority of data structures in their benchmarks are predictable, but they have not presented the overhead of the transformed program. Neither have they reported results for WCET or `WCMP` using their approach.

Campoy *et al* [22] introduce the use of locking instruction caches for multitasking systems. They use static locking, and present a genetic algorithm in an attempt to reduce the solution space when selecting the best contents for the cache. They represent each memory block by means of one bit, which flips between 0/1 (in-cache/out-cache). On one hand, we have shown that static locking is not a good solution for data caches. On the other hand, while this approach may work for small programs, it is not easy to see how it can be extended to data caches:(i) each possible solution would occupy a lot of memory (data is typically much larger than programs), and (ii) we would need a static analysis to evaluate each potential solution. Puaut and Decotigny [125] extend it by introducing two polynomial algorithms to select the instructions to lock in cache.

7.8 Chapter Summary

Our approach combines cache partitioning and dynamic data cache locking with static cache analysis to estimate the worst-case memory performance of a multitasking system in a safe, exact and fast way.

Our method partitions the cache in equally-sized partitions, which are assigned to tasks. Cache partitioning allows us to eliminate unpredictability due to inter-task conflicts. In order to overcome the problem of data-dependent constructs, we combine it with dynamic cache locking. Finally, we run a static analysis. This results in a tool that predicts the worst-case memory performance in an *exact* and

safe way, with an acceptable loss of performance. Combined with a timing analysis platform, we may estimate a tight worst-case performance.

Overall, we contribute a new technique that provides a considerable step toward a useful worst-case execution time prediction for actual architectures. To the best of our knowledge, this is the first approach that presents a method to estimate worst-case performance for multitasking systems in the presence of set-associative data caches.

We believe this approach is highly attractive for hard-real time systems, where the problem sizes are not very big. Moreover, while being not really large, the compilation time can be amortized across the number of products shipped. We also believe that the higher throughput of the systems due to the smaller overestimation of the WCET may make this approach very useful. A better use of the cache is very useful in order to reduce power consumption and better utilize the CPU, which allows running more real-time tasks simultaneously.

8

Related Work

CHAPTER 8

RELATED WORK

In this chapter we present previous research in related areas.

8.1 Cache Analysis

Programs must exhibit sufficient locality to achieve good cache performance. Compiler optimizations for improving the cache behavior need to have detailed knowledge about the number and causes of cache misses. There are different approaches to analyze data locality which provide different trade-offs between accuracy, speed, flexibility (i.e., adaptability to different memory configurations) and information provided.

Memory simulation techniques are very accurate, flexible and can provide rich information. They are usually based on trace-driven simulation [83, 56, 116, 138, 51, 106, 58, 13, 114, 149]. However, these techniques may demand a lot of space to store traces and are very slow (typical slowdowns are several orders of magnitude). For instance, the slowdown exhibited by all simulators surveyed in [148] is in the range of 45-6250.

There are some innovative methods that have been proposed with the objective of reducing the exhibited slowdown [108, 96, 165]. However, these methods provide little information (usually only miss ratios), trading information for speed.

Martonosi *et al.* [109] introduced the use of trace sampling techniques in order to further reduce the overhead of such simulators (the slowdown shown is between 3 and 8). They take samples from the full reference trace so that they are representative of the full trace. The sizes of the samples as well as the number of samples depend on both the cache that is analyzed and the characteristics of the program being traced. Even though the results present a good degree of accuracy, neither the error bounds can be chosen, nor the sample process can be set to achieve a degree of accuracy. In addition, when sampling is applied to simulators inaccuracy can result from the unknown state of the cache at the beginning of the sample. Recently, Wunderlich

et al. [172] claim that it is not possible to have an accurate simulator that makes use of rigorous sampling without keeping an exact state of the cache.

There are other tools based on hardware counters (e.g., [6]) provided by some microprocessors. These tools are fast and accurate. They have no flexibility since they can only be used to analyze the memory architecture of the actual microprocessor. In addition, they provide a limited set of results depending on the particular counters provided by a particular machine. Information like conflict misses between two particular memory references cannot be obtained with current hardware counters.

Analytical methods use mathematical formulas to provide a characterization of a program's cache behavior so that we cannot only obtain the number of cache misses but also reason about the causes of such misses from these formulas. The ultimate goal is to develop an analytical method that can provide accurate assessments of when and why cache misses occur using a reasonable amount of computational resources (e.g., CPU time, memory and disk usage). Such a method would be useful in guiding various automatic memory optimizations and also in improving the simulation times of cache simulators and profilers.

Some static analysis techniques [141] have limited accuracy, due to unknown information at compile time. For instance, unknown loop bounds or unknown initial addresses of data structures can degrade the accuracy of the results.

A solution to this problem is to use hybrid techniques (which combine the very best from both approaches), such as SPLAT [133]. SPLAT is a static analysis technique improved with some profile (dynamic) information. This hybrid technique is fast, flexible and can provide much information like other static techniques. In addition, it is accurate because the information unknown at compile time is provided by a profiling. Due to simplifications in the analysis, SPLAT is not capable of analyzing interferences in applications with complex interference patterns and can only analyze direct mapped caches.

Porterfield [124] introduces the concept of overflow iteration for predicting the miss ratio for a fully set-associative LRU cache. Ferrante *et al.* [47] provide closed-form formulas to estimate the capacity misses of a loop nest. Temam *et al.* [141] also consider conflict misses but for a subset of array references studied in this paper. Wolf and Lam [168] propose to use vectors to describe data reuse for uniformly generated references in a perfect loop nest. They also use reuse vectors to derive an estimate of cache misses to guide their data locality algorithm. Xue and Huang [174] report an improvement. Gannon *et al* [50] and Wolfe [171] discuss the use of reference windows for predicting cache misses.

Recently, Weikle *et al* [161] introduce a trace-based idea of viewing caches as filters. Their framework can potentially handle any program consisting of any pattern

of memory references.

8.2 Compiler Optimizations

Caches improve the speed of programs by reducing the number of accesses to the slow upper levels of the memory hierarchy. Conflict misses may represent the majority of intra-nest misses and about half of all cache misses for typical programs and cache architectures [114].

Many hardware techniques have been proposed to reduce conflict misses, such as the victim cache [79] or pseudo-random placement functions [146]. Software techniques are attractive because they do not increase the hardware complexity and may be very effective for regular programs where the compiler can perform an accurate locality analysis. Moreover, they can complement hardware techniques.

Among software techniques to avoid self-interferences we can point out the tile size selection proposed by Lam *et al.* [93]. Coleman and McKinley [35] improved that technique by allowing rectangular tiles. Temam *et al.* [142] proposed to use a buffer where the data to be manipulated is copied. Unfortunately, the copy operation itself causes cache conflicts and has some overhead.

There are many other proposals to transform the order in which the iteration space is traversed, such as loop interchange, loop permutation, loop distribution, etc. [25, 84, 170]. Although these transformations may affect the number of conflict misses, they are not specially targeted to minimize them.

There has also been some work on combining loop transformations and certain data transformations, such as changing the data layout of an array from column-major to row-major [33, 81]. There are some earlier attempts on partitioning the cache and mapping arrays to distinct cache portions [26, 107]. Kodukula *el al.* [91] propose a data shackling technique to tile imperfect loop nests. This transformation is very similar to loop tiling, and as such, it does not modify the actual layouts of the arrays used. Besides, the authors do not provide an algorithm to choose the tile sizes.

