UNIVERSITY OF CALIFORNIA,
IRVINE

The X-Legion
A Compiler-Approach to Exploit Locality and Portability of Divide-And-Conquer Algorithms

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Paolo Nicola D'Alberto

Dissertation Committee:
Professor Alexandru Nicolau, Chair
Professor Alexander Veidenbaum
Professor Nikhil Dutt

2005

The dissertation of Paolo Nicola D'Alberto
is approved and is acceptable in quality
and form for publication on microfilm:

_____
_____
_____
_____
Committee Chair

University of California, Irvine
2005

Lame Deer, Seeker of Visions
John (Fire) Lame Deer and Richard Erdoes
Medicine, Good and Bad, pag 162-163.

Even animals of the same kind - two deer, two owls - will behave differently from each other. Even your daughter's little pet hamsters, they all have their own ways. I have studied many plants. The leaves of one plant, on the same stem -none is exactly alike. On all the earth there is not one leaf that is exactly like another. The Great Spirit likes it that way. He only sketches out the path of life roughly for all the creatures on earth, shows them where to go, where to arrive at, but leaves them to find their own way to get there. He wants them to act independently according to their nature, to the urges in each of them.

[...]

I believe that being a medicine man, more than anything else, is a state of mind, a way of looking at and understanding this earth, a sense of what it is all about. Am I a wićaśa wakan? I guess so. What else can or would I be? Seeing me in my patched-up, faded shirt, with my down-at-the-heels cowboys boots, the hearing aid whistling in my ear, looking at the flimsy shack with its bad-smelling outhouse which I call my home - it all doesn't add up to a white man's idea of a holy man. You have seen me drunk and broke. You heard me curse or tell sexy jokes. You know I am not better and wiser than other men. But I've been up on the hilltop, got my vision and my power; the rest is just trimmings. That vision never leaves me - not in jail, not while I'm painting funny signs advertising some hash house, not when I am in a saloon, not while I am with a woman, especially not then.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLE

ACKNOWLEDGMENTS

The support of my wife Shanyi made this work possible. She is truly everything to me.

In all these years, Prof. Alexandru Nicolau has been the greatest mentor. Our relationship has had both highs and lows, as every relationship, however, now that I look back on my life, I understand that he has always been there to help me, to guide me and to let me grow both personally and professionally. Now, I appreciate it and I know that nobody could do any better.

A heartfelt thanks goes to Prof. Gianfranco Bilardi. He sowed the seed of the recursion-DAG while we were working on the fractal matrix multiplication. Another thank you goes to Prof. Rajesh Gupta, who found ways to support my research emotionally/economically when I needed it and let my teaching experience start in a nice environment such as UCSD.

I would like to thank also Prof. Nikil Dutt and Prof. Alexander Veidenbaum (members of my topic-defense and final-defense committee), for their useful advises during the preparation of my final-defense talk and thesis.

In the following, I also thank who have collaborated with me at any time and anyway since the beginning till the final stages of this thesis. I will also give the context/chapter where they had a remarkable contribution.

Acknowledgments: Chapter 3

This work has been supported in part by Phi Beta Kappa Alumni in Southern California, and by NSF Contract Number ACI 0204028 and AMRM DABT63-98-C-0045. I want to thank Arun Kejariwal and Rafael Lopes for their help and time for correcting my presentation(s)/draft(s) of this chaprter. I also thank Paul Stodghill for the live discussions from which JuliusC got started.

Acknowledgments: Chapter 4

This work has been supported in part by AMRM DABT63-98-C-0045. I thank Prof. Victor Prasanna, who introduced me to the first recursive algorithm for all-pair shortest path. I thank Nicolae Savoiu for making possible the access and the use of Windows systems to collect data, and for introducing me to Unreal Tournament, swimming and other silly things.

I thank Peter Gabi Grun and Ashok Alambi, who have been there answering my questions about scheduling and register renaming, and playing basketball.

Acknowledgments: Chapter 5

Financial support has in part been provided by DARPA/ITO under contract DABT63-98-C-0045. A huge thank you goes to Prof. Vincent Loechner, Somnath Ghosh, Prof. Dan Hirschberg and the members of AMRM project. In particular, I thank immensely Arun Kejariwal. They helped on Ehrhart polynomials and the existence test, cache miss equation determination, interference estimation and moral/technical support, respectively.

Acknowledgments: Chapter 6

This work is supported in part by NSF, Contract Number ACI 0204028. I want to especially thank Prof. Alexander Veidenbaum and Prof. Tony Givargis for helping me in the hardware organization of a cache with dynamic mapping.

I would also like to thank the following people (in no particular order): Melanie Sanders who made my experience in *using* UCI's facilities painless, I have always appreciated her friendship; Radu Cornea who helped me find my way in learning Linux's secrets and watching silly movies; Weiyu Tang for his advise on how to interpret the simulation results of adaptive memory hierarchies and for helping me out in settling-down in Irvine seven years ago. I also thank Haitao Du and Sanjay Velamparambil for the effort to disclose the *inner beauty* of Methods of Moments (MoM) and for collecting experimental results; Carol Rapp has always been kind to me and made me feel welcome; Patrick Murphy who made me appreciate the quarter final week as well as the teaching practice in a completely different way. I thank all my TAs and graders who put up with my demands and my students' requests.

I thank my mum Angela, my brothers Antonio and Fedele who really supported me and still believe in me unconditionally. Unfortunately, I wish my father Giacomo could see me now receiving my final (and last) degree and having my first real job. I wish my friend Giangiacomo was still riding this planet and, therfore, we could have yet another motorcycle trip.

CURRICULUM VITAE

Paolo Nicola D'Alberto

1995 B.S. In Electrical and Computer Engineering, University of Padua. Thesis title: "Space Complexity of DAG Computations in Memory Hierarchies"

1997 First modulo of Master in Software Engineering, Tecnopadova and S.E.R.C. Padua, Italy.

1998-2002 Research Assistant,
Department of Computer Science, University of California, Irvine.

2000 *Dottorato di Ricerca.*
Thesis title: "Performance evaluation of Data exploitation"
Joint collaboration: University of Bologna, Dept. of Compute Science; University of Padua, Dept. of Mathematics Pure and Applied; University of Venice, Dept. of Compute Science

2002-2004 Teaching Assistant,
Department of Computer Science, University of California, Irvine.

2003-2004 Teacher Associate,
Department of Computer Science, University of California, Irvine; Department of Computer Science and Engineering, University of California, San Diego.

2005 Ph.D. in Information and Computer Science,
Department of Computer Science, University of California, Irvine.

PUBLICATIONS

- In Journals

  1. P. D'Alberto, A. Nicolau, A. Veidenbaum and R. Gupta: "Line Size Adaptivity Analysis of Parameterized Loop Nests for Direct Mapped Data Cache". Transactions on Computers, IEEE Society, Feb 2005.

- In Book Chapters

  1. P. D'Alberto, A. Veidenbaum, A. Nicolau and R. Gupta: "Static Analysis of Parameterized Loop Nests for Energy Efficient Use of Data Caches". Workshop on Compilers and Operating Systems for Low Power 2001 (COLP01) - Chapter in a Kluwer book.

- In Conferences/Workshops

  1. P. D'Alberto and A. Nicolau: "JuliusC: A Practical Approach for the Analysis of Divide-And-Conquer Algorithms". The 17th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2004
  2. A. Kejariwal, P. D'Alberto, A. Nicolau and C.D. Polychronopoulos: "A Geometric Approach for Partitioning N-Dimensional Non-Rectangular Iteration Space". LCPC 2004
  3. P. D'Alberto, A. Nicolau and A. Veidenbaum: "A Data Cache with Dynamic Mapping". The 16th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2003.
  4. G. Bilardi, P. D'Alberto and A. Nicolau: "Fractal Matrix Multiplication: a Case Study on Portability of Cache Performance". Workshop on Algorithm Engineering WAE 2001
  5. G. Bilardi, A. Pietracaprina and P. D'Alberto: "On the space and access complexity of computation DAGs". Workshop on Graph-Theoretic Concepts in Computer Science 2000

- In Technical Reports

  1. H. Du, P. D'Alberto, R. Gupta, A. Nicolau and A. Veidenbaum: "A quantitative evaluation of adaptive memory hierarchy". UCI Technical Report (2002)
  2. P. D'Alberto: "MIPS R12000 Processor Performance Evaluation by SPEC2000 Benchmarks and Performance Counters". UCI Technical Report (2001)
  3. P. D'Alberto: "Fractal LU-decomposition with partial pivoting". UCI Technical Report (2001)
  4. P. D'Alberto: "Performance Evaluation of Data Locality Exploitation". (2000 University of Bologna Technical Report - Thesis)

ABSTRACT OF THE DISSERTATION

The X-Legion

A Compiler-Approach to Exploit Locality and Portability of Divide-And-Conquer Algorithms

by

Paolo Nicola D'Alberto

Doctor in Philosophy in Information and Computer Science

Department of Computer Science

University of California, Irvine, 2005

Professor Alexandru Nicolau, Chair.

The solution of linear systems is an ancient and inexhaustible problem. In time, the theory and the techniques to solve this problem have evolved and, today, hardware and software technologies thrive to achieve high-performance systems for the efficient solution of systems based on what is now defined as **matrix linear algebra**, and, in turn, efficient matrix computations and matrix algorithms.

My main contribution is in the investigation and implementation of techniques for the analysis of data locality in **divide-and-conquer** (D&C) algorithms, both recursive and non recursive, and their application to drive novel memory-hierarchy adaptations at compile time –i.e., static– or at run time –i.e., dynamic. I show the practicality and benefits of these techniques when applied to algorithms for linear algebra, matrix computation kernels and graph manipulation. In practice, I investigate and propose techniques for code organization through data-layout and computation re-organization in order to exploit data locality. Especially in this work, I present compiler-driven optimizations for the best utilization of the

memory hierarchy starting from the register files to the efficient utilization of the translation look-aside buffer (TLB) and, thus, hard-disk access(es).

These compile-time techniques are deployed in an investigative framework: X-Legion compiler –i.e., the tenth-legion compiler. JuliusC (Julius Caesar) is the leading component of the X-Legion compiler and it analyzes the division process of recursive D&C algorithms. Such an information leads to the modeling of the computation unfolding, the (automatic) reduction of the algorithm division work, and the targeting of optimizations driven by another compiler component, STAMINA. In fact, STAMINA is an analysis tool for the estimation of cache interference due to data memory reference in loop nests. In turn, the interference analysis drives the application of further optimizations to exploit spatial and temporal locality such as cache-line size adaptation and dynamic mapping.

CHAPTER 1

Thesis Statement and Contribution

Intuitively in a **Divide-and-Conquer** (D&C) algorithm, a computation has *locality* when we can find a suitable division of the computation into sub-computations, and every sub-computation has a high ratio between number of operations and number of input data. The ratio estimates the computation's inherent ability to amortize the cost of reading the input data over the amount of processing based on the input data. In practice, a high ratio means that the computation exploits data reuse and it spends little time reading data and more time performing *useful* computations. Otherwise, a low ratio means that the computation has little data reuse, and, thus, the same data read are used for few computations.

The exploitation of locality is crucial for any applications executed on modern architectures with deep memory hierarchy. Indeed, if the application has data locality, the architecture is able to store the most frequent data in small and fast caches so reducing the average data access time and speeding up the computation progression. In fact, data locality exploitation must be addressed starting from the design of the algorithm, while building the basic blocks of the application, to the design of the algorithms' data structures and layout.

For a specific application, a developer may devise an estimation-measure and she may use it to guide the design of **cache conscious codes** for the application; that is, codes that are tailored to take advantage of the presence of caches and their specific organization. Though in practice, an estimation often suffices an experienced developer in the design and

implementation of specific codes – e.g., for a first and preliminary solution to a problem; however, an estimation may not suffice an application-specific code generator and, certainly, it will not suffice a general-purpose compiler.

In practice, an estimate must be validated so as to be reliable and truthful, and it should be practical to use and apply. Moreover, an automatic approach needs more than a guideline, it needs a measure, a number, which quantifies the contribution of a technique precisely and concisely. In fact, the capability to quantify an application locality –within a certain accuracy– is essential for any automatic approach. For example, an optimizing compiler needs to determine statically the trade-off among different optimizations for an adaptive processor-cache system, and it may need to choose a particular configuration among several ones (e.g., for optimal performance). A compiler must have a quantitative and precise measure of the benefits, so that it may choose the best solution without any further help from the developer.

This chapter is organized as follows. In Section 1.1, we state our motivation for the investigation and we summarize it by a thesis statement. In Section 1.2, we present the problem in detail and, thus, the compelling reasons for its solution and the effect of such a solution in its context. In Section 1.3, we outline briefly our contribution and our preliminary implementation as an investigative compiler aimed at the implementation of our solution. In this work, we show and enforce the potentiality of compiler-driven approaches such as the ones implemented in X-Legion compiler (pronounced the tenth-legion compiler) [1]

1.1 Thesis Statement

The X-Legion Compiler implements hybrid techniques, that is, software optimizations or

[1]The name of the compiler emerges from the components and their interaction. In fact, JuliusC is the first component and it stands for Julius Caesar; JuliusC analyzes the application solution strategies and it drives some of the optimizations at the highest level of abstraction. Indeed, in the Gaul's war the tenth legion was under the direct control of Julius Caesar (on the field), it was composed by the most loyal and experienced legionaries who showed the most stamina in action.

hardware adaptations based on compile-time analysis and run-time deployment based on static analysis. Such techniques are tailored to the optimization of recursive divide-and -conquer (D&C) matrix algorithms and they exploit data locality and use direct-mapped data caches.

The contribution of this thesis is based on the importance of the applications and on the importance of the cost/performance/power advantages of using simple data caches and new hybrid techniques.

In practice, matrix algorithms are the core of scientific computing and applied matrix algebra, therefore even a small improvement in one of the basic kernels can be extremely beneficial. The optimal implementation (i.e., with minimum number of cache misses) for a family of matrix algorithms is compelling for performance purpose and for the efficient utilization of complex (and often expensive) systems. We explore the extent of D&C matrix algorithms and their cache obliviousness achieving portable and optimal performance. Moreover, the data locality of these applications yields potential avenues for the automatic exploitation of coarse and fine grain of parallelism.

A simple cache design has the following effects on the trade-off among performance, cost and energy consumption:

- Large caches improve average data reuse and therefore performance, and, with the same cache size, a direct-mapped cache has faster hit access time than an associative cache. However, an associative cache may reduce cache interference, distributing interfering memory accesses to different blocks, reducing average access time.

  If we achieve the same cache miss ratio using a direct-mapped and an associative cache, we have better performance using direct-mapped caches only.

- With caches of the same size, direct-mapped caches dissipate less energy per access

than associative caches, because associative caches perform a search for data in every block, either in parallel or sequentially.

If we achieve the same cache miss ratio using a direct-mapped and an associative cache, we obtain better energy consumption using direct-mapped caches only.

- Modern processors have already embedded one or two levels of caches; these caches occupy the majority of the chip real estate. A direct-mapped data cache will simplify the design and the saved real estate may be used for other purposes.

- Multiprocessor systems may have thousands of processors and a small cost reduction per processor can be significant overall. A simple memory hierarchy translates into a simple architecture design and in low costs.

We evince our thesis as follows: We show how we can exploit locality by using: only code organization, only memory-hierarchy re-configuration, and a combination of the two. We explain how performance may vary as a function of the input sets and we show how adaptation of code and architecture is key to achieving optimal performance in any input scenario. We propose methods for the estimation of locality and we propose practical approaches for the quantification and exploitation of locality. We show the benefits of our approach by experimental results that are obtained as a combination of the following: direct measurement of performance (e.g. wall clock), by simulation (e.g. cache-processor simulators), and by hardware counters (e.g. MIPS R12K, UltraSparc II). We also discuss an interpretation of such metrics and their interdependencies.

1.2 The Problem

If we observe the unfolding of the computation of a D&C matrix algorithm and especially its division in subcomputations, we may measure the locality by counting directly the number of computations and memory accesses. In this way, for a given fixed application and its

execution, we may verify whether or not the computation has optimal locality performance rather quickly –e.g., optimal = the minimum number of cache accesses over the number of arithmetic computations. However, if we want to steer the algorithm division strategy and the computation so as to achieve optimal performance, we face an arduous task because this problem is reducible to the scheduling problem, which, in turn, is an NP-complete problem.

The definition of **optimum** is dependent on the context, and in general it is the minimization of a **goal function**. If the goal is to achieve optimal performance, the application must be reorganized so that it has minimum execution time or minimal number of cycles. When the goal is to achieve **optimal work**, the application must be reorganized in such a way that it has minimum number of (basic) operations. In the literature, the **cache miss ratio** –i.e., the ratio of the number of cache misses over the number of cache accesses– is the common measure for the locality of an application. When the goal is to achieve optimal locality, the application must have minimum cache miss ratio. Also the term **cache miss rate** is commonly used and it is defined as the miss ratio multiplied by 100, that is, the number of misses as the percentage of accesses. Finally, when the goal is the minimization of power consumption, we can show that the optimal implementation minimizes a weighted sum of all of the previous goal functions, such as number of cycles, number of instructions and number of cache hits and misses.

In modern architectures, the relation among these goal functions can be puzzling at best. For example, an optimal application for its work may not be optimal for performance or locality, or vice versa. In general, the choice to optimize an application for one goal function, instead of another, depends on the context. If the technology advancement continues, the CPU's speed will double every 18 months – Moore's law, caches and memory will be twofold faster and larger every two-three years, and batteries lasting power will double only every five years; we can see that caches, memory and battery will be crucial resources: caches will represent the performance bottleneck of an architecture and the battery power will limit

considerably mobile computing. In this work, we address these problems by estimating the effect of our techniques measuring three metrics of performance: execution time (i.e., direct measure of wall clock or synthesized measure of it as mega floating point operations per second MFLOPS), data cache miss rate (ratio), and, sometimes, power consumption.

The problem escalates in proportion and difficulty when we search for an optimal implementation for a family of goal functions (or, even more difficult, a family of applications for a family of goal functions); for example, we may want to find a single implementation of an algorithm that, unchanged, has optimal performance for several different architectures. Even though in general such a *champion* does not exist and we need to select one champion for one application and one architecture, we present examples of algorithms that, *practically unchanged*, achieve *nearly* optimal/predictable performance for several different architectures.

In this work, we introduce a family of applications that have optimal and inherent data locality. We also present techniques to reorganize this family of applications to exploit –for every input– both data locality and system resources fully. D&C algorithms exploit locality naturally. The division of the main computation in sub-computations exposes an independent set of data and computations, which can be divided further. The division process stops when the computation can be performed –efficiently. If sufficient resources are available, we may ply low and high levels of parallelism and we may reduce inter-processors communication. If a single processor is the host of the computation and the memory hierarchy has multiple levels of caches, the division of the problem in smaller problems is tremendously beneficial as well. In fact, we divide large problems, for which data must reside in slow storage devices (e.g., disk and tape), in more manageable problems that we solve as soon as we bring the data in main memory; if we divide the problem a level further, we can solve smaller problems as soon as the data is in cache.

Algorithms applying the D&C approach are ubiquitous for problems on volumes of data.

We present and investigate D&C algorithms for linear algebra for dense matrices [1, 2], and algorithms characterization for hierarchical memories [3, 4]. We work with the following examples of linear-algebra applications:

- **LU-Matrix Factorization** is used in non-iterative algorithms for the solution of systems of equations, such as $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ (matrix of coefficients) and $\mathbf{b}$ are known and $\mathbf{x}$ is unknown. The solution follows these steps: first, we determine two matrices $\mathbf{L}$ and $\mathbf{U}$ so that $\mathbf{LU} = \mathbf{A}$, where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{U}$ is an upper triangular matrix [5]. Thus, we have reduced the system to two simpler triangular systems. Second, we solve the system $\mathbf{Ly} = \mathbf{b}$ and and then $\mathbf{Ux} = \mathbf{y}$ so to compute the final solution $\mathbf{x}$.

  Notice that the solution of triangular systems (upper and lower), as well as the matrix factorization, have matrix multiplication as kernel computation, and, they also inherit the data locality properties of matrix multiply, [2] see the following point.

- **Matrix Multiplication** (**MM**), $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A}$, $\mathbf{C}$, $\mathbf{B}$ are square matrices of size $n \times n$, is kernel for basic matrix factorization algorithms and linear algebra applications [6, 7, 8]. There is a countless number of implementations for MM and we can distinguish them by their computational complexity. For example, if we start from the fastest to the slowest, we have: Coppersmith-Winograd's [9] with complexity $O(n^{2.31})$, Strassen's [10] with complexity $O(n^{\log 7})$, and the definition, ijk-matrix multiply with complexity $O(n^3)$.

  The latter, ijk-matrix multiply has been proved to achieve optimal cache locality [11], and, in fact, MM is one of the **cache oblivious algorithms**; that is, there is an implementation that, unchanged, is optimal (asymptotically) in the number of cache misses for an ideal cache [3] of any cache size.

[2] We use interchangeably matrix multiplication and matrix multiply

[3] I.e., fully associative cache with ideal optimal cache line replacement policy

- **Matrix by vector multiplication** is basic computation in BLAS 2 and, it is especially used for the change of representation of a vector using different space bases. In particular, the spectral analysis of digital signals, such as the application of **Discrete Fourier Transform** (DFT) on discrete digital signals, is commonly used for the design and implementation of discrete digital filters. When the discrete signal is considered **periodic** for a certain number of points (i.e., after a certain number of samples or **points** the signal has the same repetitive form) the DFT algorithm is amenable to be optimized: the algorithm is known as the **Fast Fourier Transform** (FFT). The most famous implementation of FFT it may be the FFTW[12]. Notice that FFT is another cache oblivious algorithm.

The inherent cache locality of linear-algebra applications make them an excellent test set where we are able to investigate optimizations aimed to exploit fully the potentials of these applications. To exploit locality and performance even further, matrix algorithms may require *ad-hoc* organization of the data matrices; in fact, the layout of matrices may speed up the transfer of data to/from different levels of the memory hierarchy (i.e., allowing fast data-block transfers), and it may reduce cache interference (i.e., especially self interference).

Aggarwal et al. [13] present the first clear performance model for memory hierarchy with block transfer. Their model aims to represent the communication cost in real systems, where in practice, all transfers among different level of the memory hierarchy are among blocks of data. In fact, all data transfers between disk-memory and memory-caches break down eventually to a transfer sequence of large blocks of continuous data. For a hard disk, the seek time –i.e., the time to move the hard-disk head on the wanted data– dominates the transfer time, because it is of the order of milliseconds; however, when the hard disk head is moved successfully on top of the first sector, the data transfer involves large data blocks (of up to 512KB each) and is extremely fast (GB/s). The transfer of data among caches and memory has similar spatial locality characteristics, but instead of large sectors,

the data transfer involves smaller data blocks, called lines, usually having sizes between 32B and 128B.

To return to the subject of how data layout may affect performance, the reorganization of matrix layouts changes how the computation accesses the data as well as how the data are mapped in cache during the computation. For example, a careful organization of the data into the hard disk can minimize data transfer latency and maximize the use of the data block at every memory level: for example, exploiting page locality in main memory, exploiting cache-line locality in direct-mapped and associative caches, and, finally, continuous loads/stores in registers.

Data layouts have been adopted in conjunction with blocked algorithms early in their design and application, but they have received particular attention for recursive algorithms. Indeed, these layouts are known as *recursive layouts* [14, 15, 16]. Recursive layouts have been applied first for surface-rendering problems with massive data sets, then proposed for standard linear-algebra applications. In the following, we introduce the most compelling reasons why recursive algorithms are attractive, and especially for code portability across architectures and for code analysis by compilers.

In this work also, we propose a new processor-cache system that allows a partial adaptation of the data-cache organization to the application requirements at run-time. Our processor-cache system does not need to change dramatically during the computation; the system morphs from one configuration to another by small steps and, in general, adapting the *functionality* without changing the hardware configuration. That is, we propose a direct-mapped cache (e.g., 32KB) that may change its cache line size (e.g., 8, 16, 32, 64 or 128B) or its mapping function (e.g., the memory address x is mapped to the cache line $f(x) = \ell_i$), but it will remain a direct-mapped cache of the same size (e.g., 32KB). We show that we can achieve optimal performance across different architectures, different cache sizes and cache organizations, because we make the application and the system work together for a common

goal. We do this by using compile-time techniques to reorganize the code application and annotating the application itself; the annotations are read and interpreted at run-time and the application-architecture adapts.

In our search for optimal performance, even when highly tuned or hand-coded applications show better performance on a specific architecture than our codes, we find a system configuration (i.e., software and hardware combination) that presents predictable performance for all input sizes. In a modern view of a system, the interaction between architecture and application is one component in a more complex equation. We believe that the predictable performance of an application is a crucial feature –especially for power management– because it allows the operating system, or the user, to plan in advance the further utilization of the architecture. For example, the operating system may plan to set to idle part of the system when the application has reached completion, thus, saving energy.

1.3 Thesis Contribution: The X-Legion Compiler

In the following, we introduce our novel compiler techniques for the analysis and optimizations of D&C algorithms. In fact, we discuss the implementation of these techniques in an investigative compiler that we call **X-Legion compiler** – the tenth-legion compiler. The main goal of this compiler is the analysis and model of D&C algorithms in such a way to exploit locality and software portability by addressing code optimizations and compiler-driven memory-hierarchy adaptations such as cache-line size and cache mapping for data caches.

The organization of the following section mirrors the organization of the thesis. In fact, we present each component of the experimental compiler X-Legion, in Figure 1.1. In short, we describe the dynamic of X-Legion as follows. A D&C input application, written in C, is transformed into an **intermediate representation** IR, which represents the entire program in a single hierarchical structure. Then, **JuliusC** (briefly introduced in Section 2.1 and fully described in Chapter 3) takes the hierarchical structure of the application and determines the

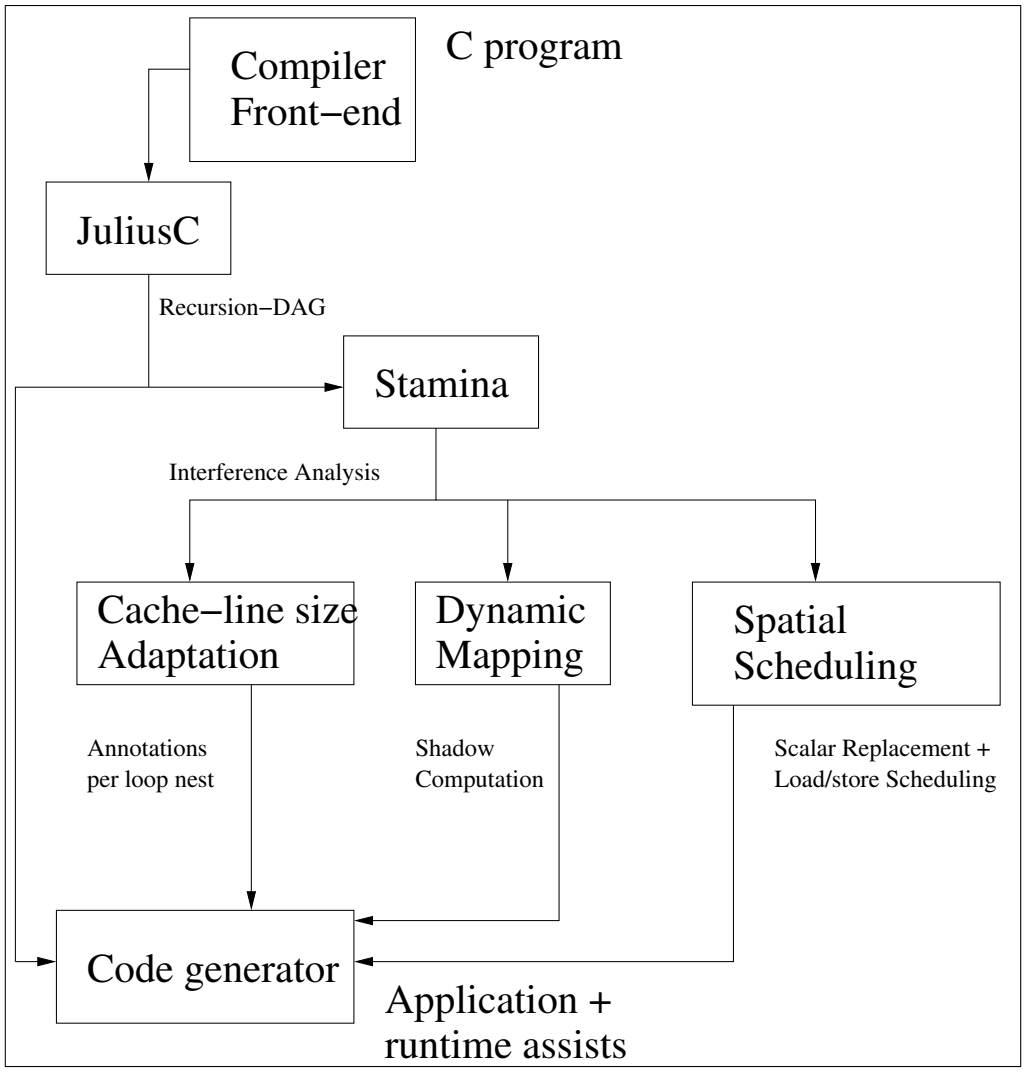

Figure 1.1: X-Legion Compiler

call graph of the program. In practice, JuliusC determines whether a function is recursive or a function is a computation leaf. The main two goals of JuliusC are: first, the analysis of the problem division and, especially, the division of a recursive D&C implemented in the input application; and, second, the concise representation of such a process by a **recursion-DAG**. A recursion-DAG is a direct acyclic graph that describes the unfolding a recursive function.

If this is possible at compile-time, the recursion-DAG can be used for the analysis of the **leaf computations** and their interaction. In fact using the recursion-DAG, we are able to

synthesize information about the hot spots of the application by computing the frequency of a function call. Also, the recursion-DAG can be used to collect statistics about the problem size of each function call and, thus, it can be used to reduce the complexity of specific analysis performed by **STAMINA**. So the recursion-DAG has two purposes: to expedite the application and the compilation execution time.

Otherwise, that is, if the recursion-DAG cannot be built at compile-time, we can reorganize the application so that we can generate the recursion-DAG at runtime and use it to speed up the execution of the recursive section/part of the application. We introduce this topic in Section 2.2 and we explain the approach in Chapter 4.

Independently, whether or not the recursion-DAG can be determined at compile time, STAMINA extracts the loop nests of the application and analyzes only the perfect loop nests, which are the most computation demanding (leaf computation). We give an introduction to STAMINA in Section 2.3 and we discuss the approach in Chapter 5. The goal of STAMINA is to analyze and quantify the cache interference per memory reference in the inner loop of a loop nest. Such an analysis drives the activation of three possible optimizations/adaptations: cache-line size, dynamic mapping and spatial scheduling. In fact, if major cache interference afflicts the leaf computations, then the leaf computation performance and the overall application performance will slowdown.

The result of the interference analysis, generated by STAMINA, is used to drive the data cache-line size –on a per loop nest basis– by introducing a special instruction (or annotation). In fact, a large cache-line size reduces the number of data fetches but it may increase cache interference; otherwise a short cache-line size may reduce cache interference, because it also changes the data cache mapping, but it increases the number of data fetches. We give an introduction in Section 2.3 and present the approach in Chapter 5.

An adaptation, which is orthogonal to adaptive cache-line size, is dynamic mapping. In fact, we may tailor the data cache mapping to the application and make the cache believe

that the memory addresses used to store and retrieve data in cache is different from what it is actually used in memory. We call this alternative memory space **shadow space** and the computation in the new space, **shadow computation**. This approach introduces extra computations, such as the ones introduced while padding matrices to reduce cache interferences [17]. The main differences are that no extra space is required, the extra computations are introduced only when interference is estimated heavy, and the speed-ups achievable overcome the overhead introduced by the extra computations.

At last, we may circumvent the effects of cache interference by a careful memory access scheduling and register allocation only. In practice, we propose **spatial scheduling**: this is a source-to-source transformation that, using scalar replacement, compacts consecutive in-memory accesses into a sequence of loads into scalar variables, thus into registers; this new load scheduling exploits fully spatial cache locality and it reduces the effect of cache interference. We give an introduction in Section 2.4 and present the approach in Chapter 6.

CHAPTER 2

Introduction and Related Work

The goal of this chapter is to introduce the thesis contribution and the related work in a fashion that is context sensitive. Indeed, here we introduce the components of the X-Legion compiler, Figure 2.1, and the related work, which is interdisciplinary in nature and comprises ideas ranging from algorithm engineering to data allocation to registers.

This chapter is organized as follows. In Section 2.1, we introduce JuliusC and our approach to extract information about D&C algorithms by using a data structure that we define as a recursion-DAG. In Section 2.2, we present four applications for which we used the recursion-DAG directly so to achieve an efficient implementation. In Section 2.3, we present the software package STAMINA for the analysis and determination of the best cache-line size. We present in the following sections how STAMINA's analysis is used. In practice, in Section 2.4, we present two techniques exploiting such an analysis: first, we propose a data allocation to registers to circumvent the effects of cache interference, secondly, we present a technique to tailor the cache mapping to the application's needs.

2.1 JuliusC and Recursive-DAG

With the introduction of D&C algorithms for the solution of matrix computations, for example, BLAS 3 [18], the performance of scientific applications has improved markedly. In fact, D&C computations expose spatial and temporal data locality; in turn, this data locality

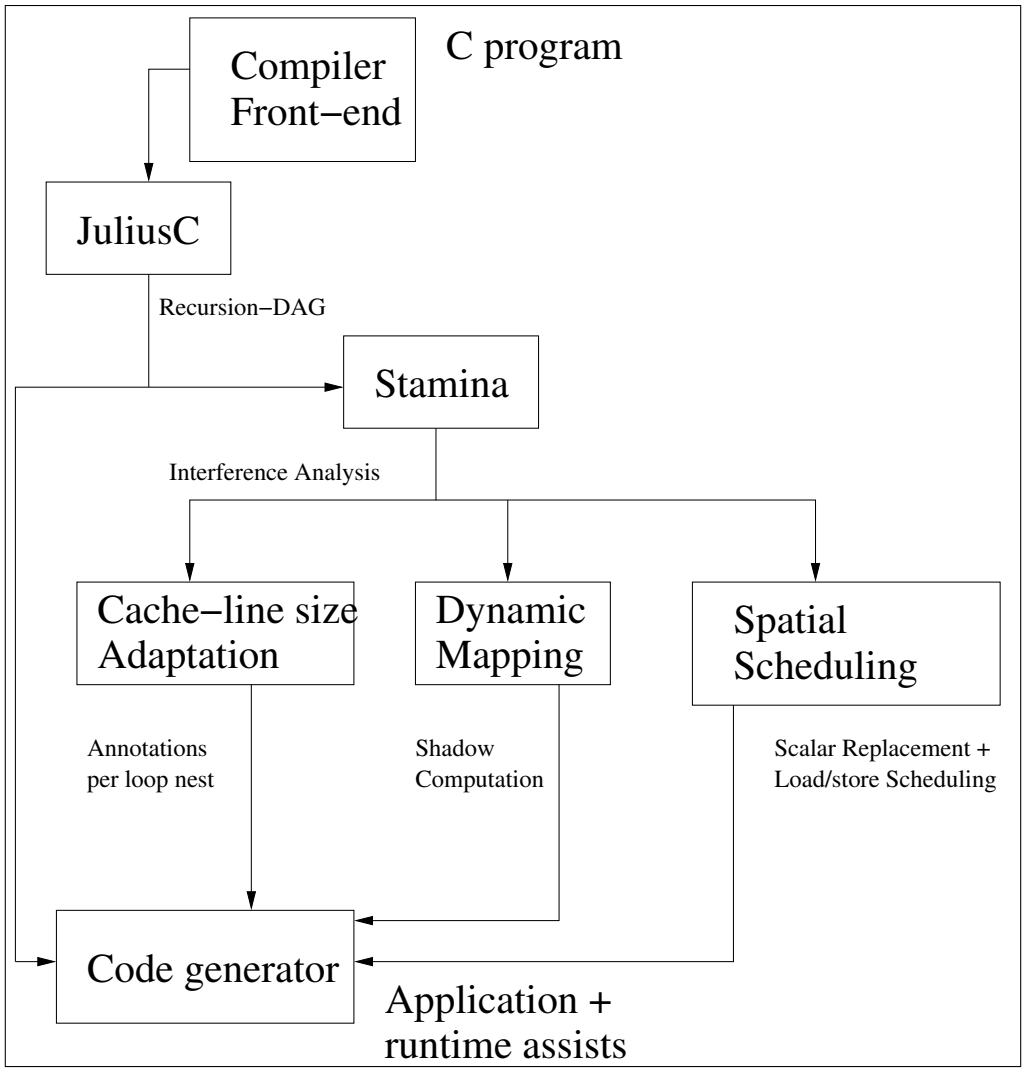

Figure 2.1: X-Legion Compiler

results in efficient utilization of resources on modern uniprocessor/multiprocessor systems; therefore, it results in extremely good performance. In practice, D&C algorithms can be *a priori* implemented using either of two language constructs: loop nests or recursion. From hereafter, we refer to the algorithms implemented using loop nests as **blocked**, and to those using recursion as **recursive** algorithms.

Though, most implementations are blocked algorithms [1, 8] and only a few implementations are recursive algorithms [6, 19]; however, recursive algorithms are appealing because of

their intuitive formulation and implementation. In fact, recursive algorithms are top-down solutions; they exploit data locality naturally at every cache level in architectures with deep memory hierarchies; they expose data dependency intuitively and, in turn, parallelize naturally along the recursive calls [20]. Indeed, most recursive matrix algorithms, such as matrix multiply and LU-factorization [5], are *cache oblivious* [21]. That is, they have optimal cache reuse at any cache level with no need for tuning –except for the register file [22] where some code tuning may further increase performance. Ultimately, recursive algorithms represent an efficient solution in a more abstract and intuitive format because the developer does not need to know the details of the architecture on which these applications will run.

Despite the above advantages, recursive algorithms have been considered impractical (for high-performance applications) for two reasons. First, a compiler has a difficult time optimizing at which point the recursion should stop (**leaf computation**). Thus, if only recursion is applied, the leaf computation may have too few instructions leaving little room for optimizations, exploiting little register reuse, and, especially, incurring a high overhead due to the (otherwise avoidable) proliferation of small recursive procedure calls. Second, any recursive algorithm *inherits* an overhead due to the **division process**, or partitioning, which is intrinsic in D&C algorithms, into small subproblems. This overhead is proportional to the number of function-recursion calls and to the work involved in computing the actual parameters of a function call.

Due to the importance of such computations, practical and efficient solutions of basic matrix computations have been the center of substantial efforts in terms of time and money in the last decades. As a result, ready-to-use and highly-tuned libraries for several systems have been proposed and used [7] and, more recently, self-installing and self-tuning libraries arose as new standards [8, 23] delivering astonishingly good performance. Briefly, self-installing and self-tuning libraries extract the parameters of an architecture and tailor the source codes to the architectures using exhaustive or tailored search techniques. These libraries are based on

blocked algorithms rather than recursive ones because of the mentioned recursive algorithm problems and for a number of other reasons: the first that comes to mind is code legacy, for example, FORTRAN has no-recursion; a second is the compactness of the code, thus the amenability to traditional aggressive compiler optimizations; and, finally, the most famous reasons are the negligible overhead introduced by loops instead of recursive calls, and ability to tile a loop nest so as to exploit temporal locality of references [24].

However, with the explosive increase in the complexity of systems due to the rapid advances in semiconductor technology, the number of parameters describing a system also increases. This undermines the portability and efficiency of current self-tuning high-performance applications. The complexity of probing the parameters as well as the complexity of tailoring the source code to an architecture will become impractical –if not quixotic. As a result, recursive algorithms for matrix computations should become increasingly appealing, if only they can be made more efficient by addressing their inherent problems as mentioned previously.

Several authors, among them us, target the solution of the recursive algorithm problems proposing applications based on both loop nests and recursion [25, 22, 12]. A recursive algorithm implements the division process, exploiting the *oblivious* data-cache locality. A blocked algorithm implements the leaf computation, exploiting aggressive compiler optimizations. These authors propose to stop the recursion at a certain level and then call high-performance non-recursive routines. This is also known as **pruning**. Pruning reduces the number of function calls, and therefore it reduces the overhead; however, it does not reduce the work per function call; that is, the computation of the actual parameters for every function call.

We propose a novel technique for greatly improving efficiency of D&C algorithms. We introduce the concept of a DAG data structure, **recursion-DAG**, to model recursive algorithms; for example, using a hand-coded implementation of the recursion-DAG, we were able to reduce the integer computation –index computation to access matrix elements– by 30% in

matrix multiplication, see Section 4.2, on page 77, [22]. In practice, we propose an automatic approach for the determination of the recursion-DAG of general recursive algorithms and we have four main contributions:

1. In Chapter 3, we present **JuliusC** [26], a (lite) C compiler. We present techniques for the determination of a recursion-DAG, and provide algorithms to fully automate it. We also show that the approach has practical time and space complexity.

2. We present techniques to model the run-time unfolding of recursive algorithms concisely and to abstractly represent function calls with the same division work as a single node in a recursion-DAG.

   The model is a concise and precise representation of the computation and we envision its application as support to drive further compiler techniques/optimizations such as dynamic cache mapping [27] and automatic parallelization, for example, using sophisticated abstract data description [28] and in combination with parallelizing techniques/compilers [29].

3. We show how the system embodying the above techniques can be useful as an analysis tool for performance evaluation purposes, for software design purposes and also for debugging purposes.

4. To illustrate our techniques and provide a feel for the improvements achievable, we show that we can reduce the division work by 14 or more times for the computation of the binomial coefficients –actually, the original exponential execution time becomes polynomial– and by 20 million times for all-pair shortest path of an adjacency matrix (of size 7500×7500). We show that the division-work reduction is a function of: the problem size, the algorithm and the pruning technique.

Notice that in parallel applications on large data sets, the division process constitutes

the main overhead and it may be driven by a single processor in a multiprocessor system. A recursion-DAG may be used to reduce the division work, therefore speeding up the sequential execution of the problem division. Also a recursion-DAG may assist the parallelization process helping the efficient processor allocation to parallel function calls at run-time; for example, it may help reduce the number of process spawns, thus, initialization time and initialization communications.

2.1.1 Recursive-DAG, Related Work

Our approach has several similarities with approaches in two very active research areas: one is *self-applicable partial evaluation* and the other is *dynamic programming*.

Self-applicable partial evaluation is the problem of optimizing an algorithm when a partial number of arguments is known at compile time and, therefore, specializing a function, or the entire program, to partially precompute the result to the extent feasible at compile time. For references, see [30, 31] or a survey by Jones et al. [32]. Object-oriented class templates, and in particular function specialization in C++, are familiar examples of partial evaluations. Some general purpose compilers also apply partial evaluation for optimizations such as dead code elimination [33]. All authors propose an approach composed of two phases: a static phase and a dynamic phase. The static phase determines what can be computed at compile time, and what cannot be computed, the **residual**.

Our approach is also a combination of static and dynamic phases (or analyses). The static analysis takes the input program and annotates the formals parameters of function definitions so as to clearly mark the formals involved in the recursive division process only. [1] The dynamic analysis partially executes the program, and using the annotations, binds

[1] Notice that other algorithms may deploy a few global variables to keep record of the decomposition process and they may use no auxiliary data structures such as a stack. First, this is not an example of *good* programming because the behavior of a function is not dictated by the values assigned to its interface. Second, and more importantly, all the problems presented in this work have a division work that cannot be executed correctly using only a fixed number of a few global variables.

function calls to nodes in the recursion DAG. When no argument values are known at compile time previous techniques are not applicable; however, our technique is applicable and enables us to precompute (at run time) data that will be used during the recursive algorithm execution; in practice, we *specialize* the computation of the recursive algorithm at run time.

In **dynamic programming**, there are basically two *philosophies*: function caching and incrementalization. **Function caching** is a top-down approach to solve a problem remembering the result of a function invocation and reusing this result when the function is invoked again with the same arguments. For example, Fibonacci number [2] $F(4)$ involves the computation of $F(2) + F(1) + F(2)$. With function caching, the first invocation of $F(2)$ is computed once, and the result is reused for the second invocation [34, 35, 36, 37]. While function caching may require an interpretive overhead to manage and store partial results, **incrementalization** is a bottom-up approach proposed to reduce or annihilate the overhead typical in function caching [38] reducing the working set of cached function results to a minimum –at any time– and having a fast access to them.

In general, dynamic-programming approaches are not applicable for matrix algorithms, because the final result of matrix algorithms is highly dependent on the contents of the input matrices (making reuse impossible). However, matrix algorithms have such regularity in their computation that some of the dynamic-programming techniques may be applied, at least in principle. In fact, our technique borrows the same basic ideas of function caching, especially in generating the recursion-DAG. Actually, we may see the recursion-DAG as the collection of all cached function-call results organized in a DAG. (However, we do not necessarily store any function return value). In this scenario, incrementalization is no better than function caching at the expense of a more complex design and implementation.

In addition, our approach is not meant for the conversion of loop nests to recursion

[2] $F(n) = F(n-1) + F(n-2)$.

[39] or vice versa [40], but it offers a means to speed up the conversion process and a quantitative measure of the overhead due to recursion. A compiler may then be able to make an informed choice between recursion or blocked instantiation. Our approach is not meant to optimize the function call mechanism (e.g., register minimization, solving allocation of the actuals-formals, inlining or recursion elimination, state elimination) because modern compiler technology already exists for these tasks and it is orthogonal to (and it could be use in combination with) our approach.

2.2 Recursive D&C Algorithms: Examples

We applied the concept of the recursion-DAG to four recursive algorithms. In practice, we analyzed the original recursive algorithm, we then determined the best division process, and we reorganized the algorithm in a such a way to use the recursion-DAG as guide-line for the division process and reduction of the division work. In these following sections, we introduce the applications investigated and the related work; in Chapter 4, we present our complete investigation.

We investigate a representative set of recursive algorithms to achieve the following goals: we show that recursion-DAG has practical uses for the developer and, especially, for design and implementation of linear algebra applications as recursive algorithms. As another result of our codes implementation, we show the positive effects of a balanced division process on performance for recursive algorithms: Indeed, a balanced division process produces an algorithm with a *more* predictable performance, a smaller overhead and a significative reduction of the division work.

In this thesis, we have turned our attention to the implementation features of LU-factorization and matrix multiply. In practice, we shall present an introduction here in Section 2.2.1 and 2.2.2, however we shall present a more detailed investigation in Section 4.1 and 4.2. We have also found that matrix multiply is also a basic computation in an

important application in graphs theory, **all-pair shortest paths** (APSP). We shall present an introduction in Section 2.2.3 and, in detail, our approach and results in Section 4.4.

2.2.1 LU-factorization

The first example of linear algebra application, where we used the recursion-DAG to model and to drive the computation, is LU-factorization with partial pivoting of non-singular matrices. The *beauty* of the LU-factorization with partial pivoting lies into its elegant blocked description [2, 5, 41]. In practice, matrix factorization is used to solve linear, dense and large systems as well as for the determination of the *rank* of a matrix; that is the number of independent rows/columns of a matrix. For example, in a system $\mathbf{AX} = \mathbf{B}$, where $\mathbf{A}$ and $\mathbf{B}$ are constant matrices and $\mathbf{X}$ is a matrix of free variables, we factorize matrix $\mathbf{A}$ as the product of one lower triangular matrix $\mathbf{L}$ by one upper triangular matrix $\mathbf{U}$, that is, $\mathbf{A} = \mathbf{LU}$. So the system is reduced to two easier-to-solve triangular systems: $\mathbf{LY} = \mathbf{B}$ and $\mathbf{UX} = \mathbf{Y}$. This approach is computationally easier than the direct computation of the matrix inversion $\mathbf{U}^{-1}$ and, in practice, it has also better numerical stability.

In practice, **partial pivoting** is a simple and efficient technique adopted to achieve numerical stability for most practical cases. Basically, one column of the matrix at a time, we seek for the matrix element with maximum absolute value in the leading column. We permute the matrix rows in order to set this matrix element as top element in the leading column. Finally, we execute the **Gaussian elimination** in such a way to reduce the element below the top element in the leading column to zero. We repeat the process until all elements below the major diagonal are reduced to zero. Partial pivoting assures a bounded value to the components of the lower triangular matrix $\mathbf{L}$ and, thus, a bounded error. Unfortunately, there are ill conditioned matrices, for which partial pivoting is not numerically stable, and the components of the factor $\mathbf{U}$ can become arbitrary large. To assure numerical stability for both matrix factors $\mathbf{L}$ and $\mathbf{U}$, we may deploy a **complete pivoting**: it is a technique to

seek the element with the maximum in absolute value of the leading sub-matrix and perform both rows and columns permutation in order to have this element as a new pivot for the Gaussian elimination.