Chatterjee *et al.* [28] consider nonlinear array layouts and achieve impressive performance speedups in some of the benchmarks used when combined with loop tiling. Recently, Huang *et al.* present a method that combines loop tiling and a new mapping that proves to be effective for SOR-like solvers.

8.3 Genetic Algorithms

Many researchers have applied machine-learning methods to optimize compilation process. Calder *et al.* used supervised learning techniques to fine-tune static branch prediction heuristics [20]. They employ neural networks and decision trees to search for effective static branch prediction heuristics. This scheme has also been used for determining accurately the branches' outcome at run time [77].

Monsifrot *et al.* use a classifier based on decision tree learning to determine which loops to unroll [118]. Cooper *et al.* use genetic algorithms to solve compilation phase ordering problems and to reduce code size [36]. Recently, another group presented results when applying GAs for solving the phase ordering problem [92]. The COGEN(t) compiler uses GAs to map code to irregular DSPs [59]. The compile-once nature of DSP applications may warrant the long, iterative compilation process.

Recently, Stephenson *et al.* [136] present an evolutionary algorithm to automatically find effective compiler heuristics. Instead of evolving the application, they evolve the compiler.

8.4 Path Information

The computation flow can be calculated either manually or automatically. Once the different paths are computed, these are communicated to the WCET tool which uses this information to calculate the path that generates the worst-case scenario.

Kirner and Puschner [90] add flow information (like number of iterations of a loop) into the C source code by extending the original syntax with new keywords. A different approach that does not modify the compiler is taken by Park [122], who defines a new language based on regular expressions which describes the possible paths. Li and Malik [98] and Theiling and Ferdinand [144] use linear constraints on the object code to describe possible paths.

Ermedahl and Gustafsson [40, 60] use abstract interpretation on the source code to calculate the flow information automatically. Lundqvist and Stenström [103] obtain this information using symbolic instruction-level simulation. Healy *et al.* [65, 66, 67] use data flow analysis to calculate loop bounds. Based on this information, they also try to eliminate infeasible paths. Holsti *et al.* [71] use Presburger formulas to obtain the loop bounds from object code.

9

Conclusions

CHAPTER 9

CONCLUSIONS

As a result of the increasing disparity between processor and memory speeds, the problem of hiding and reducing memory latency has become a key challenge for achieving high performance. Current processors use the memory hierarchy as a mean to bridge this gap. While caches reduce the memory latency in many cases, cache misses remain as a performance impediment for many applications.

Cache misses depend on both hardware and software. Unfortunately, cache memory behavior is very hard to predict. Simulators may be used for describing it accurately. However, they are very slow and do not provide too much insight into the causes of the misses.

This thesis addresses the question of how effective compile-time techniques that deal with cache behavior can be in practice. We have addressed this question by proposing and implementing a framework that analyzes, optimizes and times cache behavior. Even though it is applicable to general codes, its natural environment is well-structured and array-based codes.

9.1 Thesis Contributions

The key results of this dissertation are the following:

1. We have introduced a new characterization of reuse for quantifying reuse across multiple nests. If we combine it with *abstract inlining*, we obtain a method that allows analyzing whole programs. Based on these "new" reuse vectors, we have developed an analytical method for statically predicting the cache behavior of complete programs with regular computations. We outlined two algorithms for computing cache misses. *FindMisses* analyzes all iteration points and can predict exactly the cache misses for programs of small problem sizes. *EstimateMisses* analyzes a sample of all memory accesses and can achieve close to real cache miss ratios in practical cases efficiently.

2. We have shown how to analyze control flow which is analyzable at compile time. We can analyze if statements with compile-time-analyzable conditionals. In the presence of these conditionals, different references may be executed in different parts of the iteration space, which are not necessarily convex. We have described how reuse is calculated and how the miss equations are formed and solved. Our replacement miss equations are formulated and solved by taking into account the fact that the *RISs* for different references can be different.

3. A compiler can make use of this very accurate static analysis to transform codes in such a way that misses are reduced and overall performance increases. We have presented an accurate cost model that considers the effects of L1 and L2 caches, and that also considers the possible overhead due to added miss-predicted branches.

4. We have used our cost model to implement two well-known compiler optimizations: padding and tiling. We have introduced the use of genetic algorithms to explore the solution space. We have shown how our approach can guide compiler optimizations efficiently; with a small compile-time overhead (average of 4.9 seconds per kernel), we obtain very important runtime improvements.

5. Data-dependent constructs make predicting the cache behavior of a task very difficult. We have used dynamic cache locking to avoid these situations. We have discussed how to load the cache in order to achieve a good performance while having a predictable program.

6. We have combined cache partitioning and dynamic data cache locking with static cache analysis to estimate the worst-case memory performance of a multitasking system in a safe, exact and fast way. We have shown that partitioning the cache with equally-sized partitions, which are assigned to different tasks, allows eliminating unpredictability due to inter-task conflicts without compromising throughput. The use of cache partitioning and dynamic cache locking makes it possible to obtain an *exact* and *safe* worst-case memory performance.

7. Our framework combines approaches from both the high-performance and the real-time community. While being two different worlds, they share a rather large number of problems. Our framework uses cache partitioning and locking (which are mainly used in the real-time community), with a static cache analysis and loop optimizations (which are broadly used by the high-performance community).

9.2 Future Work

The main goals of our compiler framework were threefold: we wanted to analyze cache memory behavior for a broad range of programs, to present a model to optimize cache memory behavior and to have a predictable cache behavior for a large set of applications.

The scope of our static cache analysis was well-structured and array-based programs. Perhaps the most obvious extension is to address programs that make extensive use of pointers or that have very complex control flow. Having a larger set of programs analyzable statically, would allow us to reduce those situations where the cache has to be locked to ensure predictability.

Regarding code optimization, future work will both investigate the application of padding and tiling for whole programs and the addition of other compiler techniques such as loop fusion, loop interchange and unrolling. It may also be interesting to take into account the overall performance when selecting the size of the cache partitions, data to load and placement of lock/unlock instructions. We plan to investigate these issues in order to have increased predictability and better performance.

Even though the results show that a model that assumes an LRU replacement policy is good enough, we plan to incorporate other replacement policies into our model. While implementing other replacement policies may be straightforward, they may be time consuming and thus infeasible.

A

Codes – Analysis

APPENDIX A

CODES – ANALYSIS

We present the codes from different benchmarks used to evaluate the accuracy of our static cache analyzer. They consist of kernel-based applications, which implement very intensive regular computations. This set of benchmarks contains programs from Livermore, Linpack, Lapack, SPECfp95 and Perfect Benchmarks suites.

```
REAL*8 U,Z
DIMENSION U(M,M,M), Z(M,M,M)
DO 400 I3=2,M-1
  DO 200 I2=2,M-1
    DO 100 I1=2,M-1
      U(2*I1-1,2*I2-1,2*I3-1)=U(2*I1-1,2*I2-1,2*I3-1)
        +Z(I1,I2,I3)
100 CONTINUE
    DO 200 I1=2,M-1
      U(2*I1-2,2*I2-1,2*I3-1)=U(2*I1-2,2*I2-1,2*I3-1)
        +0.5D0*(Z(I1-1,I2,I3)+Z(I1,I2,I3))
200 CONTINUE
  DO 400 I2=2,M-1
    DO 300 I1=2,M-1
      U(2*I1-1,2*I2-2,2*I3-1)=U(2*I1-1,2*I2-2,2*I3-1)
        +0.5D0*(Z(I1,I2-1,I3)+Z(I1,I2,I3))
300 CONTINUE
    DO 400 I1=2,M-1
      U(2*I1-2,2*I2-2,2*I3-1)=U(2*I1-2,2*I2-2,2*I3-1)
        +0.25D0*(Z(I1-1,I2-1,I3)+Z(I1-1,I2,I3)
        +Z(I1, I2-1,I3)+Z(I1, I2,I3))
400 CONTINUE
END
```

Fig. A.1: MGRID_K.

```
 REAL*8 ZA, ZP, ZQ, ZR, ZM, ZB, ZU, ZV, ZZ
 DIMENSION ZA(JN+1,KN+1), ZP(JN+1,KN+1), ZQ(JN+1,KN+1)
 DIMENSION ZR(JN+1,KN+1), ZM(JN+1,KN+1), ZB(JN+1,KN+1)
 DIMENSION ZU(JN+1,KN+1), ZV(JN+1,KN+1), ZZ(JN+1,KN+1)
 T= 0.003700D0
 S=0.004100D0
 DO k= 2,KN
   DO j= 2,JN
     ZA(j,k)=(ZP(j-1,k+1)+ZQ(j-1,k+1)-ZP(j-1,k)-ZQ(j-1,k))*
*(ZR(j,k)+ZR(j-1,k))/(ZM(j-1,k)+ZM(j-1,k+1))
     ZB(j,k)= (ZP(j-1,k)+ZQ(j-1,k)-ZP(j,k)-ZQ(j,k))*
(ZR(j,k)+ZR(j,k-1))/(ZM(j,k)+ZM(j-1,k))
   ENDDO
 ENDDO
 DO k= 2,KN
   DO j= 2,JN
     ZU(j,k)= ZU(j,k)+S*(ZA(j,k)*(ZZ(j,k)-ZZ(j+1,k))-ZA(j-1,k)*
*(ZZ(j,k)-ZZ(j-1,k))
              -ZB(j,k)*(ZZ(j,k)-ZZ(j,k-1))+ZB(j,k+1) *(ZZ(j,k)-ZZ(j,k+1)))
     ZV(j,k)= ZV(j,k)+S*(ZA(j,k)*(ZR(j,k)-ZR(j+1,k))-ZA(j-1,k)*
*(ZR(j,k)-ZR(j-1,k))
              -ZB(j,k) *(ZR(j,k)-ZR(j,k-1))+ZB(j,k+1) *(ZR(j,k)-ZR(j,k+1)))
   ENDDO
 ENDDO
 DO k= 2,KN
   DO j= 2,JN
     ZR(j,k)= ZR(j,k)+T*ZU(j,k)
     ZZ(j,k)= ZZ(j,k)+T*ZV(j,k)
   ENDDO
 ENDDO
 END
```

Fig. A.2: HYDRO_K.

```
REAL*8 A, B, D, WB
DIMENSION A(N,N), B(N,N), D(N,N), WB(N.N)
DO J2 = 1,N,BJ
  DO K2 = 1,N,BK
    DO J=J2,J2+BJ-1
      DO K=K2,K2+BK-1
        WB(J-J2+1,K-K2+1)=B(K,J)
      ENDDO
    ENDDO
    DO I = 1,N
      DO K=K2,K2+BK-1
        RA=A(I,K)
        DO J=J2,J2+BJ-1
          D(I,J)=D(I,J)+
               WB(J-J2+1,K-K2+1)*RA
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
END
```

Fig. A.3: MMT.

```
REAL*8 X(N,N), Y(N,N), Z(N,N)
DO i = 1,N
  DO j = 1,N
    Z(j,i) = 0.0
    DO k = 1,N
      Z(j,i)=Z(j,i)+X(k,i)*Y(j,k)
    ENDDO
  ENDDO
ENDDO
END
```

Fig. A.4: MMI.

```
DOUBLE PRECISION xt, yt, xc, yc, zc
DOUBLE PRECISION zero, wsin, wcos, z, xs
DIMENSION xc(natoms, ns), yc(natoms, ns)
DIMENSION zc (natoms, ns), xt (natoms)
DIMENSION wsin(1), wcos(1), zero(1), z(1)
DIMENSION xs(1), yt (natoms)
DO i = 1, ns, 1
  xt(1) = xt(2)+wcos(1)
  xt(3) = xt(1)
  yt(2) = zero(1)
  DO j = 1, ns, 1
    yt(1) = yt(2)+wsin(1)
    yt(3) = yt(2)-wsin(1)
    z(1) = zero(1)
    DO k = 1, ns, 1
      DO l = 1, natoms, 1
        xc(l,k) = xt(l)
        yc(l,k) = yt(l)
        zc(l,k) = z(1)
      ENDDO
      z(1) = z(1)+xs(1)
    ENDDO
    yt(2) = yt(2)+xs(1)
  ENDDO
  xt(2) = xt(2)+xs(1)
ENDDO
END
```

Fig. A.5: LWSI_K.

B

Codes – Optimization

APPENDIX B

CODES – OPTIMIZATION

We present the codes from different benchmarks used to evaluate the performance of our optimization technique.

```
INTRINSIC abs, max
DOUBLE PRECISION a, aa, abx, aby, b, c, d, dd, pxx, pxy, pyy
DOUBLE PRECISION qxx, qxy, qyy, r, rx, rxm, ry, rym, x, xx, xy
DOUBLE PRECISION y, yx, yy
INTEGER*4 i, iter, j
DIMENSION aa(513, 513), d(513, 513), dd(513, 513), rx(513, 513)
DIMENSION rxm(1000), ry(513, 513), rym(1000), x(513, 513)
DIMENSION y(513, 513)
DO iter = 1, 5, 1
  DO j = 1, 511, 1
    DO i = 1, 511, 1
      xx = x(2+i, 1+j)+(-1)*x(i, 1+j)
      yx = y(2+i, 1+j)+(-1)*y(i, 1+j)
      xy = x(1+i, 2+j)+(-1)*x(1+i, j)
      yy = y(1+i, 2+j)+(-1)*y(1+i, j)
      a = 0.25D0*(xy**2+yy**2)
      b = 0.25D0*(xx**2+yx**2)
      c = 0.125D0*(xx*xy+yx*yy)
      aa(1+i, 1+j) = (-1)*b
      dd(1+i, 1+j) = 2*b+a*(2.0D0/0.98D0)
      pxx = x(i, 1+j)+x(2+i, 1+j)+(-1)*2.D0*x(1+i, 1+j)
      qxx = y(i, 1+j)+y(2+i, 1+j)+(-1)*2.D0*y(1+i, 1+j)
      pyy = x(1+i, j)+x(1+i, 2+j)+(-1)*2.D0*x(1+i, 1+j)
      qyy = y(1+i, j)+y(1+i, 2+j)+(-1)*2.D0*y(1+i, 1+j)
      pxy = x(i, j)+x(2+i, 2+j)+(-1)*x(i, 2+j)+(-1)*x(2+i, j)
      qxy = y(i, j)+y(2+i, 2+j)+(-1)*y(i, 2+j)+(-1)*y(2+i, j)
      rx(1+i, 1+j) = a*pxx+b*pyy+(-1)*c*pxy
      ry(1+i, 1+j) = a*qxx+b*qyy+(-1)*c*qxy
    ENDDO
  ENDDO
ENDDO
END
```