Complete and partial pivoting differ by their numerical properties and by their computational complexities. For example, consider a square matrix $\mathbf{A}$ of size $m \times m$, then the application of partial pivoting during the entire factorization algorithm takes $O(m^2)$ steps and, in contrast, complete pivoting takes $O(m^3)$ steps –which is dominant for any implementation of the LU-factorization $O(m^3)$, [2, 42, 5]. Notice that matrix permutations involve data movements without computation and, in general, stored in slow components (e.g., memory) and, in practice, a two-row permutation may slow down the factorization computation noticeably [43, 44].

In Section 4.1, we shall show how we implemented Toledo's algorithm [5] using non-standard layout [14, 15]. In short, LU-factorization is the composition of three basic matrix computations: a lower triangular system solver, $\mathbf{LX} = \mathbf{B}$, an upper triangular system solver, $\mathbf{UX} = \mathbf{B}$ and a matrix multiplication. In turn, triangular system solvers are based on matrix multiply as basic kernel, Section 4.2.

We have three major contributions:

1. We explore the implementation issues involved in the computation of LU factorization for any square matrices stored in non-standard format, that is, Z-Morton [15].

   (a) We present algorithms for efficient row permutation and

   (b) We present algorithms for efficient column access of matrices stored in non-standard format.

2. We present how a matrix multiplication, even if designed only for square matrices, can be applied for the matrix multiplications among rectangular matrices involved in the LU-factorization.

3. We show that the recursion-DAG for LU-factorization is a sub-DAG of the recursion-DAG for matrix multiply.

We tested the performance of our code by cache simulations and by measure of the execution time. In fact, we have simulated the cache performance of our algorithm for seven architectures, such as SPARC 1 and Alpha 21164, and we show that our codes have high data and instruction locality. We also measure the execution time of our codes for four different systems and we show that we achieve good performance in comparison to, and often better performance than, hand-coded/tuned codes such as in SunPerformance library and ATLAS.

2.2.2 Matrix Multiply (MM)

In practice, the memory hierarchy helps performance only if the computation exhibits data and code locality. So algorithm design and compiler optimization increasingly need to take into account data locality.

An early paper by Aggarwal et al. [45] introduced the Hierarchical Memory Model (HMM) of computation, as a basis to design and evaluate memory efficient algorithms, then extended [46, 47]. In this model, the time to access a location x is a function $f(x)$; the authors observe that optimal algorithms are achieved for a large family of functions f. More recently, similar results have been obtained for a different model, with automatically managed caches [21]. The optimality is established by deriving a lower bound to the **access complexity** $Q(S)$, which is the number of accesses that necessarily miss any given set of S memory locations. Lower bounds techniques were pioneered in the early 80s [11] and recently extended by Bilardi et al. [48, 49]; these techniques are crucial to establish the existence of portable implementations for some algorithms, such as **matrix multiplication** (MM).

The question whether or not arbitrary computations admit optimal and portable implementations has been investigated by Bilardi et al. [50]. Though the answer is generally negative, however the computations that admit portable and optimal implementations in-

clude relevant classes such as linear algebra kernels [51, 18].

In this section, we turn our attention on MM algorithms with complexity $O(n^3)$, rather than $O(n^{\log_2 7})$ [10] or $O(n^{2.376})$ [9], and in particular we study the effect on performance of data layout with respect to: latency hiding, register allocation, instruction scheduling, instruction parallelism and their interdependencies. For examples of projects implementing libraries for linear algebra using optimized MM algorithms see ATLAS, PHiPAC, ESSL and others [8, 23, 52, 44, 53].

For compiler techniques exploiting locality used for linear algebra kernels, we may cite the loop transformations such as tiling [54, 55, 24, 56], the data dependency theory introduced by Banerjee for loop nests [57] and the effects of copy techniques to reduce interference [58]. The interdependence between tiling and sizes of caches is probably the most investigated in the literature [24, 17, 54, 59, 58, 44].

For example, vendor libraries (such as BLAS from SGI and SUN) exploit their knowledge of the destination platform and they offer efficient routines, but these routines loose optimality when used across different platforms. Automatically tuned packages, such as ATLAS and PHiPAC [8, 23] for MM, and FFTW [12] for FFT, measure the **machine parameters** by micro-benchmarking and then produce machine tuned code. This approach achieves optimal performance and *portability* at the level of package, rather than the actual application code.

In contrast, another approach defined as **auto-blocking**, has the potential to yield portable performance for the individual code. Intuitively, one can think of a tile whose size is not determined by any *a priori* information but arises automatically from a recursive decomposition of the problem. This approach has been advocated in [60], with applications to LAPACK, and its asymptotic optimality is discussed in [21]. Our algorithms belong to this framework because it is a recursive algorithm.

Recursion-based algorithms often exploit various features of non-standard layouts, **recursive layouts**, [15, 14, 61], and algorithms on recursive arrays layouts [6, 62, 25, 16]).

However, conversion from and to standard layouts (i.e., row-major and column-major) introduces $O(n^2)$ overhead, which is negligible, except for matrices small enough for the n^2/n^3 ratio to be significant, or large enough to require disk access.

Our approach, hereafter **fractal approach**, combines a number of known ideas and techniques, as well as some novel ones to achieve the following results.

1. We show that a single MM implementation exists and it achieves excellent and portable cache performance, and we prove it using simulation cache tools and collecting experimental results for seven different systems.

   - We show that the overall performance (FLOPS) is competitive by comparison with either the upper bound implied by peak performance or the best known code ATLAS [8].
   - We show that at least on one system, R5000 IP32, our approach yields the fastest known algorithm.

2. We show in Sections 4.2.1 and 4.2.1 how to apply the recursion-DAG to lead to efficient implementations of recursive procedures.

3. However, we do not consider and discuss any numerical stability problems (Lemma 2.4.1 [2], Lemma 3.4 in [42]). Nonetheless, we can assume that fractal approach has the same numerical stability than other approaches proposed in the literature because we do not change the order or the type of the multiplications and additions.

2.2.3 All Pair Shortest Path (APSP)

The **all-pair shortest-paths problem** (APSP) is a well studied and basic problem in graph theory but it is also a crucial and real problem in large networks such as sensor networks, switch networks or complex targeting systems.

Consider the scenario where many thousands of nodes are located across a large area and every node has a processor with little memory space and computational power. In this scenario, the computation of APSP is neither feasible nor practical by a single node, nonetheless it is a key feature for the efficient data routing and broadcasting. Despite the node-processor computational/memory limitations, a node in the network is able to determine the locations and distances of its neighbors rather easily. Such a local information can be coded, sent on the network and collected by an observer node such as a satellite, a global router or a computer cluster. Then, the observer node may construct the adjacent matrix, compute the solution and send the result back on the network where each node will store the necessary local information.

Any network is naturally represented by a **directed graph** and we formalize the APSP as follows. Given a graph $G = (V, E)$ where V is a set of nodes and E is a set of directed edges, we label every node in the graph by an integer $\iota \in [0, n-1]$ where $n = |V|$ ($n = |V|$ is the **cardinality** of the set V), and an edge in E is defined by a unique ordered pair of integers (i, j) with $i, j \in [0, n-1]$. In fact, we assume that there is at most one directed edge connecting two nodes and therefore, the graph has no more than $|E| = m <= n^2$ edges. To represent G, we use an adjacent matrix, $\mathbf{A} \in \mathbb{Z}^{n \times n}$. That is, an entry in row i and column j in matrix $\mathbf{A}$, $a_{i,j} \in \mathbb{Z}$, stands for the cost to reach node j from node i through the edge $(i, j) \in E$. Also, we assume that $a_{i,i} = 0$ for every $i \in [0, n-1]$ –i.e., there is no cost to stay in one node– and, if there is no direct edge from node i to node j, then $a_{i,j} = \infty$. An **elementary path** [63] of length k from node i to j in a graph G, is a sequence of k edges connecting i to j with no nodes repeated. In fact, we denote an elementary path as the set of edges $\mathbf{P_k(i,j)}$. The **cost of an elementary path** $P_k(i, j)$ is denoted as $\mathbf{C[P_k(i,j)]}$ and it is equal to $\sum_{\ell=0}^{k-1} a_{\iota_\ell, \iota_{\ell+1}}$, where $(\iota_\ell, \iota_{\ell+1}) \in P_k(i, j)$. Thus, the solution to the APSP problem is the **matrix closure** $\mathbf{A}^*$ such as for all $i, j \in [0, n-1]$ the matrix element $a^*_{i,j}$ is $\min_{k \in [0, n-1]} C[P_k(i, j)]$.

In this section, we propose a practical recursive D&C algorithm inspired by Kleene's algorithm [64], which is optimal in the number of addition–comparison operations and memory accesses, $\Theta(n^3)$. Indeed, fast MM algorithms [10, 9] that could speed up performance as proposed by Zwick [65] are not applicable, because the adjacent-matrix entries have no constraint and can be any positive or negative integer –as we shall explain in Section 2.2.3. Though, our approach is based on MM, however it is independent of the algorithm used to code MM. So we may apply R-Kleene in combination with fast-MM –see Section 4.4.1– but this is beyond the scope of this preliminary investigation. We have two major contributions and we summarize them in the following.

- First, we formulate our algorithm as a recursive MM where the result is computed in-place for dense adjacent matrices. As such, we are able to replicate the classic properties of MM, such as performance, space, I/O complexity and register utilization. In fact, Kleene's algorithm and Floyd-Warshall algorithm [64, 66, 67] impose a specific computation order exploiting little parallelism and register reuse. In contrast, with the same number of comparison-addition operations, R-Kleene leads to an in-place implementation that yields an efficient register utilization and exposes a larger number of independent operations, and, in turn, better performance. (This idea can be taken a step further and we may apply the same optimization and fine tuning used by software libraries such as ATLAS [8]; but this is beyond the scope of this work.)

- Second, we present a quantitative measure of R-Kleene performance using row-major and Z-Morton data layout [15], and we present an upper bound to the performance achievable by a recursive algorithm such as R-Kleene. We achieve this by collecting experimental results for our algorithm and three other representative algorithms on five different systems and for adjacent matrices of sizes ranging from 200 to 5,000. In fact, we show that R-Kleene achieves good, predictable, scalable and portable performance.

APSP Related Work

The literature available for APSP and its solutions is copious and, for sake of explanation, we may distinguish four main categories: APSP algorithms for dense graphs, for sparse graphs, for static networks and dynamic networks (changing in time).

For a general overview on static approaches, Zwick [68] presents a complete survey on algorithms computing the exact distances in graphs and the author also discusses the *lingering open problems* in the topic. We may notice that the most efficient algorithms discussed in the survey are based on Dijkstra's algorithm [69]. To gain a feel about the algorithms complexity, if we count the **comparison–addition** operations, *compadd* for short, as basic operation, then Dijkstra's algorithms perform $O(mn) \leq O(n^3)$ *compadd*s.

Our original contribution is a static solution of the APSP problem for dense graphs and Floyd-Warshall algorithm [66, 67] was our starting point. This algorithm has the same complexity as Dijkstra's (i.e., $O(n^3)$) but it is often preferred for its *practical* performance for dense adjacent matrices [63]. In practice, Floyd-Warshall algorithm is an in-place algorithm that constructs the shortest paths during the computation using a clever dynamic-programming approach. To exploit data locality, the algorithm can be reorganized as Kleene's algorithm [64], which can be seen as the blocked Floyd-Warshall algorithm, where the classic MM is used as basic routine. Notice that if the adjacent matrix has values that belong to a finite and small set, the domain where APSP is defined, a semiring, can be extended to a ring. Thus, we may use fast MMs [10, 9] for the solution of APSP as proposed by Zwick [65]. However, in the scenario assumed throughout this work, we are not able to extend the APSP domain to a ring and, in turn, we cannot use fast MM.

Our work is similar to the work by Park et al. and Penner et al. [19, 70], because we investigate the performance for APSP algorithms on dense adjacent matrices using different data layouts. However, our algorithm is 100% faster because it utilizes more efficiently data in registers, reducing the number of memory accesses and, thus, improving performance.

In this work, we propose an algorithm that is oblivious of the graph structure and, therefore, it performs n^2 *compadd* per node (but it is still optimal for dense and generic graphs). This approach seems less efficient than the algorithms presented by Cherkassky et al. [71]. In fact, Cherkassky et al. show that algorithms based on Dijkstra's algorithm achieve an average of 1 *compadd* per node because applied to a representative set of sparse graphs. This is a large work difference obviously. However, we shall show in Section 4.4.1, that our algorithm can be parallelized naturally, and used efficiently in multiprocessor systems, offering an appealing performance edge.

To conclude the review on related work, we may notice that all of the previous approaches are static solutions, because they assume the network is fixed and unchangeable. However, in the scenario where nodes can be introduced dynamically, and the APSP solution must be dynamically updated as well, then the solution must be dynamic as well; for example of dynamic approaches see Demetrescu et al. [72, 73].

2.3 Compiler-Driven Cache-Line Size Adaption

Caches are crucial components of modern processors; they allow high-performance processors to access data fast and, due to their small sizes, they enable low-power processors to save energy - by circumventing memory accesses. We examine efficient utilization of data caches in an adaptive memory hierarchy. We exploit data reuse through the static analysis of cache-line size adaptivity. We present an approach that enables the quantification of data misses with respect to cache-line size at compile-time using (parametric) equations, which model interference. Our approach aims at the analysis of perfect loop nests in scientific applications, it is applied to direct mapped cache and it is an extension and generalization of the Cache Miss Equation (CME) proposed by Ghosh et al. (1999). Part of this analysis is implemented in a software package STAMINA. We present analytical results in comparison with simulation-based methods and we show evidence of both expressiveness and practicability of

the analysis.

In modern uniprocessor systems, the memory hierarchy is an important concern for performance, area and energy. It is also the component requiring most of the die area in systems-on-chip and it is the principal power consumer, accounting for as much as 20-50% of the total chip power [74, 75]. In recent years, there has been a great endeavor to engineer several levels of cache for the exploitation of performance and power. In particular, we have studied the effect of adaptivity in cache subsystems and we have built an architecture as prototype that enables static and dynamic adaptation of memory hierarchy: its configuration and policies [76]. In this section, we turn our attention to (compiler-driven) cache-line size adaptation of direct mapped data caches [77, 76]. In fact, the architecture changes the cache-line size dynamically (by hardware monitoring or application instruction) during the execution of the application. To exploit fully the potential of this adaptation, we need a way to target it; that is, (statically) determine the application cache behavior to trace adaptation for maximum performance and minimum energy dissipation. The related work on cache behavior analysis can be distinguished in profiling-based and static approaches.

Profiling is an approach that uses the direct measure of performance as feed-back to drive the fine-tuning of some architecture parameters. The main goal is to improve performance of an application when applied on a representative input [78]. The approach is flexible and it can be used for the analysis of the whole application as well as part of it. However, profiling has two limitations: the performance of an application is often dependent on the inputs and, of course, the analysis cannot be faster than the execution of the application itself.

Static approaches are basically independent of the *inputs* and, thus, the analysis can be performed just once at compile time. In particular, static approaches analyze mostly perfect loop nests and these loop nests are ubiquitous in scientific applications. (As reported by Ghosh et al. [79], 244 loop nests are statically analyzable, 289 are parameterized loop nests and 189 are not analyzable - Table I, page 707 - for SPECfp 95 benchmarks.) In fact,

static approaches model data-cache misses of a memory reference in a perfect loop nest by using **cache miss equations** (CME) [79]. When the CMEs are defined for a given memory reference and a loop nest, every iteration in the loop nest (or a sampled version such as in [80, 81]) is checked as whether or not it satisfies the equations. If an iteration satisfies the equations, then the memory reference has a cache miss at that particular iteration. Thus, the approaches count the solutions of the equations to achieve an estimation of the number of cache misses. As an extension of this idea, Vera and Xue [82] propose an approach to analyze the whole program based on their cache-miss solver developed by the same group [81]. For parameterized loop nests, the authors (both Ghosh et al. and Vera et al.) suggest that the approach can be applied at run time in similar fashion, because the parameters are known. However, there are two limitations in the current static approaches. First, the loop nest bounds must be known at compile time. This is not realistic (e.g. 289 loop nest in SPECfp) because they are often parameterized. Also, even if the analysis is performed at run time, it may be impractical, because these loop nests can be very large. Second, the analyzable loops are *sensitive* to tiling loop transformation. For example, if tiling is performed on the three-loop-algorithm for MM and the tile sizes do not divide evenly the loop bounds, the inner loops bounds cannot be represented by affine functions. The resulting nest is not analyzable.

To attack and overcome these limitations, we propose a static approach to investigate perfect (parameterized) loop nests and to determine the relation between cache-line size and number of misses on a per-nest-base for a direct-mapped cache. The analysis result is annotated in the code and it can be used at run time to set the line size. The approach is especially suitable for applications with references having reuse within few iterations in the inner loop and exploiting spatial locality [83].

A well known paper on optimizing for data locality and parallelism exploitation, is by McKinley and Kennedy [84]. In practice, they assume that there is little or no interference

for a small number of iterations in the innermost loop. Spatial locality is exploited, if any is available in the inner loop, under the assumption that cache misses are independent of any interference. The authors propose different loop optimizations (i.e. loop permutations), to exploit maximum spatial and temporal locality in the innermost loop. The examples presented in this work and in particular in Chapter 5, as many other loop nests in real applications, do not satisfy McKinley and Kennedy's assumption. Cache interference can be the major contributor of cache misses in inner loops. Instead, our approach considers such interference and, in practice, the two approaches are *orthogonal*.

2.3.1 An Example and Related Work

In this section, we present the novel contribution of our approach using a simple example. We break down the problem and the solution –as our approach does– in order to present the following three points: first, the challenges that current analysis tools face determining data cache misses; second, the terminology that is adopted in our framework; third, a quantitative and informal application of our approach - we shall see a rigorous notation and analysis in Section 5.1 and 5.2.

Consider the example shown in Fig. 2.2. The two memory references $A[i][j + start]$ and $B[i][j]$ in the inner loop body are affine functions of the loop indices; that is, the indices i and j. The indices are represented as a vector $\mathbf{k} = (i, j)^t$: the first entry is the outermost index, the second entry is the innermost index. A particular iteration of the loop nest is simply identified by $\mathbf{k}_0 = (i_0, j_0)^t$.

The memory references exploit spatial reuse in the inner loop. Two consecutive accesses to matrix A (i.e., $A[i][j + start]$ and $A[i][j + 1 + start]$) and to matrix B tend to exploit spatial locality. We describe this reuse property by the vector $\mathbf{r} = (0, 1)^t$. The reuse vector is relative to an iteration; that is, the cache line read at iteration $(i_0, j_0)^t$ will be read again at the next iteration $(i_0, j_0+1)^t$ (i.e., $\mathbf{r} = (i_0, j_0+1)^t - (i_0, j_0)^t$). Note that the reuse depends

```
extern double A[2000][1024],B[100][1024];

void foo(int m, int start) {
  int i,j;
  for (i=0;i<m;i++)                /* 0<=m<100 */
    for (j=0;j<m;j++)
      A[i][j+start] += B[i][j];  /* 0<= start <1024-100 */
}
void update(int start) {
  int start1=0; /* compile time */
  int start2;
  int startin;

  start2 = start+2; /* not really at compile time */
  foo(50,start1);
  foo(50,start2);

  scanf(''\%d'',\&startin); /* run time */
  foo(50,startin);
}
```

Figure 2.2: Parameterized loop bounds and index computation, thus interference.

on no parameters.

When the two references of matrices A and B at an iteration $(i_0, j_0)^t$ are mapped to the same cache line, there is interference in cache. The cache interference prevents the spatial reuse as the same line may be reloaded. Without cache interference, we can estimate the number of cache misses as $2m^2/\ell$, where $\ell = L/8$ is the number of double precision float numbers in a cache line, L cache-line size.

Let us consider the order of memory accesses in the loop body as follows: a read of A precedes a read of B, which precedes a write of A. The read of A has spatial reuse $(0, 1)^t$ and temporal reuse $(0, 0)^t$. The read of B has spatial reuse only. The temporal and spatial reuse of A is not exploited when the access to B is mapped to the same cache line; in other words, when the address of $B[i][j]$ is the address of $A[i][j + start]$ plus a multiple of the cache size and an offset no larger than the cache-line size at the iteration specified by $\mathbf{k} = (i, j)^t$ (where $0 \leq i, j < m$). We model interference by the following equation: $B_{-1} + 8192i + 8j = B_{-1} + 16384000 + 8192i + 8j + 8start + n\mathcal{C} + q$. The constant B_{-1} is

the start addresses of B; the constant $C = 16 * 1024$ is the cache size; the variable n has positive integer values and q has integer value so that $|q| < L$. We simplify the equation to: $16384000 - 8start = nC + q$. The set of inequalities defines a parameterized *polyhedron*. Notice that static approaches based on the one proposed by Ghosh at el. are not practical for large polyhedra, because the analysis must be repeated for each parameter value.

When $8start$ **mod** $\mathcal{C} < L$, we have a solution (e.g., for $n = 1000$ and $q = 8start$ **mod** $\mathcal{C}$). The solution of the equation stands for a cache interference. The interference prevents the cache-line reuse, and we have a cache miss.

Note that the optimal cache-line size and the number of cache misses are a function of the parameter *start*. The optimal line size is $L_{opt} = 8start$ **mod** $\mathcal{C}$ (i.e., no cache interference). Notice that Polylib achieves an equivalent result. The number of cache misses is $M = 2m^2(L - \Delta)/L$; the term m^2 specifies the number of iterations; the constant 2 is the number of references we analyze; the last term $(L - \Delta)/L$, where $\Delta = 8start$ **mod** $\mathcal{C}$, specifies the fraction of accesses that effect cache misses caused by cache-line underutilization. Notice that profiling approaches use a black-box approach about the application, therefore they should test all possible values of *start* just to be confident of the performance measurements.

Now consider matrix $B[100][512]$ instead of $B[100][1024]$. The interference equation is $16384000 - 4096i = 8start + n\mathcal{C} + q$. When i is 0, it is the previous equation. Both memory references interfere in cache for the first m iterations, when $8start$ **mod** $\mathcal{C} < L$. For $i = 1$ and for the same values of *start*, there is no interference. Indeed, we have interference every four iterations of i. We define this ratio as *interference density*, denoted by $\rho = \frac{512*8}{C} = 1/4$. In the presence of cache interference, the number of cache misses is $2m^2\rho\frac{(L-\Delta)}{L}$. Notice that Polylib achieves an equivalent result, but it has to determine solution of the equation for 512 different values of *start*, and then it has to solve a system of 512 unknowns. This is a limitation of Ehrahrt's polynomial approach, rather than a Polylib limitation.

The main idea of our approach is to decouple the estimate of cache misses from the loop

iteration space so that the approach can be fast even for large loop nests. Our approach combines a static symbolic analysis with an efficient and practical implementation. We use SUIF 1.3 and the framework developed by Ghosh et al. for the determination of eligible loop nests, memory references, reuse vectors and for the manipulation of CMEs. We use *Polylib* for the estimation of the total number of iterations (e.g., m^2) and representation of parameterized polyhedra. We developed the software package STAMINA: it sorts the memory references as a function of their reuse vectors (i.e., temporal and spatial reuse, length); it determines their interference densities and it computes the total number of cache misses for each loop nest in an application. STAMINA annotates the original code with directives for the adaptation of the cache-line size for each eligible loop nest.

2.4 Application-Aware Cache Mapping

In this section, we aim to investigate means of adapting data cache locality of memory intensive application to minimize the effects of some of the most common idiosyncrasies of modern caches in embedded systems and DSP processors.

In practice, cache hierarchy can efficiently exploit the inherent data and instruction locality of applications, however, the performance and power consumption of a system –i.e., application and architecture combination– are a function of the design choices in the memory hierarchy (i.e. from memory to register file), and its utilization. This problem is exacerbated in multiprocessor systems and distributed systems because of extremely high demand of data and instructions, and because of communication through relatively slow devices. To avoid CPU stalls due to data and instruction starvation, several approaches have been proposed. In this thesis, we turn our attention to one particular approach that we summarize in the following.

Hybrid adaptive approaches: we consider in this class, on-the-fly hardware and software adaptations. An example of at-run-time hardware adaptation is cache-associativity

adaptation, and an example of the software adaptation is the at-run-time reorganization of registers allocation to reduce register file power dissipation, [85, 86, 87]. Another example of algorithm adaptation is the work by Gatlin et al. [88], where data-copy strategies are applied to exploit cache locality.

In practice, our approach is a hybrid approach and we propose it to solve the problem of cache interference in blocked algorithms. Blocked algorithms, such as MM and FFT, achieve good cache performance on *average*; however, we notice a quite erratic cache behavior on *individual* input sets – due to cache interference. We propose an approach to minimize cache misses due to cache interference changing the cache mapping for some memory references dynamically.