Fig. B.1: TOMCATV: loop number 1.

```
INTRINSIC abs, max
DOUBLE PRECISION a, aa, abx, aby, b, c, d, dd, pxx, pxy, pyy
DOUBLE PRECISION qxx, qxy, qyy, r, rx, rxm, ry, rym, x, xx, xy
DOUBLE PRECISION y, yx, yy
INTEGER*4 i, iter, j
DIMENSION aa(513, 513), d(513, 513), dd(513, 513), rx(513, 513)
DIMENSION rxm(1000), ry(513, 513), rym(1000), x(513, 513)
DIMENSION y(513, 513)
DO iter = 1, 5, 1
  DO j = 1, 511, 1
    DO i = 1, 511, 1
      rxm(iter) = MAX(rxm(iter), ABS(rx(1+i, 1+j)))
      rym(iter) = MAX(rym(iter), ABS(ry(1+i, 1+j)))
    ENDDO
  ENDDO
ENDDO
END
```

Fig. B.2: TOMCATV: loop number 2.

```
INTRINSIC abs, max
DOUBLE PRECISION a, aa, abx, aby, b, c, d, dd, pxx, pxy, pyy
DOUBLE PRECISION qxx, qxy, qyy, r, rx, rxm, ry, rym, x, xx, xy
DOUBLE PRECISION y, yx, yy
INTEGER*4 i, iter, j
DIMENSION aa(513, 513), d(513, 513), dd(513, 513), rx(513, 513)
DIMENSION rxm(1000), ry(513, 513), rym(1000), x(513, 513)
DIMENSION y(513, 513)
DO iter = 1, 5, 1
  DO j = 1, 510, 1
    DO i = 1, 511, 1
      r = aa(1+i, 2+j)*d(1+i, 1+j)
      d(1+i, 2+j) = 1.D0/(dd(1+i, 2+j)+(-1)*aa(1+i, 1+j)*r)
      rx(1+i, 2+j) = rx(1+i, 2+j)+(-1)*rx(1+i, 1+j)*r
      ry(1+i, 2+j) = ry(1+i, 2+j)+(-1)*ry(1+i, 1+j)*r
    ENDDO
  ENDDO
ENDDO
END
```

Fig. B.3: TOMCATV: loop number 3.

```
INTRINSIC abs, max
DOUBLE PRECISION a, aa, abx, aby, b, c, d, dd, pxx, pxy, pyy
DOUBLE PRECISION qxx, qxy, qyy, r, rx, rxm, ry, rym, x, xx, xy
DOUBLE PRECISION y, yx, yy
INTEGER*4 i, iter, j
DIMENSION aa(513, 513), d(513, 513), dd(513, 513), rx(513, 513)
DIMENSION rxm(1000), ry(513, 513), rym(1000), x(513, 513)
DIMENSION y(513, 513)
DO iter = 1, 5, 1
  DO j = 1, 510, 1
    DO i = 1, 511, 1
      rx(1+i, (-1)+513+(-1)*j) = d(1+i, (-1)+513+(-1)*j)*(rx(1+i, (-1
*)+513+(-1)*j)+(-1)*aa(1+i, (-1)+513+(-1)*j)*rx(1+i, 513+(-1)*j))
      ry(1+i, (-1)+513+(-1)*j) = d(1+i, (-1)+513+(-1)*j)*(ry(1+i, (-1
*)+513+(-1)*j)+(-1)*aa(1+i, (-1)+513+(-1)*j)*ry(1+i, 513+(-1)*j))
    ENDDO
  ENDDO
ENDDO
END
```

Fig. B.4: TOMCATV: loop number 4.

```
INTRINSIC abs, max
DOUBLE PRECISION a, aa, abx, aby, b, c, d, dd, pxx, pxy, pyy
DOUBLE PRECISION qxx, qxy, qyy, r, rx, rxm, ry, rym, x, xx, xy
DOUBLE PRECISION y, yx, yy
INTEGER*4 i, iter, j
DIMENSION aa(513, 513), d(513, 513), dd(513, 513), rx(513, 513)
DIMENSION rxm(1000), ry(513, 513), rym(1000), x(513, 513)
DIMENSION y(513, 513)
DO iter = 1, 5, 1
  DO j = 1, 511, 1
    DO i = 1, 511, 1
      x(1+i, 1+j) = rx(1+i, 1+j)+x(1+i, 1+j)
      y(1+i, 1+j) = ry(1+i, 1+j)+y(1+i, 1+j)
    ENDDO
  ENDDO
ENDDO
END
```

Fig. B.5: TOMCATV: loop number 5.

```
INTEGER*4 i, j
REAL*4 u, v, psi
REAL*4 dx, dy
DIMENSION u(513, 513), v(513, 513), psi(513, 513)
DO j = 1, 512, 1
  DO i = 1, 512, 1
    u(i+1, j) = -(psi(i+1, j+1)-psi(i+1, j))*dy
    v(i, j+1) = (psi(i+1, j+1)-psi(i, j+1))*dx
  ENDDO
ENDDO
END
```

Fig. B.6: SWIM: loop number 1.

```
REAL*4 u, v, p, uold, vold, pold
DIMENSION u(513, 513), v(513, 513), p(513, 513)
DIMENSION uold(513, 513), vold(513, 513), pold(513, 513)
DO j = 1, 513, 1
  DO i = 1, 513, 1
    uold(i, j) = u(i, j)
    vold(i, j) = v(i, j)
    pold(i, j) = p(i, j)
  ENDDO
ENDDO
END
```

Fig. B.7: SWIM: loop number 2.

```
 INTEGER*4 i, itmax, j, m, mp1, mprint, n, np1
 REAL*4 a, alpha, cu, cv, di, dj, dt, dx, dy, el, fsdx, fsdy, h
 REAL*4 p, pcf, pi, pnew, pold, psi, tdt, tpi, u, unew, uold, v
 REAL*4 vnew, vold, z
 DIMENSION cu(513, 513), cv(513, 513), h(513, 513), p(513, 513)
 DIMENSION pnew(513, 513), pold(513, 513), psi(513, 513)
 DIMENSION u(513, 513), unew(513, 513), uold(513, 513)
 DIMENSION v(513, 513), vnew(513, 513), vold(513, 513)
 DIMENSION z(513, 513)
 COMMON /cons/ dt, tdt, dx, dy, a, alpha, itmax, mprint, m, n, mp1,
* np1, el, pi, tpi, di, dj, pcf
 COMMON // u, v, p, unew, vnew, pnew, uold, vold, pold, cu, cv, z,
*h, psi
 DO j = 1, 512, 1
   DO i = 1, 512, 1
     cu(i+1, j) = .5*(p(i+1, j)+p(i, j))*u(i+1, j)
     cv(i, j+1) = .5*(p(i, j+1)+p(i, j))*v(i, j+1)
     z(i+1, j+1) = (fsdx*(v(i+1, j+1)-v(i, j+1))-fsdy*(u(i+1, j+1)-u(
*i+1, j)))/(p(i, j)+p(i+1, j)+p(i+1, j+1)+p(i, j+1))
   h(i, j) = p(i, j)+.25*(u(i+1, j)*u(i+1, j)+u(i, j)*u(i, j)+v(i,
*j+1)*v(i, j+1)+v(i, j)*v(i, j))
   ENDDO
 ENDDO
 END
```

Fig. B.8: SWIM: loop number 3.

```
 INTEGER*4 i, itmax, j, m, mp1, mprint, n, np1
 REAL*4 unew, vnew, pnew, uold, vold, pold, cu, cv, z, h
 REAL*4 tdts8, tdtsdx, tdtsdy
 DIMENSION unew(513, 513), vnew(513, 513), pnew(513, 513)
 DIMENSION uold(513, 513), vold(513, 513), pold(513, 513)
 DIMENSION cu(513, 513), cv(513, 513), z(513, 513), h(513, 513)
 DO j = 1, 512, 1
   DO i = 1, 512, 1
     unew(i+1, j) = (uold(i+1, j)+tdts8*(z(i+1, j+1)+z(i+1,j))*(cv(i
*+1, j+1)+cv(i, j+1)+cv(i, j)+cv(i+1, j)))-tdtsdx*(h(i+1, j)-h(i, j
*))
     vnew(i, j+1) = (vold(i, j+1)-tdts8*(z(i+1, j+1)+z(i,j+1))*(cu(i
*+1, j+1)+cu(i, j+1)+cu(i, j)+cu(i+1, j)))-tdtsdy*(h(i, j+1)-h(i,j
*))
     pnew(i, j) = (pold(i, j)-tdtsdx*(cu(i+1, j)-cu(i,j)))-tdtsdy*(c
*v(i, j+1)-cv(i, j))
   ENDDO
 ENDDO
 END
```

Fig. B.9: SWIM: loop number 4.

```
      INTEGER*4 i, j
      REAL*4 u,v,p, unew, vnew, pnew, uold, vold, pold
      DIMENSION u(513, 513), v(513, 513), p(513,513)
      DIMENSION unew(513, 513), vnew(513,513), pnew (513,513)
      DIMENSION uold(513,513), vold(513,513), pold(513,513)
      DO j = 1, 513, 1
        DO i = 1, 513, 1
          uold(i, j) = u(i, j)
          vold(i, j) = v(i, j)
          pold(i, j) = p(i, j)
          u(i, j) = unew(i, j)
          v(i, j) = vnew(i, j)
          p(i, j) = pnew(i, j)
        ENDDO
      ENDDO
      END
```

Fig. B.10: SWIM: loop number 5.

```
 INTEGER*4 i, itmax, j, m, mp1, mprint, n, np1
 REAL*4 a, alpha, cu, cv, di, dj, dt, dx, dy, el, h, p, pcf, pi
 REAL*4 pnew, pold, psi, tdt, tpi, u, unew, uold, v, vnew, vold
 REAL*4 z
 DIMENSION cu(513, 513), cv(513, 513), h(513, 513), p(513, 513)
 DIMENSION pnew(513, 513), pold(513, 513), psi(513, 513)
 DIMENSION u(513, 513), unew(513, 513), uold(513, 513)
 DIMENSION v(513, 513), vnew(513, 513), vold(513, 513)
 DIMENSION z(513, 513)
 COMMON /cons/ dt, tdt, dx, dy, a, alpha, itmax, mprint, m, n, mp1,
* np1, el, pi, tpi, di, dj, pcf
 COMMON // u, v, p, unew, vnew, pnew, uold, vold, pold, cu, cv, z,
*h, psi
 DO j = 1, 512, 1
   DO i = 1, 512, 1
     uold(i, j) = u(i, j)+alpha*((unew(i, j)-2.*u(i, j))+uold(i, j))
     vold(i, j) = v(i, j)+alpha*((vnew(i, j)-2.*v(i, j))+vold(i,j))
     pold(i, j) = p(i, j)+alpha*((pnew(i, j)-2.*p(i, j))+pold(i,j))
     u(i, j) = unew(i, j)
     v(i, j) = vnew(i, j)
     p(i, j) = pnew(i, j)
   ENDDO
 ENDDO
 END
```

Fig. B.11: SWIM: loop number 6.

```
PARAMETER (N=xx)
REAL*8 a(N,N), b(N,N), c(N,N)
DO i = 1, N
  DO j = 1, N
    DO k = 1, N
      a(i,j) = a(i,j) + b(i,k) * c(k,j)
    ENDDO
  ENDDO
ENDDO
END
```

Fig. B.12: MATMUL

```
PARAMETER (N=xx)
DOUBLE PRECISION a(N,N), b(N,N)
DO i=1,N
   DO j=1,N
     a(i,j) = b(j,i)
   ENDDO
ENDDO
END
```

Fig. B.13: T2D

```
 PARAMETER (N=xx)
 DOUBLE PRECISION A11,A12,A13,A21,A22,A23,A31,A32,A33
 DOUBLE PRECISION DU1(N), DU2(N), DU3(N)
 DOUBLE PRECISION U1(N,N,2), U2(N,N,2), U3(N,N,2)
 DO 8 kx=2,N-1
   DO 8 ky=2,N-1
     DU1(ky)=U1(kx,ky+1,nl1) - U1(kx,ky-1,nl1)
     DU2(ky)=U2(kx,ky+1,nl1) - U2(kx,ky-1,nl1)
     DU3(ky)=U3(kx,ky+1,nl1) - U3(kx,ky-1,nl1)
     U1(kx,ky,nl2)=U1(kx,ky,nl1) +A11*DU1(ky) +A12*DU2(ky)+A13*DU3(ky)
1 + SIG*(U1(kx+1,ky,nl1) -fw*U1(kx,ky,nl1) +U1(kx-1,ky,nl1))
     U2(kx,ky,nl2)=U2(kx,ky,nl1) +A21*DU1(ky) +A22*DU2(ky)+A23*DU3(ky)
1 + SIG*(U2(kx+1,ky,nl1) -fw*U2(kx,ky,nl1) +U2(kx-1,ky,nl1))
     U3(kx,ky,nl2)=U3(kx,ky,nl1) +A31*DU1(ky) +A32*DU2(ky)+A33*DU3(ky)
1 + SIG*(U3(kx+1,ky,nl1) -fw*U3(kx,ky,nl1) +U3(kx-1,ky,nl1))
8 CONTINUE
 END
```

Fig. B.14: ADI

```
      PARAMETER (N=xx)
      REAL*8 b, c, arow
      DIMENSION b(N,N), c(N,N), arow(N)
      DO i=1,N
        DO j=1,N
           DO k=1,N
            c(i,j) = c(i,j) + arow(k) * b(k,j)
          ENDDO
        ENDDO
      ENDDO
      END
```