We enforce the problem by an example quantitatively. We implemented an optimal blocked implementation of MM for an architecture with a direct-mapped data cache of size 16KB. We opt for matrices stored in row-major format, which are used commonly. We design the algorithm with no pre-fetching –because pre-fetching hides the latency but does not reduce cache misses. The blocked algorithm can be the result of tiling exploiting cache locality on a uniprocessor system, or the result of a parallelizing compiler for shared-memory multiprocessor systems (or both). We achieve on a uniprocessor system an average 3% data-cache miss rate. The average cache miss is close to the optimal cache performance (roughly 0.5%, [11]). When we observe the cache performance for square matrices of size $2^k \times 2^k$ only, the miss rate soars because of data-cache interference. For example, for square matrix of size 2048×2048, the data cache miss rate is 16%.

In the following, we present two approaches designed to tackle and solve such a data-cache interference problem: spatial scheduling and dynamic cache mapping.

Spatial Scheduling

Cache interference arises when we issue a sequence of memory accesses that share the same cache line in a short interval of time. We may avoid interference by just changing the order of accesses. In fact, there exist situations where just a proper scheduling of memory accesses is sufficient to minimize cache interference. In Section 6.1, we shall show a register allocation that exploits spatial locality at register level. It serializes the loads from the same cache line into registers. When data are in register there are no accesses and therefore there are no more interference. We apply our register allocation on a benchmark from SPEC 2000, `swim`. We show that we achieve predictable and, in general, better performance than using other register allocations. [89].

Dynamic Cache Mapping

Otherwise, if the computation cannot be reorganized to exploit spatial locality at register-file level, we may tailor the cache mapping in such a way to reduce the interference. We propose a hybrid approach to remove data cache miss spikes by changing the cache mapping only when needed. The name of our approach is **Dynamic Mapping**:

1. We produce a blocked algorithm, either by tiling of a loop nest or by a recursive implementation, so that we maximize temporal locality –for one or more cache levels.

2. The blocked algorithm has each elementary block computation (i.e., loop tile) accessing rectangular tiles of data (i.e., tile of matrices).

3. For each memory reference in the elementary block computation, we determine a physical address and an alternative –and unique– address, **twin address**. The physical address is used to map the element in memory; the twin address is used to map the element in cache (the details, how to determine and use twin addresses are explained through an example in Section 6.3.1).

The twin address space does not need to be physically present and, in practice, the twin space is larger than the physical space. A 64bit-register can address $2^{64} \sim 1 * 10^{19}$ bytes of memory, relatively few bytes are physically available.

4. The physical address is used whenever there is a miss in cache to access the second level of cache or memory; we assume the cache is physical tagged, and we can modify the processor and the load queue for our purpose (we shall give more details in Section 6.3.3).

In Section 6.2.1, we show that it exists a **general mapping** that can be driven by a dedicated hardware device. The cache is logically divided in buckets, each bucket is associated with a continuous memory space, which contains elements from a unique vector or matrix. When a memory reference is issued, the reference is sorted towards the unique bucket and therefore a cache location.

In Section 6.3, we show that dynamic mapping is a specific mapping that uses the processor computational power to determine the cache mapping at run time. In fact, such mapping is introduced in the code as affine functions. The affine functions are computed at run time and the results used as alternative addresses. These addresses are used to map the data in cache. It has the same effect as to have the data layout reorganized in memory at runtime [17], using the computational power of the processor, with no data movement [58], no overhead or extra accesses.

Dynamic mapping differs from IMPULSE project [90, 91], which introduces a new memory controller leaving the memory hierarchy untouched. IMPULSE supports a *configurable physical address mapping* and *pre-fetching at memory controller*. Our approach is simpler in the sense that it does not require an operating system layer and any changes to the memory controller. The cache mapping is defined completely by the application, and it can be driven automatically by a compiler.

Dynamic mapping does not need any profile-based approach or dynamic computation changes [88]; it improves portability and lets the developer focus on the solution of the original problem. We differ from Johnson et al. work [92, 93], because we do not use any dedicated hardware to keep track of memory references; the reference pattern is recognized statically.

Dynamic mapping does not change the physical data cache mapping [94, 95, 96] and, potentially, it has no increase in data-cache access latency.

Dynamic mapping is not a bypass technique: we are able to exploit data locality fully - for a level of cache - when algorithms have data locality; the processor does not need bypass a cache entirely. Cache bypassing is an efficient technique designed to increase the bandwidth between processor and memory hierarchy. In general, cache bypassing increases traffic on larger caches, which are slower and more energy demanding (see processors as R5k), and it does not aim to reduce data cache misses. Furthermore, cache bypassing makes the design more complicated and suitable for a general-purpose and high-performance processor.

Dynamic mapping is a 1-1 mapping among spaces; therefore it assures cache mapping consistency for any loads and writes to/from the same memory location. Hardware verification approaches for stale-data in registers - used by processors-compilers that allow speculative loads; for example, IA64 microprocessor [97, 98] - can be safely applied.

CHAPTER 3

JuliusC and Recursion-DAG

The development of blocked algorithms for matrix computations has led the spread use of high-performance scientific libraries. As we introduced previously, blocked algorithms can be implemented using loop nests or recursion. Recursion is extremely appealing to developers because it allows the deployment of top-down techniques naturally. However, recursion is considered non-practical for high-performance routines, mostly because of the overhead of the division process. To make recursion practical, we propose to model the behavior of recursive algorithms in such a way that a compiler can estimate and reduce such overhead. In this chapter, we present JuliusC, a (lite) C compiler. JuliusC unfolds the application call graph partially and it extracts the dynamic behavior. As a final result, it produces a direct acyclic graph (DAG) modeling the function calls, **recursion-DAG**. JuliusC combines static and dynamic analysis and we show that both have negligible time and space complexity.

3.1 JuliusC

JuliusC is a lite compiler that models the division process of a recursive D&C algorithm.

Briefly, we illustrate JuliusC's application on the recursive Fast Fourier transform in Figure 3.1. Consider that, at run time, JuliusC reaches a function call *fft(*,*,***200***,1)* (where * is wild, any valid vector pointers). JuliusC then generates a recursion-DAG rooted at the function call *fft(*,*,***200***,*)*. In Figure 3.2, we present a graphical representation of the

```
void fft(MATRIX_TYPE *re, MATRIX_TYPE *im, int n, int stride) {
  int p,q,i,prime,k;

  p = find_balance_factorization(n);
  q= n/p;

  /* leaf or n is prime */
  if (n<=LEAF || p==n ) DFT_1(re,im,n,stride,cos(M_PI/n),sin(M_PI/n));
  else {
    for (i=0;i<q;i++) {  // by column
      k = i*stride;
      fft(re+k, im+k,p, stride*q);
    }
    distribute_twiddles(re,im,n,p,q,stride);
    for (i=0;i<p;i++) {  // by row
      k = i*stride*q;
      fft(re+k, im+k,q, stride);
    }
  }

}
```

Figure 3.1: Fast Fourier Transform. The factorization is determined at run-time and the recursion stops when n is prime or no larger than $LEAF = 5$.

recursion-DAG. Each node in the DAG represents a function call (of a certain problem size) during the execution of the function call *fft(*,*,***200***,1)*. Embedded in the recursion-DAG, we may recognize a familiar structure: the **plan**, which is used in scientific libraries - e.g., FFTW [12] - to guide the self-installation of recursive algorithms. The only difference is that the recursion-DAG is not a tree (a plan is a binary tree), because of the node *fft*<5> associated with the function call *fft(*,*,5,*)*, which is a child node shared by two nodes-function calls - in the recursion-DAG, (e.g., *fft*<10> and *fft*<20> associated with the function calls *fft(*,*,10,*)* and *fft(*,*,20,*)*, respectively).

In Figure 3.2, we count only 18 different nodes. Each node is identified by the function name and by an integer summarizing the problem size (i.e., fft <5>). The integer number is determined by the factorization of the problem size of the parent node(s) (e.g., the number <5> in *fft*<5> is a factor of <10> and <20>). So building the recursion-DAG, we may store the factorization results and avoid recomputation. (For this example, if, in the worst case scenario, the factorization of an integer n takes $O(\sqrt{n})$ operations, the function call *fft(*,*,20,*)* associated with node *fft*<20> executes $10 * (\sqrt{20} + 5\sqrt{2} + 4\sqrt{5}) \sim 170$ integer

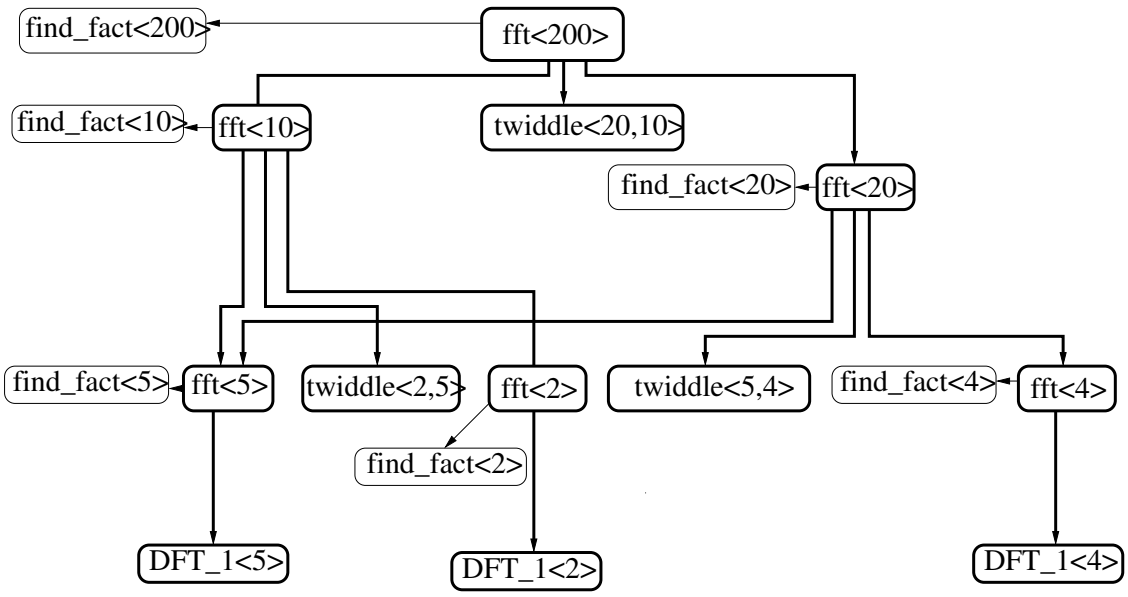

Figure 3.2: Recursion-DAG having *fft(*,*,**200**,*)* as root associated with node *fft*<200>. To simplify the recursion-DAG presentation, **find_fact** = *find_balance_factorization* and **twiddle** = *distribute_twiddle* as in Figure 3.1.

operations for its recursive factorization, and the function call *fft(*,*,10,*)* associated with node *fft*<10> executes $20*(\sqrt{10}+2\sqrt{5}+5) \sim 200$ integer operations, for a total of 370 integer operations. Instead, if we compute the factorization of 200 once and we store the factors in the recursion-DAG, it takes 14 - i.e., $\sqrt{200}$ - integer operations and 5 store instructions - i.e., 5 factors, respectively.)

3.2 JuliusC: an Overview

JuliusC is a (lite) C compiler written in C/C++. To simplify the compiler design, we accept a subset of the C language (e.g., no `struct` and no `union` are handled). For linear-algebra applications, our C-language simplifications have little effect on the design and implementation of D&C algorithms (and were chosen to allow a quick demonstration of our ideas). In the longer run, we will port our techniques into more advanced and robust compilers (e.g.; SUIF [99]) thus avoiding these limitations.

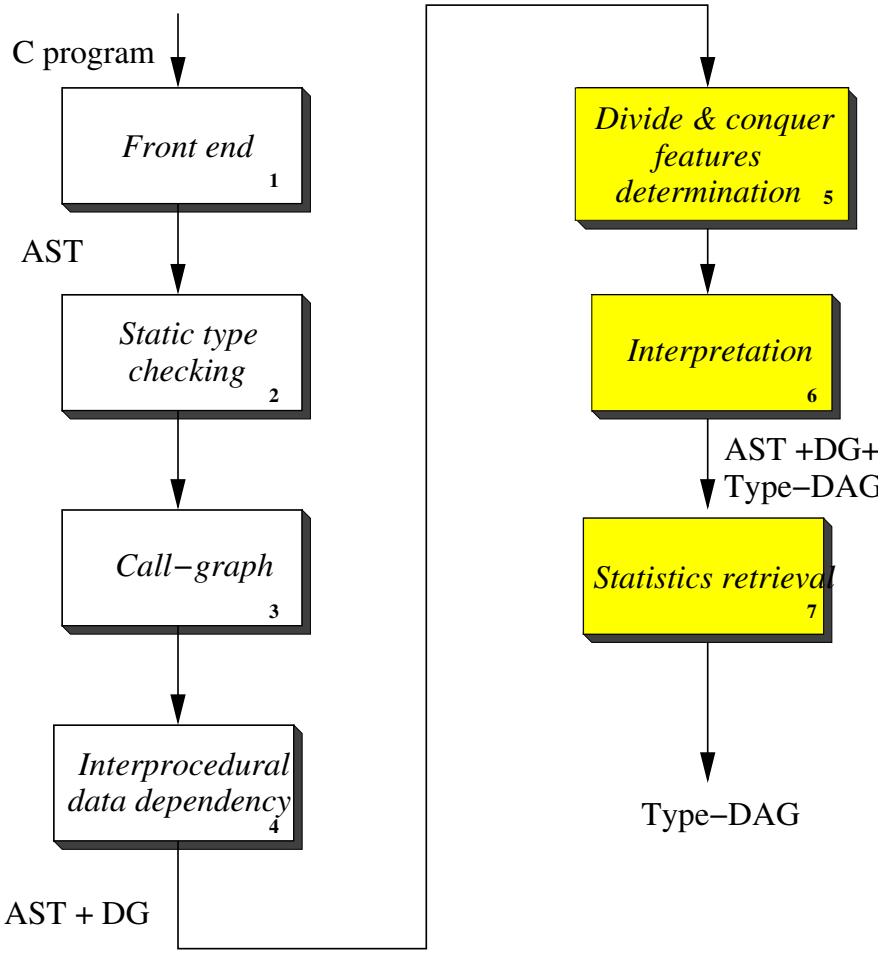

Figure 3.3: JuliusC block Diagram. Acronyms: intermediate representation = IR, abstract syntax tree = AST, data dependency graph = DG.

In different phases of the compilation, we manipulate an intermediate representation (**IR**) of the input program. After lexical and syntactical analysis, we create an abstract syntax tree (**AST**).

We perform static type checking and we annotate the result of the analysis on the AST. Using the AST for each function definition, we determine all function-call expressions. We then determine a call-graph and its all-pair shortest-path matrix closure. Using this analysis, we may mark whether a function definition is explicitly self recursive, recursive, non-recursive or a **leaf** (a leaf is a non-recursive function that has calls to non-recursive functions only).

For example, if the closure matrix shows a loop for a function definition, the function is recursive; otherwise it is not recursive.

We employ a simplified inter-procedural data dependency analysis: the data dependency of each function definition is executed in one pass; arrays are considered as monolithic units (we do not need more accurate analysis because we rediscover the array decomposition from the recursive algorithm - which we assume is correct); for loops, we perform the data dependency in one pass (i.e., we do not determine the loop carried dependencies) of the loop body. We would benefit from a more powerful data dependency analysis, which we plan to obtain by using a more sophisticated infrastructure, but this is not necessary to demonstrate the proposed techniques on useful applications. As a result of the data-dependency analysis, we enrich our IR, building a data dependency graph structure (**DG**) upon the AST.

The last three steps of JuliusC are our original contributions, and we shall discuss them shortly from Section 3.2.1 to Section 3.2.2.

3.2.1 Static Analysis: the Art of Divide et Impera

Our static analysis summarizes the division process of a recursive algorithm by annotating the **formal** parameters of function definitions (composing the application) with three attributes, described as follows:

Divide and Conquer (D&C) formal is a formal that specifies the size of the problem and how the algorithm divides the problem into smaller problems. In practice, a formal is

```
int factorial(int N) {
 if (N<=1)
    return 1;
 else
    return N*factorial(N-1);
}
```

Figure 3.4: Factorial: N is a D&C formal.

annotated as D&C, if it is used as an operand in the conditions of flow-of-control statements,

such as if-then-else and while-do statement, and these statements have an execution path leading to a function call of a recursive function. For example, consider the factorial function in Figure 3.4, the formal `N` is a D&C formal. A D&C attribute is an **inherited** attribute. [1]

Matrix Operand (MO): a formal is annotated as MO, if it is a pointer to a vector. For

```
void update(int *M, int n) {
 int i;
 for (i=0;i<n;i++)
   M[i] = i;
}
```

Figure 3.5: Update: M is a MO formal

example, the formal `M` in Figure 3.5 is annotated as MO. A MO attribute is a **synthesized** attribute.

Location Operand (LO): a formal is annotated as LO, if it is used in the index computation of a vector - identifying a particular element in the vector. For example, the formal `m` in Figure 3.6 is annotated as MO and formal `L` as LO. An LO attribute is a **synthesized** attribute.

```
void set(int *m, int L; int n) {
 int i;
 for (i=0;i<n;i++)
   m[i+L] = i;
}
```

Figure 3.6: Set: L is a LO formal.

The attribute-annotation process is based on two steps. First, the formals of self-recursive function definitions are annotated with D&C attributes, and the formals of leaf function definitions are annotated with MO and LO attributes (as described previously). Second, we determine an in-order left-to-right visit of the call graph (e.g., by a depth-first search) starting from the `main()` function definition. The visit determines a tree: the leaf function definitions

[1] Inherited and synthesized come from the way we annotate these attributes when visiting the call graph, corresponding to inherited and synthesized attributes in syntax-directed translation [100].

correspond to the leaves of the tree. As any syntax-directed translation process (e.g., type checking [100]), we visit the tree and we compute inherited and synthesized attributes using the data dependency among function calls.

Consider the example in Figure 3.7. We annotate the formals of the self-recursive function

```
int M[100];

void main() {
  int size;

  initialize(M);
  get(size);
  if (size>0 && size<100)
    put(f(M,size);
}

int f(int *m, int n) {
  if (n<5) {
    return g(m,n);
  }
  return f(m,n/2)+n;
}
int g(int *p, int q) {
  return p[q+1];
}
```

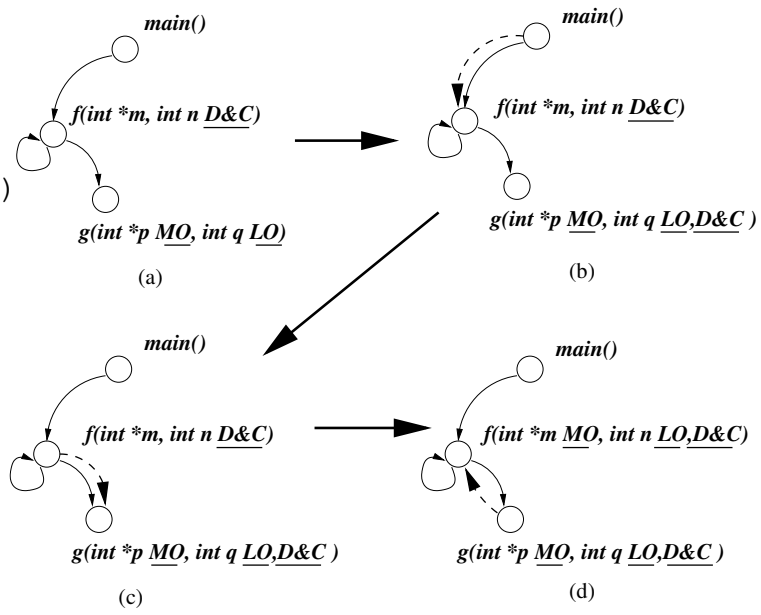

Figure 3.7: Example of multiple attributes: n is eventually a D&C and a LO formal, the evaluation process is schematically represented in Figure (a), (b), (c) and (d).

definition *int f(int *m, int n)*: formal `n` is annotated as D&C. We annotate the formals of leaf function definition *int g(int *p, int q)*: the first formal has attribute MO and the second formal has attribute LO (Figure 3.7 (a)).

We start the second phase of the annotating process from `main()`, which does not have formals. We then visit `f()`: the second formal `q` of `g()` inherits an attribute D&C (Figure 3.7 (b)) from the formal `n` of function definition `f()`. We then visit `g()` (Figure 3.7 (c)). We then backtrack to `f()` and we synthesize the attributes from its child `g()`: the first formal has attribute MO, and the second has attribute LO (Figure 3.7 (d)). Eventually, the formals `n` in `f()` and `p` in `g()` have attributes D&C and LO. For example, `n` in `f()` is an operand

in the condition of the only if-then-else statement in `f()` and, indirectly, it is used to access the vector `p` in function `g()`.

An attribute summarizes the use of a formal in the body of the function definition and, in turn, in the functions called as well.

3.2.2 Dynamic Analysis: Interpretation and Recursion-DAG

During the dynamic analysis, each function call will be associated with a node in a **recursion-DAG** as follows.

When we reach an function call (e.g., `ftt(Re,Im,n,stride)` as in our example in Section 3.1), we evaluate the actuals (e.g., `fft(Re,Im,200,1)`). We look up for the definition of the function call (e.g.,*void fft(MATRIX_TYPE *re, MATRIX_TYPE *im, int n /*D&C*/, int stride)* { }). Because only D&C formals are used by the recursive algorithm for the division process, we consider the function name and the actual values associated with D&C formals as the **key** of a function call (e.g., `fft(*,*,200,*)`). Using this key, we look up whether or not a node in the recursion-DAG already exists (with the same key). If it does not exist, a new node is added.

The recursion-DAG is an unfolding of the call graph for an initial input, where only the division work is recorded using function caching.

Static Profiler

The recursion-DAG can be used as a means for a fast and precise collection of statistics about the execution of the recursive algorithm. For example, to compute how many times a function (solving a particular problem size) is called, we just need to exploit the acyclic nature of the recursion-DAG. In fact, we determine a topological sort of the recursion-DAG by a depth-first search. The first node is associated with `main()` and we set its **count** = 1. For every node in the topological sort v, we consider each child u and we update the child count $u.count+ = v.count$. In this way, while we compute the number of function calls per

node by visiting each node once, we can compute the total number of function calls and the total number of nodes in the recursion-DAG.

The **reuse ratio**, $\mathcal{R}$, is the ratio between the number of functions calls over the number of nodes in the recursion-DAG. It represents a concise estimate of the work reduction. For example, we may precompute the common computations and store them in the recursion-DAG, the reuse ratio represents how many time we reuse those precomputed values during the execution of the recursive algorithm.

$$\mathcal{R} = \frac{\#\text{function calls}}{\#\text{nodes in recursion-DAG}}. \tag{3.1}$$

3.3 Experimental Results

In this section, we show that our approach is practical; that is, the execution time of our static and dynamic analysis is negligible. We show that we may exploit reuse of the division work and we present the reuse ratio for 6 recursive algorithms for various inputs.

Our compiler does not yet fully implement the more traditional optimizations (e.g., register allocation) and thus we cannot present here automated results of actual speedups. However, our contribution is to present the automatic derivation of the recursion-DAG. The practicality of the overall approach has been already demonstrated previously [22, 101, 27] via *ad-hoc* implementations of matrix multiply and FFT, that use the same ideas as we advocate here. Specifically, these recursive implementations used DAG structures that were hand derived and with optimized leaf codes; they produced actual performance - on a range of architectures - comparable to that of the best blocked-tuned approaches.

We apply JuliusC to 6 recursive algorithms (for various inputs); three algorithms are from linear algebra; two algorithms are from number theory and one is a classic graph algorithm. The results presented in Table 3.1 are a significant excerpt and they can be reproduced running JuliusC on-line (see author website `www.ics.uci.edu/~paolo`). Table

3.1 reports the reuse ratio (Equation 3.1) for recursive algorithms, the ratio represents the work reduction we can achieve.

Table 3.1: Reuse ratio $\mathcal{R}$ is a function of the problem size, the algorithm definition and the static analysis.