Fig. B.15: MATVEC

```
 PARAMETER (N=xx)
 DOUBLE PRECISION A(N,N), B(N,N), C(N,N), D(N,N),
$ E(N,N), F(N,N,3), X(N,N), Y(N,N)
 DO j = 3,N-2
   DO k = 1,N
     RLD2 = A(J,K)
     RLD1 = B(J,K) - RLD2*X(J-2,K)
     RLD = C(J,K) - (RLD2*Y(J-2,K) + RLD1*X(J-1,K))
     RLDI = 1./RLD
     F(J,K,1) = (F(J,K,1) - RLD2*F(J-2,K,1) - RLD1*F(J-1,K,1))*RLDI
     F(J,K,2) = (F(J,K,2) - RLD2*F(J-2,K,2) - RLD1*F(J-1,K,2))*RLDI
     F(J,K,3) = (F(J,K,3) - RLD2*F(J-2,K,3) - RLD1*F(J-1,K,3))*RLDI
     X(J,K) = (D(J,K) - RLD1*Y(J-1,K))*RLDI
     Y(J,K) = E(J,K)*RLDI
   ENDDO
 ENDDO
 END
```

Fig. B.16: VPENTA

Bibliography

[1] J. Abella, A. González, J. Llosa, and X. Vera. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. In *Proceedings of 31st International Conference on Parallel Processing (ICPP02) Workshops*, Aug. 2002.

[2] W. Abu-Sufah. *Improving the performance of virtual memory computers*. PhD thesis, University of Illinois at Urbana-Champaign, Nov. 1978.

[3] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *10th European Conference on Object-Oriented Programming (ECOOP'96)*, pages 142–166, 1996.

[4] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, 1987.

[5] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behaviour prediction by abstract interpretation. In *Proceedings of Static Analysis Symposium (SAS'96)*, Lecture Notes in Computer Science (LNCS) 1145, pages 52–66. Springer-Verlag, Sep. 1996.

[6] G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, pages 85–96, 1997.

[7] R. Arnold, F. Müeller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of 15th Real-Time Systems Symposium (RTSS'94)*, pages 172–181, 1994.

[8] E. Ayguadé, C. Barrado, A. González, J. Labarta, J. Llosa, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera, and M. Valero. Ictineo: a tool for research on ilp. In *Proceedings of Supercomputing (SC'96)*, 1996. Research Exhibit "Polaris at Work".

[9] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. Technical report, University of California, 1994.

[10] J-L. Baer and T-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of ACM International Conference on Supercomputing (ICS'97)*, pages 176–186, Nov. 1991.

[11] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[12] S. Basumallick and K. Nielsen. Cache issues in real-time systems. In *Proceedings ACM Workshop on Languages, Compilers and Tools for Real-Time Systems (LCTES'94)*, Jun. 1994.

[13] R. Bedichek. Talismam: Fast and accurate multicomputer simulation. In *Proceedings of ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'95)*, pages 14–24, May 1995.

[14] N. Bermudo, X. Vera, A. González, and J. Llosa. An efficient solver for cache miss equations. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, 2000.

[15] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21:475–480, 1995.

[16] A. Burns and A. Wellings. The impact of an Ada run-time system's performance characteristics on scheduling models. In *Proceedings of 12th Ada-Europe International Conference*, pages 240–248, Jun. 1993.

[17] J. V. Busquets-Mataix, J. J. Serrano, R. .Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of 2nd Real-Time Technology and Applications Symposium (RTAS'96)*, Jun. 1996.

[18] J. V. Busquets-Mataix, J. J. Serrano, and A. Wellings. Hybrid instruction cache partitioning for preemptive real-time systems. In *Proceedings of 9th Euromicro Workshop on Real-Time Systems (EUROMICRO-RTS'97)*, Jun. 1997.

[19] M. Butler, T-Y. Yeh, Y.N. Patt, M. Alsup, H. Sales, and M. Shebanow. Instruction level parallelism is greater than two. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA'91)*, pages 276–286, 1991.

[20] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222, 1997.

[21] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of IV International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, pages 40–52, Apr. 1991.

[22] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, 2001.

[23] D. Carmean. *Inside the Pentium 4 Processor Micro-Architecture (www.intel.com/pentium4)*, 2000.

[24] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing (SC'92)*, pages 114–124, Nov. 1992.

[25] S. Carr, K.S. McKinley, and C-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of VI Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pages 252–262, Oct. 1994.

[26] C.-Y. Chang, J.-P. Sheu, and H.-C. Chen. Reducing cache conflicts by multilevel cache partitioning and array elements mapping. In *Proceedings of 7th International Conference on Parallel and Distributed Systems (ICPADS'00)*, 2000.

[27] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software — Practice and Experience*, 25:249–369, 1992.

[28] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layout for hierarchical memory systems. In *Proceedings of ACM International Conference on Supercomputing (ICS'99)*, pages 444–453, Rhodes, Greece, Jun. 1999.

[29] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI'01)*, pages 286–297, 2001.

[30] T-F. Chen and J-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of 21st International Symposium on Computer Architecture (ISCA'94)*, 1994.

[31] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 191–202, 2001.

[32] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209, 2002.

[33] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 205–217, 1995.

[34] P. Clauss. Counting solutions to linear and non-linear constraints through Ehrhart polynomials. In *Proceedings of ACM International Conference on Supercomputing (ICS'96)*, pages 278–285, Philadelphia, 1996.

[35] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 279–290, Jun. 1995.

[36] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, pages 1–9, 1999.

[37] M. H. DeGroot. *Probability and statistics*. Addison-Wesley, 1998.

[38] M. Dyer and A. M. Frieze. On the complexity of computing the volumne of a polyhedron. *SIAM J. Comput.*, 17(5):967–974, 1988.

[39] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proceedings of 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Dec. 1999.

[40] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of Euro-Par (EUROPAR'97)*, pages 1298–1307, Aug. 1997.

[41] Y. Ermoliev and R. J.-B. Wets. *Numerical Techniques for Stochastic Optimization*. Springer-Verlag, 1988.

[42] K. A. Faigin, J. P. Hoeflinger, D. A. Padua, P. M. Petersen, and S. A. Weatherford. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, Oct. 1994.

[43] P. Feautrier. Parametric integer programming. *Operations Research*, 22:243–268, 1988.

[44] P. Feautrier. Automatic parallelization in the polytope model. In G. R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 79–103. Springer Verlag, 1996.

[45] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17:131–181, 1999.

[46] A. Fernández. A quantitative analysis of the SPECfp95. Technical Report UPC-DAC-1999-12, Universitat Politècnica de Catalunya, March 1999.

[47] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *4th Workshop on Languages and Compilers for Parallel Computing (LCPC'91)*, pages 328–343, 1991.

[48] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, 1999.

[49] J. Frailong, W. Jalby, and J. Lenfant. XOR-schemes: a flexible data organization in parallel memories. In *Proceedings of International Conference on Parallel Processing (ICPP'85)*, pages 276–283, 1985.

[50] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[51] J. Gee, M. Hill, D. Pnevmatikatos, and A.J. Smith. Cache performance of the spec92 benchmark suite. *IEEE Micro*, pages 17–27, Aug. 1993.

[52] 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 (TOPLAS)*, 21(4):703–746, 1999.

[53] S. Ghosh, M. Martonosi, and S. Malik. Automated cache optimizations using CME driven diagnosis. In *Proceedings of International Conference on Supercomputing (ICS'00)*, pages 316–326, 2000.

[54] Gill, Murray, and Wright. *Practical optimization.* Academic Press, 1981.

[55] Glover and Laguna. *Tabu search.* Kluwer, 1997.

[56] A.J. Goldberg and J. Hennessy. Performance debugging shared memory multiprocessor programs with mtool. In *Proceedings of Supercomputing (SC'91)*, pages 481–490, 1991.

[57] D. E. Goldberg. *Genetic algorithms in search, optimizations and machine learning.* Addison-Wesley, 1989.

[58] S. Goldschmidt and J. Hennessy. The accuracy of trace-driven simulation of multiprocessors. In *Proceedings of ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'93)*, pages 146–157, May 1993.

[59] G. W. Gréwal and C. T. Wilson. Mapping reference code to irregular dsps within the retargetable, optimizing compiler COGEN(T). In *Proceedings of*

the 34th annual ACM/IEEE international symposium on Microarchitecture, pages 192–202, 2001.

[60] J. Gustafsson. *Analyzing Execution Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Uppsala University, May 2000.

[61] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. In *1993 Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*, pages 567–585, Portland, Ore., Aug. 1993. Springer Verlag.

[62] Hansen, Jaumard, and Mathon. Constrained nonlinear 0-1 programming. *ORSA Journal on Computing*, 1995.

[63] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical modeling of set-associative caches. *IEEE Transactions on Computers*, 48(10):1009–1024, Oct. 1999.

[64] C. A. Healey, D. Whalley, and M. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of 16th Real-Time Systems Symposium (RTSS'95)*, pages 288–297, 1995.

[65] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proceedings of 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, Jun. 1998.

[66] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.

[67] C. Healy and D. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proceedings of 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, Jun. 1999.

[68] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufman Publishers, 1996.

[69] M. Hill. *DineroIII: a uniprocessor cache simulator (http://www.cs.wisc.edu/~larus/warts.html)*.

[70] J. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, 1975.

[71] N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, Sep. 2000.

[72] R. Horst, P. M. Pardalos, and N. V. Thoai. *Introduction to Global Optimization*. Kluwer Academic Publishers, 1995.

[73] IBM Microelectronics Division. *The PowerPC 440 core*, 1999.

[74] D. T. Harper III and J. R. Jump. Vector access performance in parallel memories. *IEEE Transactions on Computers*, C(36):1440–1449, 1987.

[75] Integrated Device Technologies. *79RC64574/RC64575 Data Sheet*, 2001.

[76] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of 14th Real-Time Systems Symposium (RTSS'93)*, pages 212–221, Dec. 1993.

[77] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems (TOCS)*, 20(4):369–397, 2002.

[78] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[79] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of 17th International Symposium on Computer Architectures (ISCA'90)*, 1990.

[80] M. Kandemir, A. Choudhary, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, Feb. 1999.

[81] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of International Conference on Microprogramming and Microarchitecture*, pages 285–296, 1998.

[82] A. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, 19:920–934, 1993.

[83] K. Kennedy, D. Callahan, and A. Porterfield. Analyzing and visualizing performance of memory hierarchy. In *Instrumentation for Visualization*. ACM Press, New York, 1990.

[84] K. Kennedy and K.S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. Technical Report COMP TR92-189, Rice University, August 1992.

[85] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.

[86] R. E. Kessler, E. J. McLellan, and D. A. Webb. *The Alpha 21264 microprocessor architecture*, 1999.

[87] S. K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, 1996.

[88] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proceedings of 10th Real-Time Systems Symposium (RTSS'89)*, Dec. 1989.

[89] Kirkpatrick, Gelatt, and Vecchi. Optimization by simulated annealing. *Science 220*, 1983.

[90] R. Kirner and P. Puschner. Transformation of path information for WCET analysis during compilation. In *Proceedings of 13th Euromicro Conference of Real-Time Systems (ECRTS'01)*, Jun. 2001.

[91] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI'97)*, pages 346–357, 1997.

[92] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES'03)*, pages 12–23, 2003.

[93] M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance of blocked algorithms. In *Proceedings of IV International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, Apr. 1991.

[94] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, 1994.

[95] D. Lawrie and C. Vora. The prime memory system for array access. *IEEE Transactions on Computers*, C(31):435–442, 1982.

[96] A.R. Lebeck and D.A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, Oct. 1994.

[97] 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 Transaction on Computers*, 47, 1998.

[98] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 88–98, 1995.

[99] Y. T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of 16th Real-Time Systems Symposium (RTSS'95)*, pages 298–307, 1995.

[100] Y. T. S. Li, S. Malik, and A. Wolfe. Cache modeling and path analysis for real-time software. In *Proceedings of 17th Real-Time Systems Symposium (RTSS'96)*, 1996.

[101] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, 1997.

[102] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of 15th Real-Time Systems Symposium (RTSS'94)*, pages 97–108, 1994.

[103] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniqes. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, pages 1–15, Jun. 1998.

[104] T. Lundqvist and P. Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 255–262, Dec. 1999.

[105] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of 20th Real-Time Systems Symposium (RTSS'99)*, Dec. 1999.

[106] P. Magnusson. A design for efficient simulation of a multiprocessor. In *Proceedings of the Western Simulation Multiconference on Int. Workshop on MASCOTS-93*, pages 69–78, 1993. La Jolla, CA.

[107] N. Manjikian and T. Abdelrahman. Array data layout for the reduction of cache conflicts. In *Proceedings of 8th International Conference on Parallel and Distributed Systems (ICPADS'95)*, 1995.

[108] M. Martonosi, A. Gupta, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *Proceedings of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'92)*, pages 1–12, Jun. 1992.

[109] M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'93)*, 1993.

[110] The PIP System. Solving systems of affine (in)equalities: PIP's user's guide. http://www.prism.uvsq.fr/~paf.

[111] The SUIF Compiler Group. SUIF: An infrastructure for research on parallelizing and optimizing compilers. http://suif.stanford.edu.

[112] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Researh Lab, 1993.

[113] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, Jul. 1996.

[114] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of VII Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, 1996.

[115] Z. Michalewicz. *Genetic algorithms+Data structures=Evolution Programs*. Springer-Verlag, 1994.

[116] MIPS. *RISCompiler Languages Programmer's Guide*. MIPS, 1988.

[117] MIPS Technologies. *MIPS32 4Kp- Embedded, MIPS Processor Core*, 2001.

[118] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Artificial Intelligence: Methodology, Systems, Applications*, pages 40–50, 2002.

[119] Motorola Inc. *PowerPC 604e RISC Microprocessor Technical Summary*, 1996.

[120] T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of V International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, pages 62–73, Oct. 1992.

[121] F. Müeller. Compiler support for software-based cache partitioning. In *Proceedings ACM Workshop on Languages, Compilers and Tools for Real-Time Systems (LCTES'95)*, Jun. 1995.

[122] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[123] E. Petrank and D. Rawitz. Hardness of cache conscious data placement. In *Proceedings of International Conference on Principles of Programming Languages (POPL'02)*, 2002.

[124] A. K. Porterfield. *Software Methods for improvement of cache performance on supercomputer applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.

[125] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of 23th Real-Time Systems Symposium (RTSS'02)*, Dec. 2002.

[126] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, 35(8):102–114, Aug. 1992.