Notation	Name	$\mathcal{R}$	Inputs
$\binom{n}{k}$	**Binomial**	14 3053	$\binom{10}{5}$ $\binom{20}{10}$
$n = pq$	**Integer factorization**	93 133	$n = 65536$ $n = 524288$
FFT_n	**Balanced Cooley-Tookey**	34 2076 12683	$n = 128$ $n = 5000$ $n = 65536$
$\mathbf{A}^*$	**All-pair shortest path**	60728 23967500	$\mathbf{A} \in \mathbb{Z}^{750\times750}$, $\mathbf{A} \in \mathbb{Z}^{7500\times7500}$
$\mathbf{A} = \mathbf{LU}$	**LU-factorization**	16110 93757	$\mathbf{A} \in \mathbb{R}^{300\times300}$, $\mathbf{A} \in \mathbb{R}^{500\times500}$
C+=AB	**Matrix multiply**	1950 107176 2765800	$\mathbf{C}, \mathbf{A}, \mathbf{B} \in \mathbb{R}^{100\times100}$ $\mathbf{C}, \mathbf{A}, \mathbf{B} \in \mathbb{R}^{517\times517}$ $\mathbf{C}, \mathbf{A}, \mathbf{B} \in \mathbb{R}^{1123\times1123}$

We consider briefly each illustrative example as follows.

Binomial is a straightforward recursive algorithm but its time complexity is exponential. Function caching - in JuliusC - allows the reuse of already computed values such that the final time complexity is just polynomial. For the input $\binom{10}{5}$, the reuse is 14; however, larger reuse ratios are achievable for larger problems.

Integer factorization is used in algorithms such as Cooley-Tookey FFT. Given an integer n, we determine the factors p and q so that $n = pq$ and $\min_{p,q} |p - q|$. We determine the factorization for p and q recursively. We analyze an exact and a heuristic factorization (e.g., integer factorization is important in cryptography [102]).

Balanced Cooley-Tookey is the Cooley-Tookey FFT algorithm using balanced factorizations (e.g., Cormen et al. [63] and D'Alberto et al. 2003 [27]). In Section 4.3, we introduce the detail of our implementation.

All-pair shortest path is an algorithm based on matrix multiplication and Floyd-

Wharshall algorithm (Floyd 19962, Ullman et al. 1990 [66, 64]). In Section 4.4, we present a detailed investigation.

LU-factorization is an algorithm based on matrix multiply with no row pivoting [101, 5]. In Section 4.1, we present the LU-factorization with partial pivoting.

Matrix multiply is matrix multiplication with matrices stored in non-standard layout [22, 6]. In Section 4.2, we present our original work.

For completeness, in Figure 3.8 we show results for a representative example, FFT (the main algorithm is presented in Figure 3.1 Section 3.1). In Figure 3.8, we present an excerpt from JuliusC's output, which is reproducible on-line on the native system - Fujitsu HAL 100MHz. The data dependency analysis takes 0.05 seconds and the annotation of the formals

```
Call Graph from Main ....
----------> get time 0 sec<------
Function calls properties ....
----------> get time 0 sec<------
Data Dependency Analysis ...
----------> get time 0.05 sec<------
Data dependecy result ....
----------> get time 0 sec<------
Marking D&C formals ...
----------> get time 0.01 sec<------
Check the formals on the function definitions ...
Interpretation ...
----------> get time 5.4 sec<------
On count
Reuse Ratio 1007.26
----------> get time 0 sec<------
main<> [1] {} |Rec|
   fft<3780> [1] {0} |Rec|
      find_balance_factorization<3780> [1] {60} |LeaF|
      fft<60> [63] {0} |Rec|
         find_balance_factorization<60> [63] {6} |LeaF|
         fft<6> [630] {0} |Rec|
            find_balance_factorization<6> [630] {3} |LeaF|
            fft<3> [3780] {0} |Rec|
               find_balance_factorization<3> [3780] {3} |LeaF|
               DFT_1<1260,3> [3780] {0} |LeaF|
```

Figure 3.8: JuliusC's output excerpt for Balanced Cooley-Tookey

using attributes is negligible (static analysis). The interpretation takes 5.4 seconds (dynamic analysis). The collection of the statistics is negligible. The result of the analysis is a text-

based recursion-DAG. Each function call has an entry that we can describe using a single line:

```
name< size > [ number of times this function is called ] |Rec| or |LeaF|
```

The problem `fft()` has size `<6>`; it is called 630 times; the last attribute is `|Rec|`, which stands for *Recursive*. The problem `DFT_1()` has problem size `<1260,3>` (problem size and *stride*); it is called 3780 times; the last attribute is `|LeaF|`, non recursive.

We use a somewhat limited graphical representation of the final recursion-DAG. We use indentation to present the relation among function calls so we can identify the root immediately: the node `main`. Using this simplified format, two siblings function calls have the same indentation and two function calls with a caller-callee relation have different indentation; for example, *fft(*,*,3,*)* is called by *fft(*,*,6,*)*:

Conclusions

Our original contribution is a concise representation of recursive algorithm unfolding by using an intuitive data structure, the recursion-DAG. In this chapter, we have presented an automatic approach to determine such a data structure. We showed that the approach is practical and it can be incorporated into a generic compiler. We apply our approach to 6 recursive algorithms and we present an estimation of the potential improvements.

CHAPTER 4

Recursive D&C Examples from Linear Algebra

Though, matrix algebra was formulated as theory in the eighteenth century by mathematicians such as Gauss, however, the topic of matrix computations (and algorithms) has attracted the interest of applied mathematicians for about two thousand years. In fact, Chinese mathematicians discovered and used algorithms for the solution of systems of equations in the third century; unfortunately, most of the Chinese early contributions were destroyed by emperor dictation.

In this chapter, we present four applications following a top-down approach. In Section 4.1, we shall present LU-factorization with partial pivoting. We show that though the recursive algorithm is elegant and natural, however the division process hides an implementation problem that affects performance and correctness of the computation. In Section 4.1, we explain the problem and we present our solution and implementation. In Section 4.2, we introduce our approach for matrix multiply (MM), **fractal matrix multiply**. We investigate the effects of matrix-layout optimizations as well as data allocation to registers on performance. In fact, we show that a recursive matrix layouts exploit cache locality, however aggressive data allocation to registers have easier implementation for matrices stored in row-major format. In Section 4.3, we shall conclude with our implementation of FFT. Finally in Section 4.4, we introduce algorithms for the all-pairs shortest path (APSP) problem (and

Transitive Closure). We show that the same optimizing techniques proposed for MM can be applied for APSP achieving twofold speedups with respect to previous algorithms.

4.1 LU-factorization

This section is organized as follows. In Section 4.1.1, we recall Toledo's algorithm. In Section 4.1.2, we show that Theorem 2.1 in [5] can be applied and therefore we estimate data cache misses for LU-factorization. In Section 4.1.3, we present the details of our implementation. In Section 4.1.4, we present experimental results for seven different memory hierarchies (with fixed code) and performance evaluation on a subset of four machines.

4.1.1 LU-factorization with Partial Pivoting

LU-factorization with partial pivoting for a square matrix $\mathbf{A}$ $m \times m$ is an algorithm able to find three matrices $\mathbf{P}$, $\mathbf{L}$ and $\mathbf{U}$ such that $\mathbf{PA} = \mathbf{LU}$. The matrix $\mathbf{L}$ is a $m \times m$ **unitary lower triangular matrix** (i.e., the diagonal elements are all one, all the elements above the diagonal are zero), $\mathbf{U}$ is an $m \times m$ **upper triangular matrix** and $\mathbf{P}$ is a **row permutation matrix** (or plane rotation matrix). .

Algorithm (Toledo's algorithm) [5] Matrix A can be logically composed of four sub-blocks:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_1 \\ \mathbf{A}_2 & \mathbf{A}_3 \end{bmatrix} \tag{4.1}$$

where $\mathbf{A}_0$ is a $\lceil m/2 \rceil \times \lceil m/2 \rceil$ square matrix. For any $M > 0$:

1. Leaf Computation: if $m < M$, determine directly the row permutation matrix $\mathbf{P}$, the lower triangular matrices $\mathbf{L}$, and the upper triangular matrix $\mathbf{U}$ so that $\mathbf{PA} = \mathbf{LU}$.

2. Otherwise, recursively determine the row permutation matrix $\mathbf{P}_1$, the lower triangular

matrices $\mathbf{L}_0$, the matrix $\mathbf{L}_2$, and the upper triangular matrix $\mathbf{U}_0$ so that

$$\mathbf{P}_1 \begin{bmatrix} \mathbf{A}_0 \\ \mathbf{A}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{L}_0 \\ \mathbf{L}_2 \end{bmatrix} \mathbf{U}_0 = \begin{bmatrix} \mathbf{L}_0\mathbf{U}_0 \\ \mathbf{L}_2\mathbf{U}_0 \end{bmatrix} \tag{4.2}$$

3. Apply permutation $\mathbf{P}_1$ on the other half of the matrix:

$$\begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_3 \end{bmatrix} = \mathbf{P}_1 \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_3 \end{bmatrix} \tag{4.3}$$

4. Determine $\mathbf{U}_1$ in the lower triangular system $\mathbf{A}_1 = \mathbf{L}_0\mathbf{U}_1$.

5. Compute $\mathbf{A}_3 - = \mathbf{L}_2\mathbf{U}_1$.

6. Recursively determine the permutation matrix $\mathbf{P}_2$, the lower triangular matrix $\mathbf{L}_3$ and the upper triangular matrix $\mathbf{U}_3$ so that $\mathbf{P}_2\mathbf{A}_3 = \mathbf{L}_3\mathbf{U}_3$.

7. Permute $\mathbf{L}_2$ by $\mathbf{P}_2$, that is, $\mathbf{L}_2 = \mathbf{P}_2\mathbf{L}_2$

8. The final permutation matrix is the combination $\mathbf{P} = \mathbf{P}_1\mathbf{P}_2$.

♠

4.1.2 Access Complexity of LU Factorization

In this section, we demonstrate the optimal data locality of LU-factorization by following the same approach proposed by Toledo in his original work [5]. In practice, we show that

$$Q_{LU}(s, m) \leq \zeta[2m^2(\frac{m}{2\sqrt{s/3}} + \log_2 m) + 2m^2(1 + \log_2 m)] \tag{4.4}$$

Where $Q_{LU}(s, m)$ is the memory-access number of the LU-factorization applied to a matrix of size m^2 and the data cache has size s, and where $\zeta \geq 0$ is introduced to model interference in the data cache.

To show that the factorization has the access complexity represented in Equation 4.4, we must address the access complexity for the components of the algorithm such as the

lower triangular system (LTS) solver and MM. In fact in this section, we show the access complexity of such algorithms and then we apply Toledo's proof – not reported here.

First we consider the MM algorithm. MM has access complexity $Q_{MM}(s,m) = \Theta(\gamma\sqrt{\frac{3}{s}}\frac{3m^3}{2})$ for $m \geq \sqrt{s/3}$ (details of the analysis can be found in Section 4.2, otherwise see Hong and Kung 1981 [11] or Frigo et al. [21]). The coefficient γ is introduced to consider the fact that caches are not ideal. The MM satisfies Equation 2.2 [5] - when $\gamma = 1$.

Second, we consider the LTS algorithm. LTS is represented by an equation $\mathbf{LX} = \mathbf{B}$, where $\mathbf{L}$ is a lower unitary triangular matrix of constants (e.g., $\mathbf{L} = \begin{bmatrix} \mathbf{L}_0 & \mathbf{0} \\ \mathbf{L}_2 & \mathbf{L}_3 \end{bmatrix}$ with $\mathbf{L}_0$ and $\mathbf{L}_3$ lower unitary triangular matrices), $\mathbf{B}$ is a matrix of constants and $\mathbf{X}$ is a matrix of free variables, $\mathbf{X}$ is computed in place of $\mathbf{B}$. In Figure 4.1, we present the LTS pseudo-code.

```
LTS(L,X,B) {
   if (|L|+|B| < s) solve AX=B and store X in B.
   else {
     LTS(L0,X0,B0)
     LTS(L0,X1,B1)
     B3 -= L2X1;
     B2 -= L2X0;
     LTS(L3,X2,B2)
     LTS(L3,X3,B3)
   }
}
```

Figure 4.1: Lower triangular system solver

Theorem 1 *The access complexity of LTS for square matrices of size m is $Q_{LTS}(s,m) \leq \frac{m^3}{\sqrt{s/3}} + m^2$ for $m > \sqrt{s/2}$.*

Proof: When m is an integer power of two, the recurrence equations for the access complexity is $Q_{LTS}(s,m) \leq 4^i Q_{LTS}(s, \frac{m}{2^i}) + \sum_{j=0}^{i-1} 4^j 2 Q_{MM}(s, \frac{m}{2^{j+1}})$. We stop when $i = 1/2\log_2(2m^2/s)$ achieving $Q_{LTS}(s,m) \leq \frac{2m^2}{s} Q_{LTS}(s, \sqrt{s/2}) + \gamma\frac{3\sqrt{3}m^3}{2\sqrt{s}}[1 - \sqrt{\frac{s}{2m^2}}]$. At the leaves, there is locality exploited among LTSs and MMs so that $Q_{LTS}(s, \sqrt{s/2}) \leq 3s/4$. For $n > \sqrt{s/2}$, the solution to the equation is $3m^2/4 + \gamma\frac{3m^3}{4\sqrt{s/3}}(1 - \sqrt{\frac{s}{2m^2}})$. □

Thus, we have Equation 4.4 because the LTS algorithm satisfies Equation 2.1 and the MM satisfies Equation 2.2 in Toledo's original proof [5], and thus the result.

To conclude this section, we define here and we use in the experimental results section (i.e., Section 4.1.4) the **number of cache misses per floating point operation** as $\mu(m)$. In practice, $\mu_{LU}(m) = Q_{LU}(s,m)/(\frac{2}{3}m^3)$, which is asymptotically a function of the cache size; that is, $\mu(m) = O(1/\sqrt{s})$.

4.1.3 Factorization Implementation

We present the pseudo-code of our algorithm in Figure 4.2.

```
/****************
 *       | A0 A1 |
 *   A = | A2 A3 |
 *
 * */

int  LU(A,min_col,max_col) {
  width = max_col - min_col;
  if  (width<=ENDFRACTALLAYOUT ) {          /* Stop Recursion */
    flag = MAX(a,min_col,max_col);          /* Find Pivot */
    if (flag) PERMUTATION_ROWS();           /* Permute */
    for (i=0; i<width;i++) {                /* classic LU*/
      if (PIVOT==Zero) return 0;            /* singular */
      flag= GELM(A,min_col+i);              /* Gauss elimination */
      if (flag) PERMUTATION_ROWS();         /* Pivoting */
    }
  }
  else {                                    /* Recursion */
    if (!LU(A,min_col,max_col-(width/2)))   /* First Half */
      return 0;
    PERMUTE(A1,A3);                         /* Permute */
    LTS_and_GEMM(A,min_col,max_col);        /* LTS + MM */
    if (max_col==ColumnOf(A)) {             /* Second Half */
      return LU(A+width/2,0,max_col/2);     /* Rectangular */
    }
    else
      return LU(A3,max_col-width/2,max_col);/* Near Square */
  }
}
```

Figure 4.2: LU factorization

For explanation purpose, we hide some of the implementation details and we turn our

attention to the main ideas of the algorithm. In practice, the algorithm is composed of two parts: the recursive computation and the leaf computation.

During the recursive computation, the matrix $\mathbf{A}$ is logically divided into two rectangular matrices: the first half is $\begin{bmatrix}\mathbf{A}_0\\ \mathbf{A}_2\end{bmatrix}$ and the second half is $\begin{bmatrix}\mathbf{A}_1\\ \mathbf{A}_3\end{bmatrix}$. The recursive algorithm determines the factorization of the first half; then it performs a *conquer work* such as row permutations, LTS and MM; at last, it factorizes the second half of the matrix $\mathbf{A}$ and it distinguishes two cases: whether the second half of $\mathbf{A}$ is a rectangular matrix or a (near) square matrix (i.e., $\mathbf{A}_3$).

The leaf computation is composed of three basic steps. First, we search for the maximum element in absolute value, the **pivot**. Second, we permute the current row with the **pivot row**; that is, where we have found the pivot. Third, we perform the Gaussian elimination and scaling and we store the result in place. All these steps are a **column computation** because the computation accesses a set of sub matrices of $\mathbf{A}$ stacked as a column.

A column computation can be described as a set of MM of the type $\mathbf{A_x} = \mathbf{L_x} * \mathbf{U}_{x_0}$ where $\mathbf{x}$ is an ordered set of indices $\{x_0, x_1, \ldots, x_k\}$. The indices determine uniquely what sub-matrices the computation accesses and they can be are determined in two steps. First, the column computation determines $\mathbf{A}_{x_0} = \mathbf{L}_{x_0} * \mathbf{U}_{x_0}$. In fact, the computation performs a binary search of the column of the matrix $\mathbf{A}$ following this rule: the column x_0 must be in either $\mathbf{A}_0 = \mathbf{L}_0\mathbf{U}_0$ or $\mathbf{A}_3 = \mathbf{L}_3\mathbf{U}_3$. The result of this binary search is stored in a **trace**. Second, when a leaf computation is found and the result is stored, the search backtracks the binary search tree using the trace and it determines the next index, thus,the next leaf computation.

The computation, $\mathbf{A}_x = \mathbf{L}_x * \mathbf{U}_y$, is associated with a node at any level in a recursion-DAG. Type and size of a node specify computation and data lay out of the operands. When we perform Gaussian elimination, the matrices $\mathbf{L}_x$ and $\mathbf{U}_y$ are stored as results in $\mathbf{A}_x$.

In practice, especially if we use a recursive data layout for matrices that are not power-

of-two size, we must address the possibility that the operands can be stored in different formats. Thus, the computation should be aware of the operands layout and access the data correctly.

In fact, we use the following recursive layout of an $m \times n$ matrix $\mathbf{A}$ into a one-dimensional array $\mathbf{v}$ of size $m * n$ that we identify as **fractal layout**. If $m < M$ and $n < M$ where M is a specific threshold, then $v[i * n + j] = a_{i,j}$. Otherwise, $\mathbf{v}$ is the ordered concatenation of the layouts of the submatrices $\mathbf{A}_0, \mathbf{A}_1, \mathbf{A}_2$, and $\mathbf{A}_3$ of the following **balanced** composition.

- $\mathbf{A}_0 = \{a_{i,j} : 0 \leq i < \lceil m/2 \rceil, 0 \leq j < \lceil n/2 \rceil\}$,
- $\mathbf{A}_1 = \{a_{i,j} : 0 \leq i < \lceil m/2 \rceil, \lceil n/2 \rceil \leq j < n\}$,
- $\mathbf{A}_2 = \{a_{i,j} : \lceil m/2 \rceil \leq I < m, 0 \leq j < \lceil n/2 \rceil\}$ and
- $\mathbf{A}_3 = \{a_{i,j} : \lceil m/2 \rceil \leq i < m, \lceil n/2 \rceil \leq j < n\}$.

A $m \times n$ matrix is **near square** when $|n - m| \leq 1$. Notice that if $\mathbf{A}$ is a near-square matrix, so are the submatrices $\mathbf{A}_0$, $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$ of its balanced composition. Indeed, a straightforward case analysis ($m = n-1, n, n+1$ and m even or odd) shows that, if $|n-m| \leq 1$ and $S = \{\lfloor m/2 \rfloor, \lceil m/2 \rceil, \lfloor n/2 \rfloor, \lceil n/2 \rceil\}$, then $\max(S) - \min(S) \leq 1$. (This layout layout just defined can be viewed as a generalization of the Z-Morton layout for square matrices, [14, 6] or as a special case of the Quad-Tree layout [16]).

The factorization computation may stop at a level where the operands are stored in different formats, as a function of the threshold MM and the we can enumerate the following four cases: $\mathbf{A}_x = \mathbf{L}_x = \mathbf{U}_x$, all matrices coincide, and they are in either Z-Morton format or in row-major format; $\mathbf{A}_x = \mathbf{L}_x$ is in Z-Morton format and $\mathbf{U}_y$ is in row-major format; and, finally, $\mathbf{A}_x = \mathbf{L}_x$ is in row-major format and $\mathbf{U}_y$ is in Z-Morton format. The leaf computations must use carefully the information available so that they can perform the computation correctly.

Gaussian Elimination Routine

When the pivot is determined and the permutation takes place, the row determined as leading row for Gaussian elimination is defined as **normalizing row**. In practice, Gaussian elimination subtracts the normalizing row multiplied by a proper scalar value to each following row. Gaussian elimination and scaling is logically composed of two steps; in Figure 4.3, we can see a possible access pattern on different stages in the computation. First, $\mathbf{A}_x = \mathbf{L}_x * \mathbf{U}_x$, the

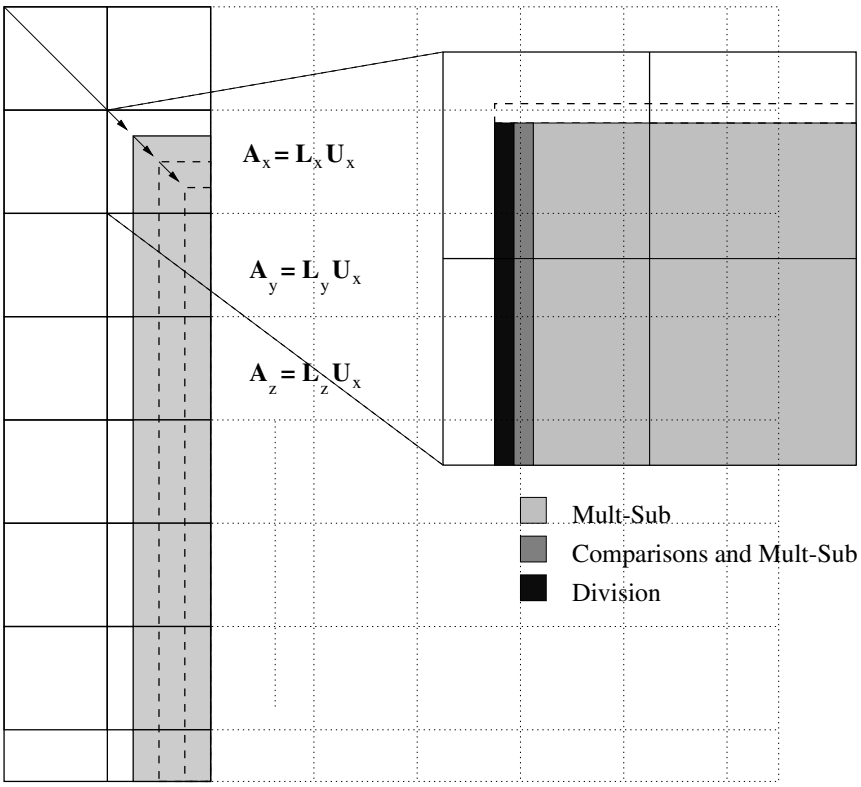

Figure 4.3: Visualization of the access pattern when consecutive Gaussian eliminations are computed.

normalizing row is locally stored and the computation always accesses a square sub matrix of $\mathbf{A}_x$. Second, $\mathbf{A}_y = \mathbf{L}_y * \mathbf{U}_x$, the normalizing row is not locally stored (i.e., it is stored into another submatrix not currently under computation) and the computation always accesses a sequence of sub-matrices of $\mathbf{A}$ composing a column of the original matrix.

Notice that, we compute only a sub matrix at each step: on the right side of Figure 4.3,

we can see, through a zoom effect, the operations performed in the first step. Notice that we determine the next pivot element (comparison operation) while we compute the Gaussian elimination.

In our implementation, for the computation of $\mathbf{A}_x = \mathbf{L}_x * \mathbf{U}_x$, there are three possible cases and we present them in Figure 4.4, because of the sub-matrix may be composed of four blocks in row-major format.

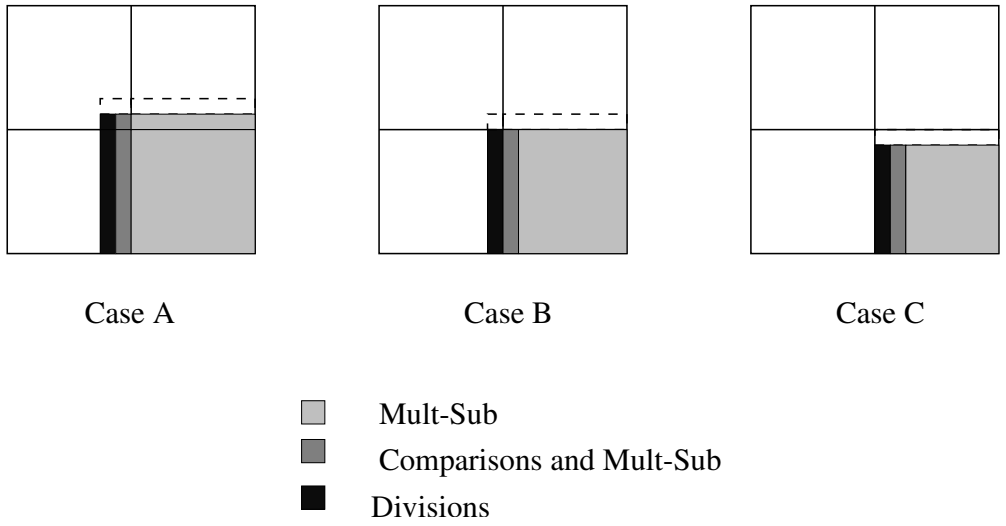

Figure 4.4: Case A: The first quadrant has all types of computations. Case B: first quadrant has only a pivot element. Case C: the computation is done only in the fourth quadrant

Case A: we perform the normalization of the rows in every quadrant.

Case B: we perform the normalization only in the fourth quadrant and divisions only in the third.

Case C: the computation involves only the fourth quadrant.

We present the computation involving the other sub-matrices in Figure 4.5. The computation is described by the equation $\mathbf{A}_y = \mathbf{L}_y * \mathbf{A}_x$ and, notice that the sub-matrices are composed of four blocks in row-major format.

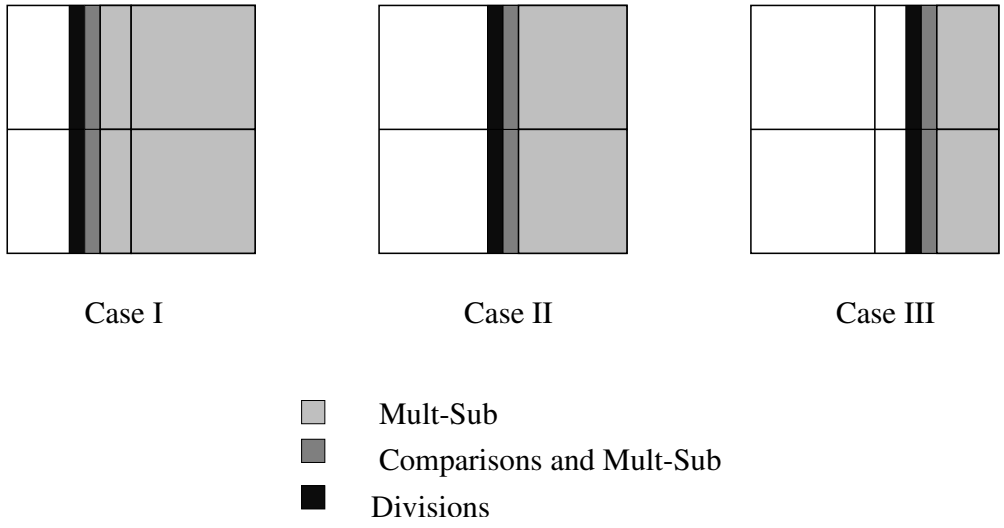

Figure 4.5: Case I: in all four quadrants there are multiplication-subtractions. Case II: divisions are performed in the first and third quadrant and Multiplication-subtractions in the others. Case III: the computation is limited in the second and fourth quadrants.

Once the normalizing row is determined, every row computation is independent and involves a division between scalar values in single/double precision. In modern processor, though a division floating point unit is not pipelined and it has a long latency, however it does not stall the microprocessor execution. For example, SparcUltraIIi has a non-blocking division floating point unit that takes 22 cycles for operands in double precision format. In practice, we can hide such a long computation latency by exploiting the parallelism among rows. Though, the division at the first row cannot be hidden, however we can issue the division of the second row as soon as the first division is committed and while the operations in the first row are not committed yet.

Permutation

In the original definition of LU-factorization, partial pivoting involves the permutation of full rows of **A**. Instead, Toledo's Algorithm proposes the application of a lazy row permutations. A row is logically divided in segments and the permutation takes place among segments at different times during the computation. The application of a Z-Morton layout explicitly

breaks a matrix row among different segments across different submatrices.

Due to the importance of row permutation in the matrix factorization, we introduce a specific data structure used to ease the implementation. In fact, we use a *sub structure* of the recursion-DAG, the **data-layout DAG** (DLDAG), which is a Quad Tree. A C-like description of the structure follows:

```
typedef struct node_structure_layout DLDAG;
struct node_structure_layout {
  DLDAG *sons[4];      /* links to the sub-blocks A0, A1, A2 and A3   */
  int m,n;             /* size of A */
  int di[3];           /* offset of A1, A2, A3 with respect to A   */
};
```

In practice, row permutation using the DLDAG involves similar algorithms to the ones implemented for the column computations described previously. Notice that the row permutation using DLDAG for matrices stored in Z-Morton layout involves extra computations to access and to move data; however, its complexity it is still linear to the size of the rows to permute.

4.1.4 Experimental Result

This section is organized as follows. In Section 4.1.4, we present the cache performance of our algorithm with respect to the codes available in the SunPerformance library. In Section 4.1.4, we present the actual performance of our implementation.

Cache Miss

The results of this section are based on simulations. We simulate the cache miss rates (on an SPARC Ultra 5) using the *Shade* software package for Solaris, of Sun Microsystems. The applications are compiled for the SPARC ultra2 processor architecture (V8+) and then simulated for various cache configurations. We choose specific configurations to correspond to those of a number of commercial systems. Thus, when we refer to the R5000 IP32, we are really simulating an ultra2 CPU with the memory hierarchy of the R5000 IP32.

Table 4.1: Summary of simulated configurations

Configurations	**Cache Size**	**Line Size**	**Ways/ Write Policy**	$\zeta(1300)$
SPARC 1				
U1	64KB	16B	1 / through	0.249
SPARC 5				
I1	16KB	16B	1	
D1	8KB	16B	1 / through	0.311
Ultra 5				
I1	16KB	32B	2	
D1	16KB	32B	1 / through	0.285
U2	2MB	64B	1 / back	0.020
R5000 IP32				
I1	32KB	32B	2 / back	
D1	32KB	32B	2 / back	0.042
U2	512KB	32B	1 / back	0.085
Pentium II				
I1	16KB	32B	1	
D1	16KB	32B	1 / through	0.181
U2	512KB	32B	1 / back	0.063
HAL Station				
I1	128KB	128B	4 / back	
D1	128KB	128B	4 / back	0.020
ALPHA 21164				
I1	8KB	32B	1	
D1	8KB	32B	1 / through	0.212
U2	96KB	32B	3 / back	0.077

We also simulate the code for Sun Performance Library, which is available as standard library for WorkShop compiler. This offers a reference, and generally fractal has fewer misses. However, it would be unfair to regard this as a competitive comparison.

In Table 4.1, we summarize the 7 memory hierarchies. We use **Shade** notation: I= Instruction cache, D=Data cache, U=Unified cache and L=any cache at a level.

In the last column in Table 4.1, we compute the value of ζ when matrix has size 1300. In fact, $\zeta(1300) = \frac{\mu_{measured}(1300)}{\mu(1300)}$ where $\mu(m)$ is the number of misses per FLOP. The coefficient represents briefly the discrepancy between the upper bound and the experimental results, due to the cache line effect, the cache associativity of each cache level, the hierarchy and the over estimation of the accesses complexity.

We can notice three interesting features: the cache line size $\ell \in \{2, 4, 8, 16\}$ for direct-

mapped cache has an (inverse) proportional effect; cache line ℓ and cache associativity α have more than multiplicative effect ($\zeta/(\ell\alpha)$); and for the second level of cache the overall effect is super-linear, only exception for R5000, ($\zeta/(\alpha_1\alpha_2\ell_1\ell_2)$).

We present separately from Fig. 4.6 to 4.12 (from page 75), we present $\mu(m)$ and also the number of code misses.

Performance: MFLOPS

We compare the performance of our code with two other implementations and with peak performance (when no best algorithm is known). When available, we compare the performance of native implementations by vendor libraries or of LU-decomposition based on ATLAS MM. Four architectures have been tested and we present a summary of their characteristics and performance in Table 4.2. We present our experimental results from Figure 4.13 to 4.16 (from page 75).

Table 4.2: Processor Configurations

Processor	Ultra 2i	PentiumII	R5000	SPARC64
Registers	32	8	32	32
MUL/ADD - latency cycles	separate - 3	separate - 8	merged - 2	merged-4
DIV-latency	22	N/A	N/A	8
Peak (MFLOPS)	666	400	360	200
Peak Fractal-size	352 - 3000	138 - 2800	112 - 2500	158 -2048
Peak ATLAS-size	260 - N/A	210 - N/A	N/A	N/A

The running time of the codes can be split in two contributions: $T(m) = T_M(m,t) + T_G(m,t)$. Consider a square matrix A of size $m \times m$ such as A is recursively composed of submatrices (i.e., tiles) smaller than $t \times t$ (also called blocking factor). Then, T_M is the running time of MM and T_G is the running time of Gaussian elimination. If we decide to have a large t, we can improve the performance of MM because we can exploit better data reuse in registers (t cannot be larger than a threshold value where performance collapses due to poor utilization of L1). However, if we use a too large t, the algorithm will call fewer

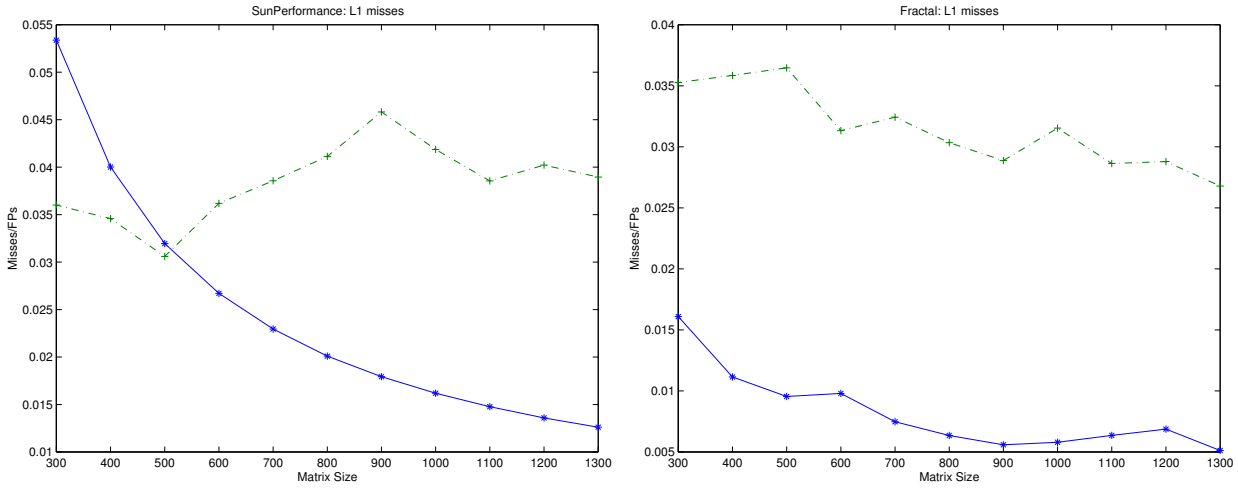

Figure 4.6: SPARC 1: * code misses, + data misses

times MM and the execution time of the Gaussian elimination, T_G, will take longer. An optimal parameter t for matrix multiply may be non optimal for LU-factorization. As proof of the previous observation, but not reported here, we increased the tile size from 32×32 to 48×48, which is optimal for MM. The performance got worse. By profiling, we could measure that the improvements of T_M did not overcome the loss of performance of T_G.