[127] W. Pugh. Counting solutions to Presburger formulas: how and why. In *Proceedings of ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 121–134, 1994.

[128] B. Rau. Pseudo-randomly interleaved memories. In *Proceedings of International Symposium on Computer Architecture (ISCA'91)*, pages 74–83, 1991.

[129] G. Rivera and C-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, 1998.

[130] G. Rivera and C-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of ACM Internacional Conference on Supercomputing (ICS'98)*, 1998.

[131] G. Rivera and C-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99)*, 1999.

[132] G. Rivera and C-W. Tseng. Locality optimizations for multi-level caches. In *Proceedings of Supercomputing (SC'99)*, 1999.

[133] F.J Sánchez and A. González. Fast, flexible and accurate data locality analysis. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, Oct. 1998.

[134] F.J Sánchez, A. González, and M. Valero. Static locality analysis for cache management. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, Nov. 1997.

[135] H. Sharangpani. Itanium microprocessor architecture. *IEEE Micro*, 2000.

[136] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 77–90, 2003.

[137] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method inlining for a Java just-in-time compiler. In *2nd Java Virtual Machine Research and Technology Symposium*, San Francisco, 2002.

[138] R. Sugumar. *Multi-configuration simulation algorithms for the evaluation of computer designs*. PhD thesis, University of Michigan, 1993.

[139] Sun Microelectronics. *microSPARC-IIep User's Manual*, 1997.

[140] Sun Microelectronics. *UltraSPARC-III Cu User's Manual*, Apr. 2003.

[141] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'94)*, pages 261–271, May 1994.

[142] O. Temam, E.D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for accessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing (SC'93)*, pages 410–419, 1993.

[143] O. Temam, W. Jalby, and C. Fricker. Cache interference phenomena. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'93)*, 1993.

[144] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Dec. 1998.

[145] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems*, 6(1):133–151, 1994.

[146] N. Topham, A. González, and J. González. The design and performance of a conflict-avoiding cache. In *Proceedings of 30th Symposium on Microarchitecture (MICRO-30)*, 1997.

[147] Torn and Zilinskas. *Global optimization*. Springer-Verlag, 1989.

[148] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 29(3):128–170, Sep. 1997.

[149] E. van der Deijl, G. Kanbier, O. Temam, and E.D. Granston. A cache visualization tool. *IEEE Computer*, 30(7):71–78, Jul. 1997.

[150] H. Vandierendonck and K. De Bosschere. Efficient profile-based evaluation of randomising set index functions for cache memories. In *Proceedings of 2nd International Symposium on Performance Analysis of Systems and Software (ISPASS'01)*, Tucson, Nov. 2001.

[151] H. Vandierendonck and K. De Bosschere. Highly accurate and efficient evaluation of randomising set index functions. *Journal of Systems Architecture*, 48(13–15):429–452, 2003.

[152] S. A. Vavasis. Nonlinear optimization, complexity issues. In *Oxford University Press*, 1991.

[153] X. Vera. Coyote project: The simulator. Technical Report MRTC Report 95/2003, Mälardalens Högskola, Apr. 2003.

[154] X. Vera, J. Abella, A. González, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proceedings of 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, New Orleans, Sept. 2003.

[155] X. Vera, N. Bermudo, J. Llosa, and A. González. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, To Appear.

[156] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, pages 272–282, Jun. 2003.

[157] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *Proceedings of International Real-Time Systems Symposium (RTSS03)*, Dec. 2003.

[158] X. Vera, J. Llosa, and A. González. Near-optimal padding for removing conflict misses. In *15th Workshop on Languages and Compilers for Parallel Computers (LCPC'02)*, July 2002.

[159] X. Vera, J. Llosa, A. González, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior. In *Proceedings of European Conference on Parallel Computing (Europar'00)*, 2000.

[160] X. Vera and J. Xue. Let's study whole program cache behaviour analytically. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA 8)*, Cambridge, Feb. 2002.

[161] D. A. B. Weikle, K. Skadron, S. A. McKee, and W. A. Wulf. Cache as filters: a unifying model for memory hierarchy analysis. Technical Report CS-2000-16, University of Virginia, Jun. 2000.

[162] R. T. White, F. Müeller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of Third IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, 1997.

[163] D. K. Wilde. A library for doing polyhedral operations. Technical Report 785, Oregon State University, 1993.

[164] R. P. Wilson. *Efficient context-sensitive pointer analysis for C programs*. PhD thesis, Stanford University, December 1997.

[165] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'96)*, May 1996.

[166] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of 29th ACM/IEEE International Symposium on Microarchitecture (MICRO-29)*, 1996.

[167] M. E. Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Stanford University, Mar. 1992.

[168] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Jun. 1991.

[169] A. Wolfe. Software-based cache partitioning for real-time applications. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*, 1993.

[170] M. Wolfe. Advanced loop interchanging. In *Proceedings of International Conference on Parallel Processing (ICPP'96)*, 1996.

[171] M. E. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[172] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA'03)*, pages 84–97, 2003.

[173] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, 1997.

[174] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. *International Journal of Parallel Programming*, 26(6):671–696, 1998.

[175] J. Xue and X. Vera. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Transactions on Computers*, To Appear.

Credits

So, I spent few months writing this dissertation in between trips and at night on my laptop, drinking way too much *diet coke with lemon flavor*, listening to a lot of music, and bugging everybody to read drafts. I thank Björn, Janne and Jaume for reading the whole thesis, and Ebbe and Toni for checking some chapters. Any mistakes are mine, and were made over all their objections.

That represents the end of a *personal* research line that started 6 years ago at UPC in Barcelona. At that time, I was working on the MHAOTEU Esprit project, where I started implementing the original CMEs with Carles Ciuraneta. Soon, he left and Nerina Bermudo, at that time already a very good friend of mine, joined the project. Under the supervision of Antonio González and Josep Llosa, we carried out our Master Thesis' and got the first practical implementation of the CMEs. That was what Nerina and I used to call the *Randy Project*, which remains at UPC and was used by Jaume Abella and me to develop the first implementation of our compiler cache optimizations.

Based on our experience, Nerina and I re-implemented the kernel that it's used in this thesis at IDt/MDH (we named it *Coyote Project*, and it's publicly available at `http://www.mrtc.mdh.se/projects/wcet/Coyote/`). I'm really indebted to her; without her collaboration, her patience fixing bugs, and our discussions about how to get the best performance, this thesis wouldn't have achieved these results.

The real-time part of this thesis couldn't have been done without Ebbe's help. Jakob Engblom has been always there with his technical comments and broad knowledge about *everything*. I'm also grateful to Thomas Häoveken from NEC Electronics (Europe) for supplying information about the NEC controllers.

I also want to thank (in no particular order), for so many hours of company:

Less Than Jake, Frenzal Rhomb, Bouncing Souls, The Living End,
One Dollar Short, Goldfinger, Millencolin,
those corny pop bands (singers) I'm embarrassed to write down (oops), Ministry of Sound, Sash!, Him, Sonique, Vendetta, Deadstar, Green Day,

all the golden oldies, Kent, Offspring, Area 7, MxPx,
Unwritten Law, Los Piratas, Rancid, Lucky 7, Nerf Herder, NoFx, Paul van Dyck,
Lash, 28 Days, all the mates at *bmwfaq*, and more than any,
Randy!