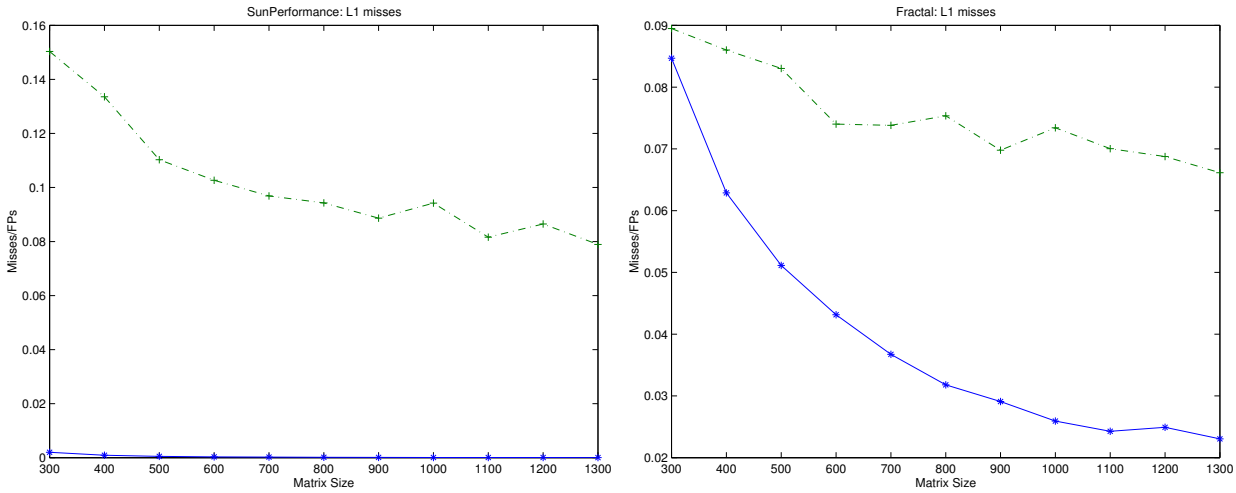

Figure 4.7: SPARC 5: * code misses, + data misses

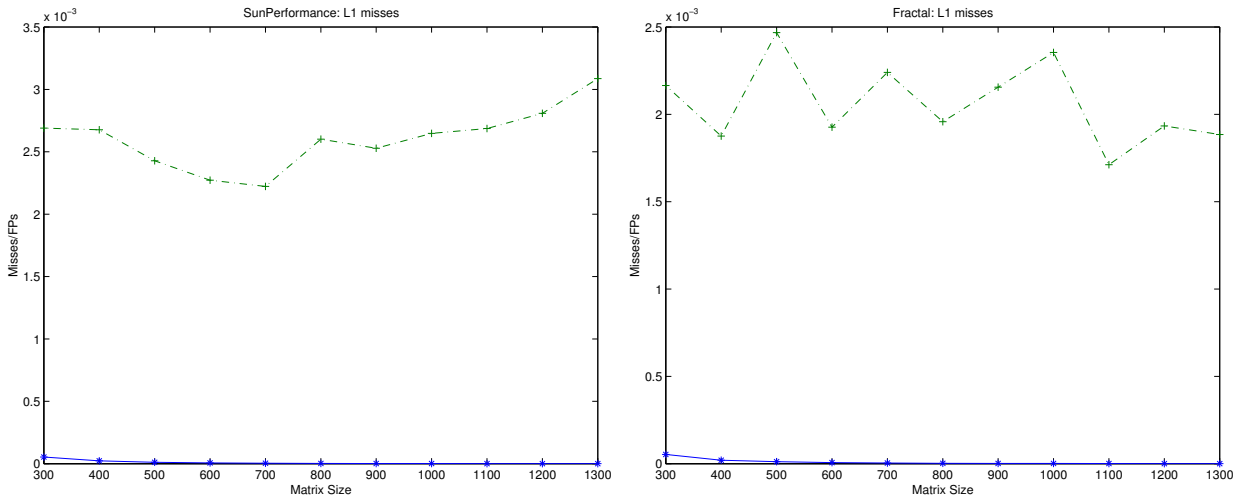

Figure 4.8: SPARC64: * code misses, + data misses

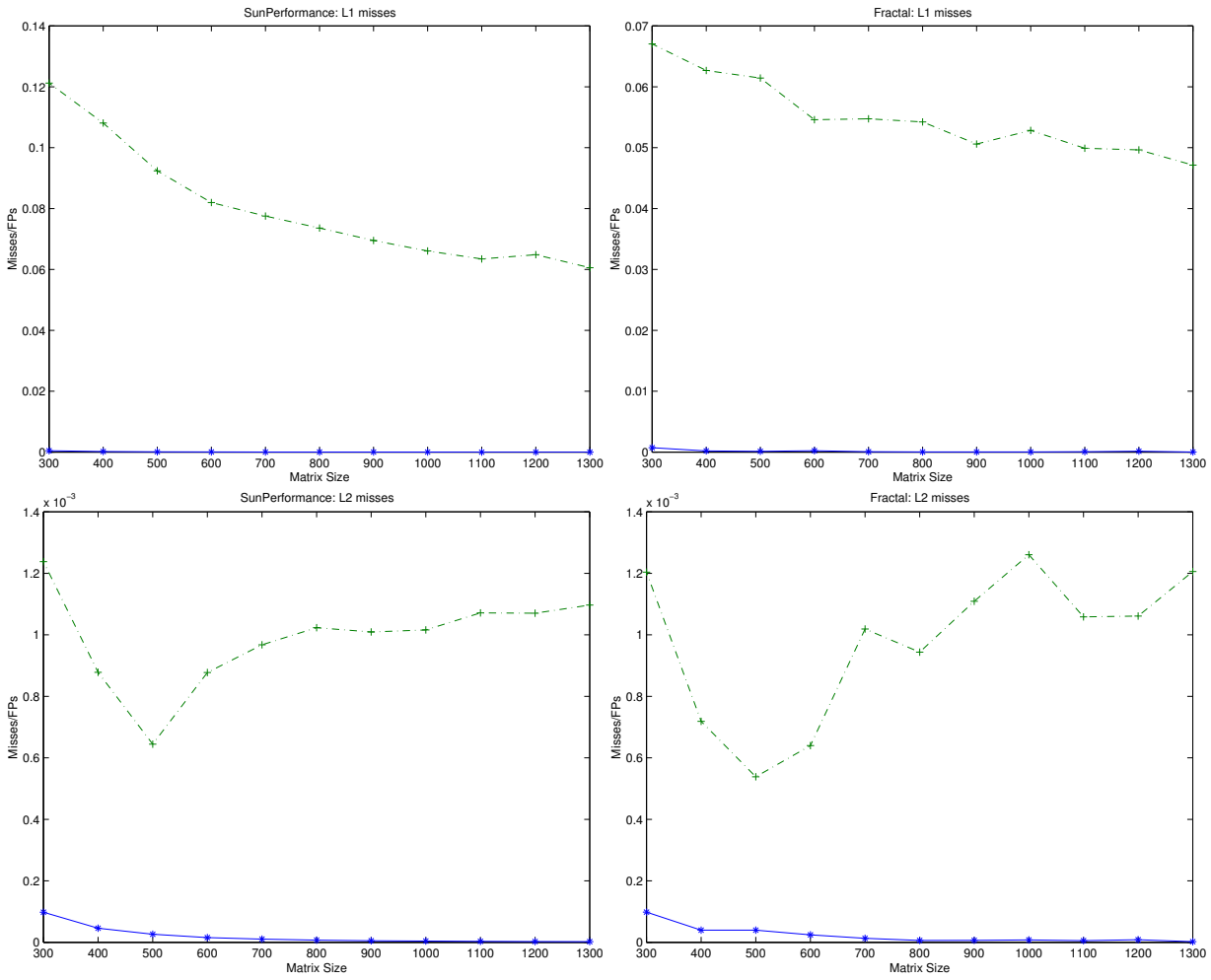

Figure 4.9: Ultra SPARC 5: * code misses, + data misses

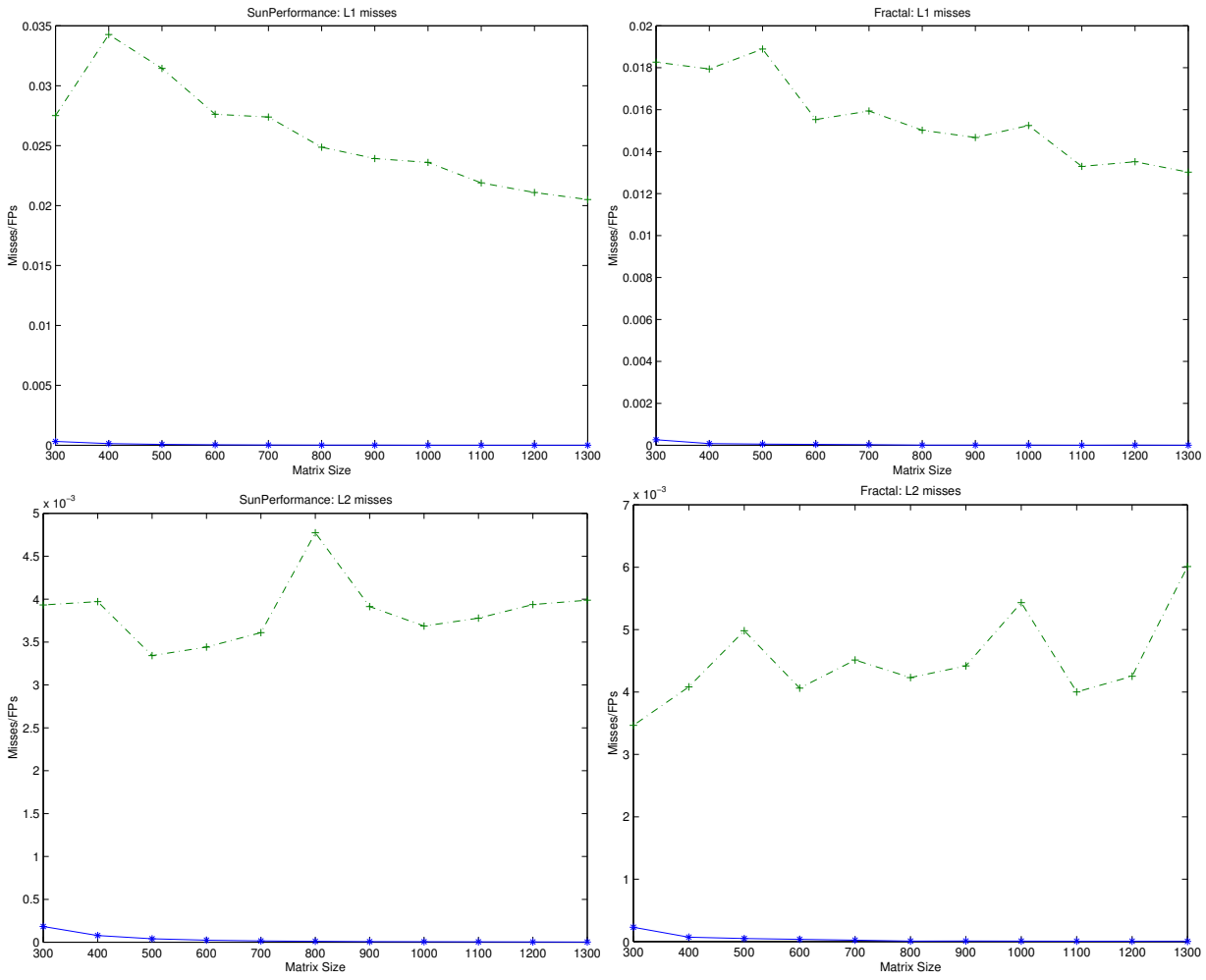

Figure 4.10: MIPS R5000: * code misses, + data misses

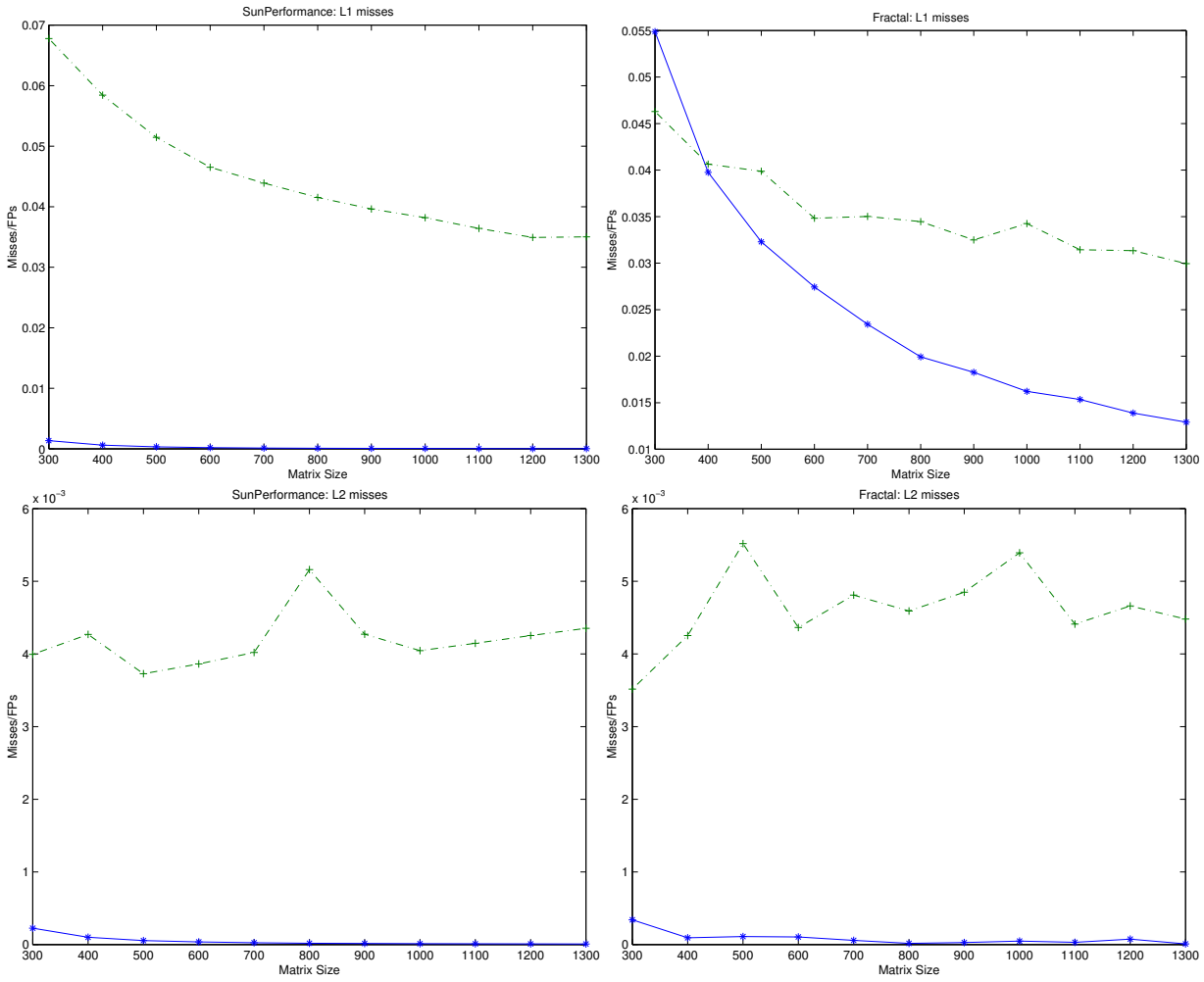

Figure 4.11: Pentium II, * code misses, + data misses

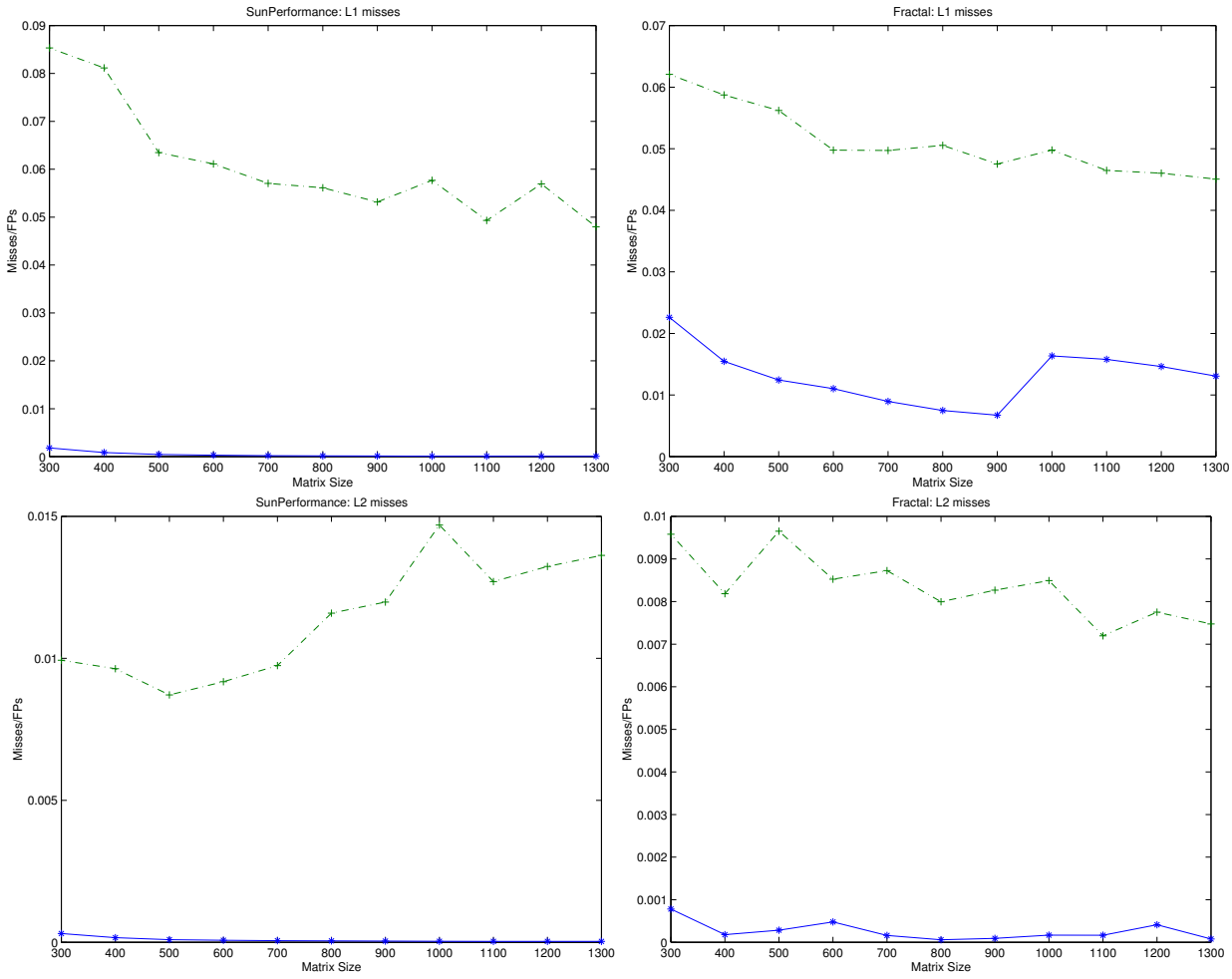

Figure 4.12: Alpha 21164: * code misses, + data misses

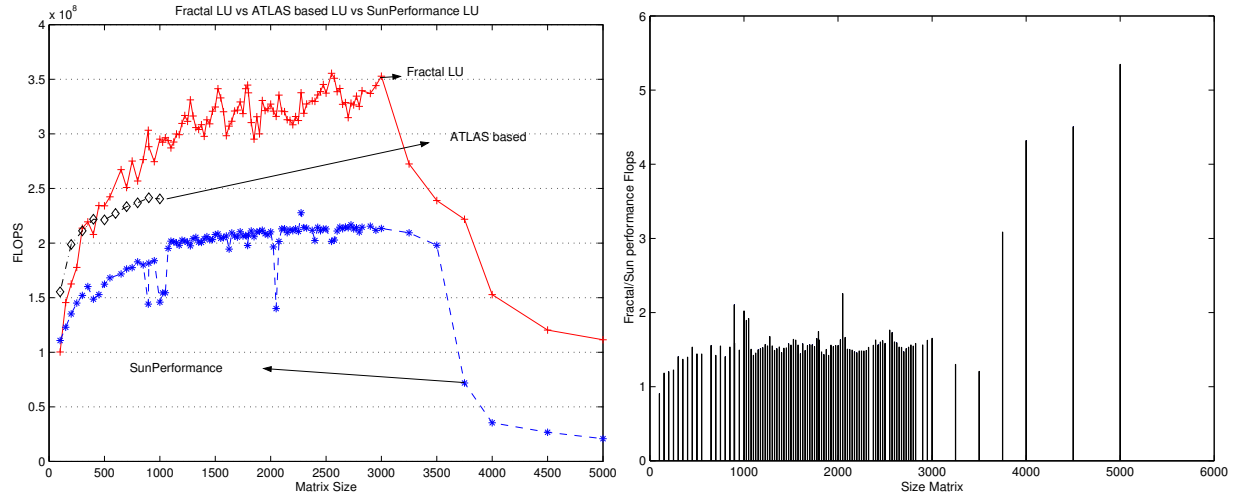

Figure 4.13: Ultra Sparc 5: FLOPS and relative performance

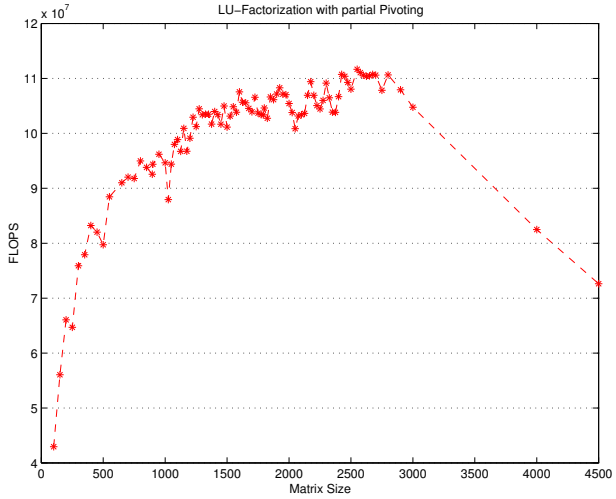

Figure 4.14: MIPS R5000: FLOPS

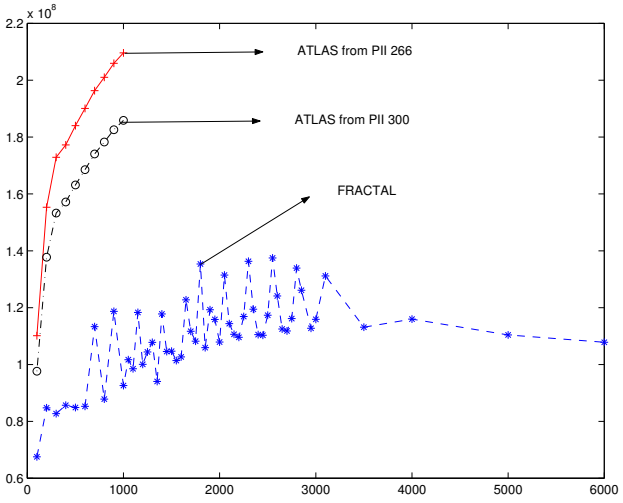

Figure 4.15: PentiumII: FLOPS

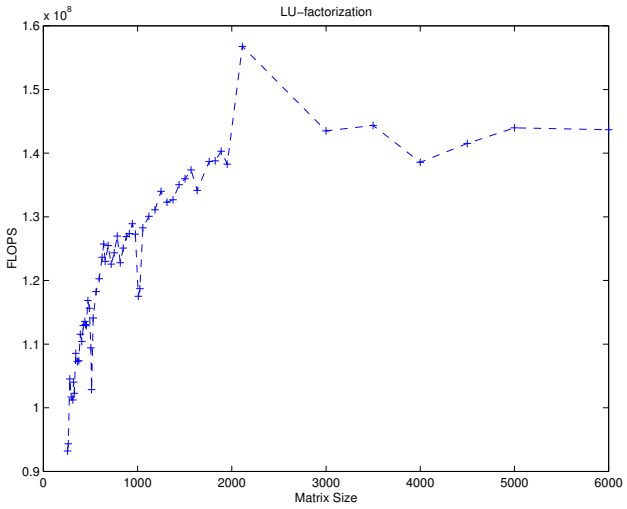

Figure 4.16: SPARC64: FLOPS

4.2 Matrix Multiply

In this section, we show the practical portability of a simple version of matrix multiplication (MM). Our algorithm is designed to exploit maximal and predictable locality at all levels of the memory hierarchy, with no *a priori* knowledge of the specific memory system organization for any particular system. By both simulations and execution on a number of platforms, we show that memory hierarchies portability does not sacrifice floating point performance; indeed, it is always a significant fraction of peak and, at least on one machine, is higher than the tuned routines by both ATLAS and vendor. The results are obtained by careful algorithm engineering, which combines a number of known as well as novel implementation ideas.

4.2.1 Fractal Algorithms for Matrix Multiplication

We store matrices using the fractal layout introduced previously in Section 4.1.3 for an $m \times n$ matrix $\mathbf{A}$ that we repeat here.

If $m < M$ and $n < M$ where M is a specific threshold, then $v[i * n + j] = a_{i,j}$. Otherwise, $\mathbf{v}$ is the ordered concatenation of the layouts of the submatrices $\mathbf{A}_0, \mathbf{A}_1, \mathbf{A}_2$, and $\mathbf{A}_3$ of the

following **balanced** composition.

- $\mathbf{A}_0 = \{a_{i,j} : 0 \leq i < \lceil m/2 \rceil, 0 \leq j < \lceil n/2 \rceil\}$,
- $\mathbf{A}_1 = \{a_{i,j} : 0 \leq i < \lceil m/2 \rceil, \lceil n/2 \rceil \leq j < n\}$,
- $\mathbf{A}_2 = \{a_{i,j} : \lceil m/2 \rceil \leq I < m, 0 \leq j < \lceil n/2 \rceil\}$ and
- $\mathbf{A}_3 = \{a_{i,j} : \lceil m/2 \rceil \leq i < m, \lceil n/2 \rceil \leq j < n\}$.

We remind also that a $m \times n$ matrix is **near square** when $|n - m| \leq 1$ and that if a matrix $\mathbf{A}$ is a near-square matrix, so are the submatrices $\mathbf{A}_0$, $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$ of its balanced composition.

We introduce now the fractal algorithms, a class of procedures all variants of a common scheme, for the operation of matrix multiply-and-add (MADD) $\mathbf{C} = \mathbf{C} + \mathbf{AB}$, also denoted $\mathbf{C}+ = \mathbf{AB}$. For near square matrices, the **fractal scheme** to perform $\mathbf{C}+ = \mathbf{AB}$ is recursively defined as follows, with reference to the above balanced composition.

fractal$(\mathbf{A}, \mathbf{B}, \mathbf{C})$

- If $dim(\mathbf{A}) = dim(\mathbf{B}) = 1$, then $c = c + a * b$ (all matrices being scalar).
- Else, execute - in any serial order - the calls **fractal**$(\mathbf{A}', \mathbf{B}', \mathbf{C}')$ for

$$(\mathbf{A}', \mathbf{B}', \mathbf{C}') \in \begin{array}{l} \{(\mathbf{A}_0, \mathbf{B}_0, \mathbf{C}_0), (\mathbf{A}_1, \mathbf{B}_2, \mathbf{C}_0), (\mathbf{A}_0, \mathbf{B}_1, \mathbf{C}_1), (\mathbf{A}_1, \mathbf{B}_3, \mathbf{C}_1), \\ (\mathbf{A}_2, \mathbf{B}_0, \mathbf{C}_2), (\mathbf{A}_3, \mathbf{B}_2, \mathbf{C}_2), (\mathbf{A}_2, \mathbf{B}_1, \mathbf{C}_3), (\mathbf{A}_3, \mathbf{B}_3, \mathbf{C}_3)\} \end{array}$$

Of particular interest, from the perspective of temporal locality, are those orderings where there is always a sub-matrix in common between consecutive calls, which increases data reuse. We model the problem of finding such orderings by an undirected graph. The vertices correspond to the 8 recursive calls in the fractal scheme. The edges join calls that share exactly one sub-matrix (notice that no two calls share more than one sub-matrix). This graph is a 3D binary cube. An Hamiltonian path in this cube is an ordering that maximizes data reuse, Figure 4.17.

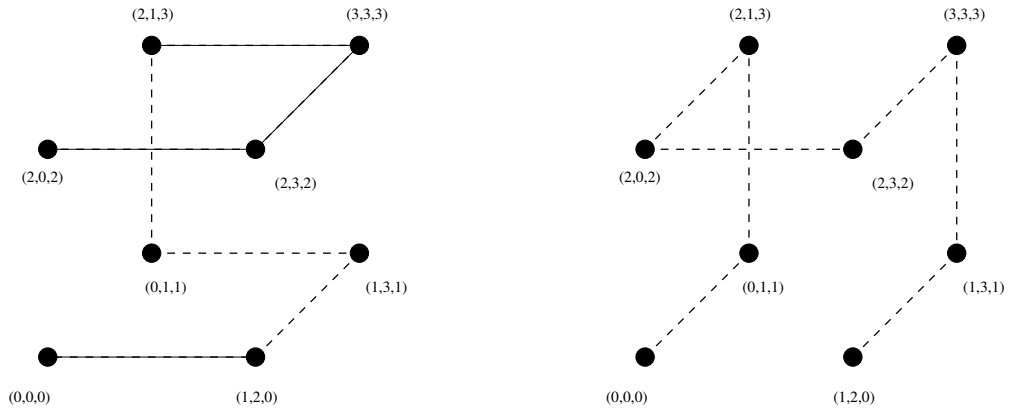

Figure 4.17: The cube of calls of the fractal scheme: the Hamiltonian path defining CAB-fractal and ABC-fractal.

Even when restricting our attention to Hamiltonian orderings, there are many possibilities. The exact performance of each of them depends on the specific structure and policy of the architecture cache(s) in a way too complex to evaluate analytically and too time consuming to evaluate experimentally. In this section, we shall focus on two orderings, Figure 4.17, that we identify as CAB-fractal and ABC-fractal. Briefly, the former reduces write misses and the latter reduces read misses.

CAB-fractal is the algorithm where the recursive calls are executed in the following order: ($\mathbf{A}_0$,$\mathbf{B}_0$,$\mathbf{C}_0$), ($\mathbf{A}_1$,$\mathbf{B}_2$,$\mathbf{C}_0$), ($\mathbf{A}_1$,$\mathbf{B}_3$,$\mathbf{C}_1$), ($\mathbf{A}_0$, $\mathbf{B}_1$, $\mathbf{C}_1$), ($\mathbf{A}_2$, $\mathbf{B}_1$, $\mathbf{C}_3$), ($\mathbf{A}_3$, $\mathbf{B}_3$, $\mathbf{C}_3$), ($\mathbf{A}_3$,$\mathbf{B}_2$,$\mathbf{C}_2$), ($\mathbf{A}_2$, $\mathbf{B}_0$, $\mathbf{C}_2$). The label "CAB" emphasizes that data sharing between consecutive calls is maximum for C (4 cases), medium for A (2 cases), and minimum for B (1 case). It is reasonable to expect that CAB-fractal will tend to better reduce write misses, since C is the matrix being written.

ABC-fractal is the algorithm where the recursive calls are executed in the following order: ($\mathbf{A}_0, \mathbf{B}_0, \mathbf{C}_0$), ($\mathbf{A}_0, \mathbf{B}_1, \mathbf{C}_1$), ($\mathbf{A}_2, \mathbf{B}_1, \mathbf{C}_3$), ($\mathbf{A}_2, \mathbf{B}_0, \mathbf{C}_2$), ($\mathbf{A}_3, \mathbf{B}_2, \mathbf{C}_2$), ($\mathbf{A}_3, \mathbf{B}_3, \mathbf{C}_3$), ($\mathbf{A}_1, \mathbf{B}_3, \mathbf{C}_1$), ($\mathbf{A}_1, \mathbf{B}_2, \mathbf{C}_0$).

Cache Performance

Fractal multiplication algorithms can be implemented with respect to any memory layout of the matrices. For an ideal fully associative cache with least recently used replacement policy (LRU) and with cache lines holding exactly one matrix element, the layout is immaterial to performance. The fractal approach exploits temporal locality for any cache independently of its size s (in matrix entries). Indeed, consider the case when at the highest level of recursion all calls use sub matrix that fit in cache simultaneously. Approximately, the matrix blocks are of size $s/3$. Each call load will cause about s misses. Each call computes up to $(\sqrt{s/3})^3 = s\sqrt{s}/3\sqrt{3}$ scalar MADDs. The ratio misses per FLOP is estimated as $\mu = (3\sqrt{3}(/(2\sqrt{s}) \approx 2.6/\sqrt{s}$. (This is within a constant factor of optimal, Corollary 6.2 [11].)

For a real system, the above analysis needs to be refined, keeping into account the effects of cache-line size ℓ (in matrix entries) and a low degree of associativity. Here, the fractal layout, which stores relevant sub matrix in contiguous memory locations, takes full advantage of cache-line effects and has no self interference for blocks that fit in cache. The misses per flop is estimated as $\mu = 2.6\gamma/\ell\sqrt{s}$, where γ accounts for cross interference between different matrices and other fine effects not captured by our analysis. In general, for a given fractal algorithm, γ will depend on matrix size (n), relative fractal arrays positions in memory, cache associativity and, sometimes, register allocation. When interference is negligible, we can expect $\gamma \approx 1$.

The Structure of the Call Tree

Pursuing efficient implementations for the fractal algorithms, we now face the performance drawbacks of recursion: overheads and poor register utilization (due to lack of code exposure to the compiler). To circumvent such drawbacks, we carefully study the structure of the **call tree**; that is, how the recursive algorithm unfold in time.

Definition 1 *Given a fractal recursive algorithm, its* **call tree** $T = (V, E)$ *is an ordered*

and rooted tree with input matrices $(\mathbf{A}, \mathbf{B}, \mathbf{C})$. *In fact,* V *contains one node for each call and the root of* T *corresponds to the main call* **fractal***(A,B,C). Each node has up to eight ordered children,* $v_1, v_2, \ldots, v_8$ *and, in practice, a node with children, also an* **internal node***, corresponds to the call made in order of execution.*

If matrix $\mathbf{A}$ is $m \times n$ and $\mathbf{B}$ is $n \times p$, the input is of **size** $< m, n, p >$ (the triplet specifies the problem size). If one among m, n, and p is zero, then the size is zero and we use the short notation $< \emptyset >$. The structure of T is uniquely determined by size of the root. We focus on square matrices, that is, a problem of size $< n, n, n >$ for which the tree has depth $\lceil \log n \rceil + 1$ and it has $8^{\lceil \log n \rceil}$ leaves. In fact, n^3 leaves have type $< 1, 1, 1 >$ and correspond (from left to right) to the n^3 MADDs of the algorithm. The remaining leaves have zero size. Internal nodes are essentially responsible for performing the problem division. An internal node has typically eight non-empty children, except when its size has at least one components equal to 1 (e.g., $< 2, 1, 1 >$ or $< 2, 2, 1 >$ in which the non empty children are 2 and 4, respectively).

The call tree has $O(n^3)$ nodes and most of them have the same size. To deal with this issue systematically, we apply the concept of recursion-DAG. Given a fractal algorithm, an input size $< m, n, p >$, and the corresponding call tree $T = (V, E)$, then we identify with $D = (U, F)$ the recursion-DAG. See Figure4.18 for an example.

Next, we study the size of the recursion-DAG D for the case of square matrix multiplication. We begin by showing that there are at most 8 types of input for the calls of a given level of recursion.

Proposition 1 *For any integers* $n \geq 1$ *and* $d \geq 0$, *let* n_d *be defined inductively as* $n_0 = n$ *and* $n_{d+1} = \lceil n_d/2 \rceil$. *Also, for any integer* $q \geq 1$, *define the set of types* $Y(q) = \{< r, s, t >: \ r, s, t \in \{q, q-1\} \}$. *Then, in the call tree corresponding to a type* $< n, n, n >$, *the type of each call-tree node at distance* d *from the root belongs to the set* $Y(n_d)$, *for* $d = 0, 1, \ldots, \lceil \log n \rceil$.

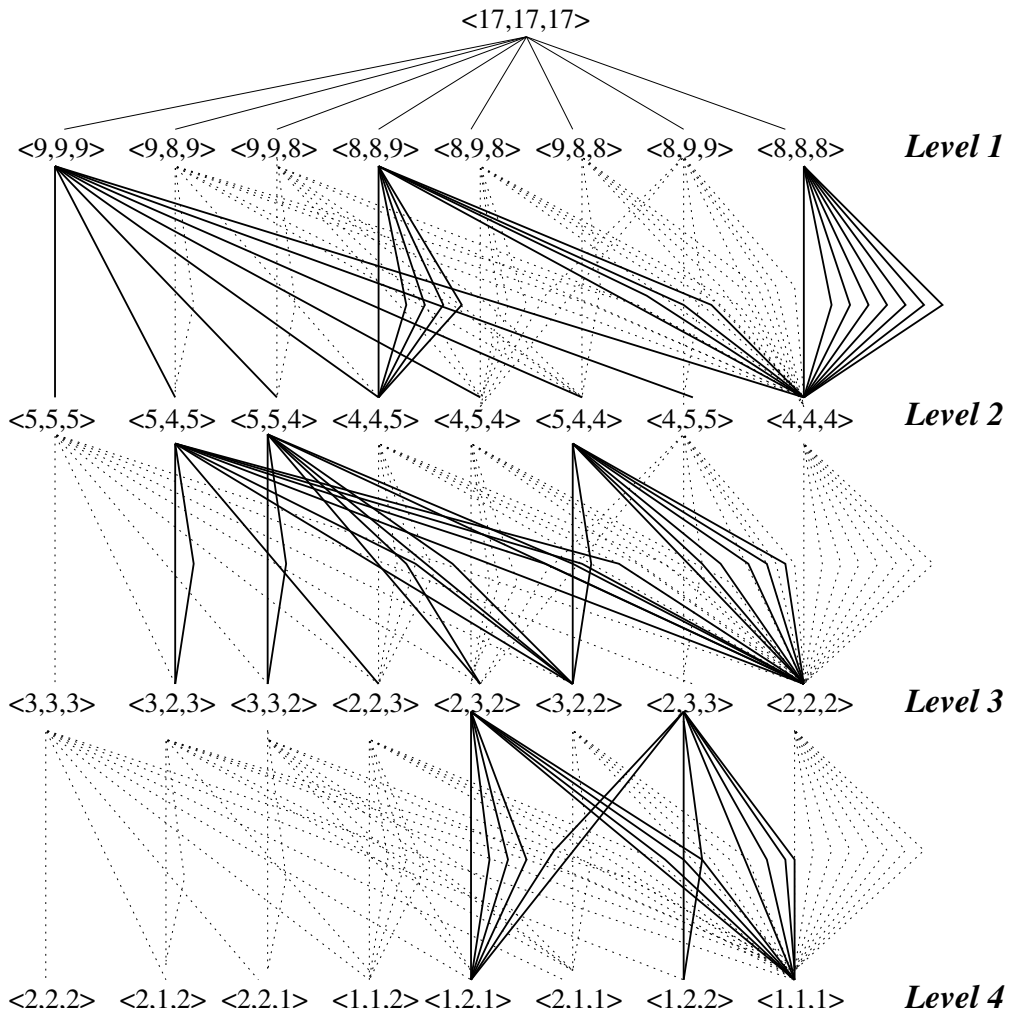

Figure 4.18: Example of call-recursion-DAG for Matrix Multiplication $< 17, 17, 17 >$

Proof: The statement trivially holds for $d = 0$ (the root), since $< n, n, n > \in Y(n) = Y(n_0)$. Assume now inductively that the statement holds for a given level d. From the closure property of the balance decomposition and the recursive decomposition of the algorithm, it follows that all matrix blocks at level $d + 1$ have dimensions between $\lfloor (n_d - 1)/2 \rfloor$ and $\lceil n_d/2 \rceil$. From the identity $\lfloor (n_d - 1)/2 \rfloor = \lceil n_d/2 \rceil - 1$, we have that all types at level $d + 1$ belong to $Y(\lceil n_d/2 \rceil) = Y(n_{d+1})$. □

Now, we can give an accurate size estimate of the call-recursion-DAG.

Proposition 2 *Let n be of the form $n = 2^k s$, with s odd. Let $D = (U, F)$ be the call-recursion-DAG corresponding to input type $< n, n, n >$. Then, $|U| \leq k + 1 + 8(\lceil \log n \rceil - k)$.*

Proof: It is easy to see that, at level $d = 0, 1, \ldots, k$ of call tree nodes have type $< n_d, n_d, n_d >$, with $n_d = n/2^d$. For each of the remaining $(\lceil \log n \rceil - k)$ levels, there are at most 8 types per level, according to Proposition 1. □

Thus, we always have $|U| = O(\log n)$, with $|U| = \log n + 1$ when n is a power of two, with $|U| \approx 8\lceil \log n \rceil$ when n is odd, and with $|U|$ somewhere in between for general n.

Bursting the Recursion

If u is an internal node of the call tree, the corresponding call receives as input a triplet of matrices **A**, **B**, and **C**, and produces as output the input for each child call. The input triplet is uniquely determined by the size $< r, s, t >$ and by the initial addresses a_i, b_j, and c_k of the submatrices.

The submatrix of A is stored in between a_i and $a_i + rs - 1$, the submatrix of B is stored in between b_j and $b_j + st - 1$, and the submatrix of C is stored in between c_k and $c_k + rt - 1$. The call at u is then responsible for the computation of the size and initial position of the sub-blocks processed by the children. For example, for the matrix A of size $r \times s$ starting at a_i, the four submatrices have respective dimensions and starting points:

Size	Starting at $a_{i_h} = a_i + \Delta i_h$
$\lceil r/2 \rceil \times \lceil s/2 \rceil$	$\Delta i_0 = 0$
$\lceil r/2 \rceil \times \lfloor s/2 \rfloor$	$\Delta i_1 = \lceil r/2 \rceil \lceil s/2 \rceil$
$\lfloor r/2 \rfloor \times \lceil s/2 \rceil$	$\Delta i_2 = \lceil r/2 \rceil s$
$\lfloor r/2 \rfloor \times \lfloor s/2 \rfloor$	$\Delta i_3 = \Delta i_2 + \lfloor r/2 \rfloor \lceil s/2 \rceil$

In a similar way, we can define the analogous quantities $b_{j_h} = b_j + \Delta j_h$ for the sub-blocks of B, and $c_{k_h} = c_k + \Delta k_h$ for the sub-blocks of C, for $h = 0, 1, 2, 3$. Notice that during the recursion and in any node of the call tree, every Δ value is computed twice, and, thus,

the recursive algorithm has redundant work. We may improve performance of the recursive algorithm following two avenues, separately or in combination. First, rather than executing the full call tree down to the n^3 leaves of type $< 1, 1, 1 >$, we can execute a pruned version of the tree. This approach reduces the recursion overheads and the straight-line coded leaves are amenable to aggressive register allocation, a subject of Section 4.2.2. Second, the integer operations are mostly the same for all recursive calls. Hence, these operations can be performed in a preprocessing phase, storing the results in an auxiliary data structure built around the recursion-DAG D, to be accessed during the actual processing of the matrices. Counting the number of instructions per node, we can see a reduction of 30%.

4.2.2 Register Issues

The effect of register management on the overall performance is captured by the number ρ of memory (load or store) operations per floating point operation, required by a given assembly code. In a single-pipeline machine with at most one FP or memory operation per cycle, $1/(1 + \rho)$ is an upper limit to the achievable fraction of FP peak performance. The fraction lowers to $1/(1 + 2\rho)$ for machines where MADD is available as a single-cycle instruction. For machines with parallel pipes, say 1 load/store pipe every f FP pipes, an upper limit to the achievable fraction of FP peak performance becomes $\max(1, f\rho)$, so that memory instructions are not a bottleneck as long as $\rho \leq 1/f$. In this section, we explore two techniques which, for the typical number of registers of current RISC processors, lead to values of ρ approximately in the range 1/4 to 1/2. The general approach consists in stopping the recursion at some point and formulating the corresponding leaf computation as a straight-line code. All matrix entries are copied into a set of scalar variables, **scalarization**, whose number R is chosen so that a compiler will keep these variables in registers. For a given R, the goal is then to choose where to stop the recursion and how to sequence the operations so as to minimize ρ (i.e., to minimize the number of assignments to and from scalar variables).

Fractal Sequences. One approach consists in sequencing the operations in the order that arises from the fractal scheme when the recursive process is followed all the way down to $< 1, 1, 1 >$ leaves. We have heuristically explored sequences that arise from changing the order of the subproblems at different nodes of the recursion trees (e.g., from ABC to CAB), generalizing an approach proposed in [6] (for caches). As an indication, for a $< 4, 4, 4 >$ leaf we obtain $\rho = 0.5$ (with $R = 28 - 32$) and for a $< 32, 32, 32 >$ leaf we obtain $\rho = 0.33$ (with $R = 32$).

C-tiling Sequences. The C-tiling approach, which generalizes the register allocation proposed in [103], partitions the result matrix of a generic $< m, n, p >$ leaf multiplication into rectangular tiles. An $r \times s$ submatrix $\mathbf{C}$ is the product of an $r \times n$ submatrix of $\mathbf{A}$ and an $n \times s$ submatrix of $\mathbf{B}$ and, hence, can be expressed as the sum of n terms, each term is a product of a column of the $\mathbf{A}$ submatrix by a row of the $\mathbf{B}$ submatrix. If $R \geq rs + r + 1$ registers (scalar variables) are available: first, we load the C submatrix into rs registers; second, we load one at the time the n A-subcolumns into r registers; third, we load one at the time the elements of the corresponding $\mathbf{B}$-subcolumn and execute the r madds involving it and the elements of $\mathbf{A}$ currently in registers; and, finally, we store back the $\mathbf{C}$ submatrix. The number of accesses is $2rs + n(r + s)$ and the number of FP operations is $2rsn$, yielding $\rho = \frac{1}{n} + \frac{1}{2r} + \frac{1}{2s}$. The value of ρ for the full $< m, n, p >$ product is a sort of average over the chosen tiles, which might be of different sizes especially at the boundaries of the tiled submatrices. As an indication, for an $< 8, 8, 8 >$ leaf we obtain $\rho = 0.50$ (with $R \geq 21$) and for a $< 32, 32, 32 >$ leaf we obtain $\rho = 0.25$ (with $R \geq 32$).

4.2.3 Experimental Results

We have studied experimentally both the cache behavior of fractal algorithms, in terms of misses, and the overall performance, in terms of running time.

Cache Misses

The results of this section are based on simulations performed (on an SPARC Ultra 5) using the *Shade* software package for Solaris, of Sun Microsystems. Codes are compiled for the SPARC Ultra2 processor architecture (V8+, no MADD operation available) and then simulated for various cache configurations, chosen to correspond to those of a number of commercial machines. Thus when we refer, say, to the R5000 IP32, we are really simulating an ultra2 CPU with the memory hierarchy of the R5000 IP32.

In practice, we stop the recursion when the size of the leaves is strictly smaller than problem $< 32, 32, 32 >$. We set the threshold for the the recursive layout to 32. We implement the computation leaves using the C-tiling register assignment using $R = 24$ variables for scalarization and this approach leaves to the compiler 8 of the 32 registers to buffer multiplication outputs before they are accumulated into C-entries. We opted to compile the computation leaf codes with cc WorkShop 4.2 and linked statically (as suggested in [103]) for its superior instruction scheduling. However, we compiled the recursive codes using gcc 2.95.1.

We have also simulated the code for ATLAS DGEMM obtained by installation of the package on the Ultra 5 architecture. This is used as another term of reference, and, in general, fractal has fewer misses. However, it would be unfair to regard this as a competitive comparison with ATLAS, which should be re-installed for each different cache configuration.

We have simulated 7 different cache configurations (Table 4.3); we use the notation: I= Instruction cache, D=Data cache, and U=Unified cache. We have measured the number $\mu(n)$ of misses per flop and compared it against the value of the estimator (Section 4.2.1) $\mu(n) = 2.6\gamma(n)/(\ell\sqrt{s})$, where s and ℓ are the number of (64 bit) words in the cache and in one line, respectively, and where we expect values of $\gamma(n)$ not much greater than one. In Table 4.3, we have reported the value of $\mu(1000)$ measured for CAB-fractal and the corresponding value of $\gamma(1000)$ (last column). We can see that γ is generally between 1 and 2; thus, our

Table 4.3: Summary of simulated configurations

Configuration	**Cache Size**	**Line Size**	**Way - Write**	$\mu(1000)$ - $\gamma(1000)$
SPARC 1				
U1	64K	16	1 - through	2.65e-2 - 1.84
SPARC 5				
I1	16K	16	1	
D1	8K	16	1 - through	5.96e-2 - 1.47
Ultra 5				
I1	16K	32	2	
D1	16K	32	1 - through	2.51e-2 - 1.75
U2	2M	64	1 - back	1.05e-3 - 1.66
R5000 IP32				
I1	32K	32	2 - back	
D1	32K	32	2 - back	1.06e-2 - 1.04
U2	512K	32	1 - back	3.61e-3 - 1.42
Pentium II				
I1	16K	32	1	
D1	16K	32	1 - through	2.50e-2 - 1.74
U2	512K	32	1 - back	3.98e-3 - 1.57
HAL Station				
I1	128K	128	4 - back	
D1	128K	128	4 - back	2.65e-3 - 2.09
Alpha 21164				
I1	8K	32	1	
D1	8K	32	1 - through	3.75e-2 - 1.85
U2	96K	32	3 - back	5.81e-3 - 0.99

estimator gives a reasonably accurate prediction of cache performance. This performance is consistently good on the various configurations, indicating efficient portability. We also have simulation results for code misses: although these misses do increase due to the comparatively large size of the leaf procedures, they remain negligible with respect to data misses and, thus, we omitted them.

MFLOPS

Of course, portability of cache performance is desirable, however, it is more important to explore the extent of these optimizations towards performance –execution time. We have

tested the fractal approach on four different processors listed in Table 4.4. We always use the same code for the recursive algorithm, which is responsible for cache behavior essentially. We vary the code for the leaves, to adapt the number of scalar variables R to the processor: $R = 24$ for Ultra 5, $R = 8$ for Pentium II, and $R = 32$ for SGI R5K IP32 and HAL Station. We compare the MFLOPS of fractal algorithms in double precision with peak performance and with the performance of ATALS-DGEMM, if available. Fractal achieves performances comparable to those of ATLAS, being at most 2 times slower on PentiumII (which is not a RISC) and a little faster on SGI R5K. Since no special adaptation to the processor has been performed on the fractal codes, except for the number of scalar variables, we conclude that the portability of cache performance can be combined with overall performance.

Table 4.4: Processor Configurations

Processor	Ultra 2i	PentiumII	R5000	SPARC64
Registers	32	8	32	32
MUL/ADD - latency	separate - 3	separate - 8	merged - 2	merged - 4
Peak (MFLOPS)	666	400	360	200
Peak Fractal - size	425 - 444	187 - 400	133 - 504	168 - 512
Peak ATLAS - size	455 - 220	318 - 848	113 - NA	NA

4.3 Fast Fourier Transform

In this section, we identify a n-point Discrete Fourier Transform as $\mathbf{DFT}(n)$, which is a matrix-by-vector product $\mathbf{y} = \mathbf{\Omega}_n * \mathbf{x}$ with $\mathbf{x}, \mathbf{y} \in \mathcal{C}^n$ and $\mathbf{\Omega}_n \in \mathcal{C}^{n \times n}$. In fact, we may determine one component of $\mathbf{y}$ as the sum: $y_k = \sum_{r=0}^{n-1} x_i \omega_n^{rk}$, where ω_n^{rk} is called **twiddle factor** and ω_n is the complex number $e^{\frac{i}{n}}$.

When n is the product of two factors such as p and q (i.e., $n = pq$), we may apply Cooley-Tookey's algorithm. The input vectors $\mathbf{x}$ can be seen as $q \times p$ matrix $\mathbf{X}$ stored in row major. The result is computed in place in $\mathbf{X}$. We can write the $DFT(n)$ algorithm as follows:

1. for every $i \in [0, p-1]$ we compute $\mathbf{X}_{[0,q-1],i} = \mathbf{\Omega}_q \mathbf{X}_{[0,q-1],i}$ – this is a computation on the columns of matrix $\mathbf{X}$;

2. distribute the twiddle factors $x_{i,j} * = \omega_n^{ij}$;

3. for every $i \in [0, q-1]$ we compute $\mathbf{X}_{i,[0,p-1]} = \mathbf{\Omega}_p \mathbf{X}_{i,[0,p-1]}$;

Algorithms implementing $DFT(n)$ on $n = 2^\gamma$ points are well studied. However, they may be inefficient on a cache using direct mapping $f(x) = x$ **mod** S, where $S = 2^k$. Interference prevents the spatial locality exploitation between the computation of $\mathbf{X}_{[0,q-1],i} = \mathbf{\Omega}_q \mathbf{X}_{[0,q-1],i}$ and $\mathbf{X}_{[0,q-1],i+1} = \mathbf{\Omega}_q \mathbf{X}_{[0,q-1],i+1}$. In fact, two elements in the same column of X will be mapped to the same cache line (self interference), therefore preventing the spatial reuse across the computation of $X_{[0,q-1],i}$ and $X_{[0,q-1],i+1}$.

The number of misses due to interference are relatively few, but for large n and in a multilevel memory hierarchy, they are misses at every level –e.g., memory pages too. Any improvement in the number of misses at the first level of cache, even small, is very beneficial for a multilevel cache system.

Implementations, such as FFTW [12] and SPIRAL [104], may exploit temporal locality

through copying the input data on a temporary work space. This approach allows a perfect data reuse by the following computations, but interference makes extremely slow the first access therefore the copy. We propose a solution to nullify interference in Section 6.3. Here, we present the basic structure of the computation using recursion-DAG, which will be used to solve the interference problem.

We decide to work with balanced FFT, that is, given n points we choose the factors p and q so that $\min_{n=p*q} |p - q|$. Such balanced division exploits a short call tree, of height $O(\log_2 \log_2 n)$, however its performance has a time complexity of $O(n \log_2 n)$ (see also [3, 4]). The calls tree of FFT is similar to the calls tree for matrix multiplication and the recursion-DAG size is moderate, $O(\log_2 n)$.

In the leaves of the computation we use the codelets from FFTW, where the twiddle factors are precomputed. Our implementation for n prime computes the twiddle factors on the fly, it is based on a loop nest with the inner loop unrolled twice. We decide this solution for n prime to exploit temporal locality and parallelism even in the worst case. In fact, this solution achieves 50% of the peak performance on a Blade 100.

In Figure 4.19 we present the actual implementation of FFT. The implementation is simple and elegant. The `Tree` structure is determined off-line. If two sub trees have the same number of points, they may be merged into one node. For some n-FFT the `Tree` may be represented by a linked list.

We shall present the performance of our implementation only in Section 6.3.4, where we shall discuss the capability to perform efficiently other important computations: then, we shall consider again the algorithm and we shall give a clear presentation of the context and performance.

```
typedef void (* Leaf_computation)(const Point *, Point *,int , int );

typedef struct decomposition_tree* Tree;

struct decomposition_tree {
  int n;
  Tree left;
  Tree right;

  Leaf_computation leaf;
};

/************
* Recursive FFT
*
*/
void fft_tree(Point *V, Tree t, int stride) {
   int p,q,i,n;


   n = t->n;
   if (t->leaf) {  /* leaf */
     if (n>LEAF) {
       t->leaf(V,V,stride,n); /* FFTW codelets */
       return;
     }
     else {
       t->leaf(V,V,stride,stride); /* large prime: ad hoc loop nest */
       return;
     }
   }
   else {   /* recursive call */
     p = t->left->n;
     q = t->right->n;

     for (i=0;i<q;i++)
       fft_tree(V+i*stride,t->left,q*stride);

     distribute_twiddles(V,n,p,q);

     for (i=0;i<p;i++)
       fft_tree(V+i*stride*q,t->right,stride);

   }
}
```

Figure 4.19: Our implementation of FFT.

4.4 All Pair Shortest Path and Matrix Multiply

We propose a novel divide-and-conquer (D&C) algorithm for the solution of all-pair shortest path (APSP) for graphs with no negative cycles and represented by (dense) adjacency matrices.

Using registers and data layout optimizations, we derive a compact, portable, and in-place recursive algorithm that yields optimal performance. We show that our algorithm delivers competitive performance for both small and very large adjacency matrices; for example, our algorithm yields between 1/2 and 1/7 of the peak performance for 6 architectures.

4.4.1 A Recursive D&C Algorithm, R-Kleene

In this section, we present a recursive D&C algorithm derived from Kleene's algorithm Figure 4.20.(a). Notice that Kleene's algorithm was originally designed to solve the **transitive closure** (TC) of an adjacent matrix. That is, finding whether or not there is a path connecting two nodes in directed graph. However, Kleene's algorithm is also a **standard** algorithm/solution for the APSP. In fact, in a closed semiring TC and APSP are the same problem and Kleene's algorithm is a solution (when the scalar operators $*$ and $+$ are specified as in the following paragraph) and it determines every edge of the (shortest) path directly [64].

A brief description of Kleene's algorithm follows. We divide the basic problem into two sub-problems, and we solve each problem directly using the Floyd-Warshall algorithm, see Figure 4.20.(a). Then, we perform several MMs to combine the results of the subproblems using a temporary matrix. Formally, matrix multiplication $\mathbf{E} += \mathbf{FG}$ (with $\mathbf{E}, \mathbf{F}, \mathbf{G} \in \mathbb{Z}^{n \times n}$) is simply $e_{i,j} += \sum_{k=0}^{n-1} f_{i,k} * g_{k,j}$; where the scalar addition of two numbers is actually the minimum of the two numbers –i.e., $a + b = \min(a, b)$– and the scalar multiplication of two numbers is the (regular arithmetic) addition –i.e., $a * b = a + b$.

In this case, the MM is defined in a closed semiring and using the properties within,

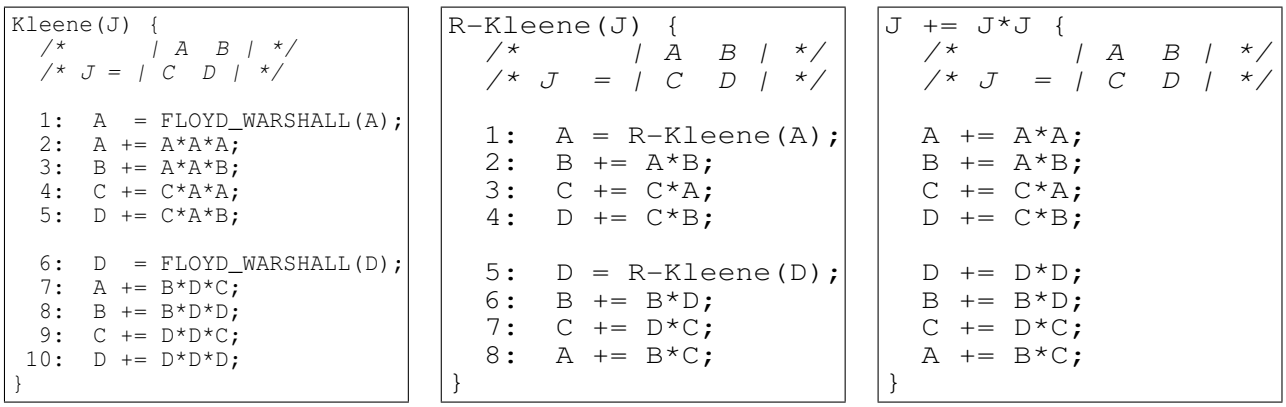

Figure 4.20: (a) Kleene, (b) R-Kleene and (c) (Self) Matrix Multiply

we reorganize the algorithm in Figure 4.20.(a) and obtain the R-Kleene algorithm in Figure 4.20.(b). Thus, R-Kleene is solution for APSP in a closed semiring.

In the following, we explain in seven steps how to achieve the algorithm in Figure 4.20.(b):

1. We start by noticing that the computations on line 2 and 10 in Figure 4.20.(a), (i.e., $\mathbf{A}+ = \mathbf{A} * \mathbf{A} * \mathbf{A}$ and $\mathbf{D}+ = \mathbf{D} * \mathbf{D} * \mathbf{D}$) do not have any effect on matrix $\mathbf{A}$ and $\mathbf{D}$ respectively, because each is a matrix closure.

   A formal proof follows. First, $\mathbf{AA}$ is equal to $\mathbf{A}$; in fact, consider $\mathbf{F} = \mathbf{AA}$, $f_{i,j} = \sum_{k=0}^{n-1} a_{i,k} * a_{k,j}$; because $\mathbf{A}$ is matrix closure, $f_{i,j} = a_{i,j} * a_{j,j} + \sum_{k \neq j} a_{i,k} * a_{k,j} = a_{i,j} + \sum_{k \neq j} a_{i,k} * a_{k,j} = a_{i,j} + \sum_{k \neq j} a_{i,j} = a_{i,j}$, therefore $\mathbf{F} = \mathbf{A}$. Moreover, because $+$ is idempotent (i.e., $a + a = a$) we have that $\mathbf{A} + \mathbf{A}$ is equal to $\mathbf{A}$.

2. Notice that the property that $\mathbf{AA} = \mathbf{A}$ –when $\mathbf{A}$ is matrix closure– is applied elsewhere such as on line 3 in Figure 4.20.(a), which becomes the operation on line 2 in Figure 4.20.(b).

3. Moreover, consider the computation on line 5 in Figure 4.20.(a) $\mathbf{D}+ = \mathbf{CAB}$, this is

equivalent to $\mathbf{D}+=\mathbf{CB}$ –Figure 4.20.(b), on line 4.

A formal proof follows. First, we show that $\mathbf{CA} \equiv \mathbf{C}+\mathbf{CA}$; in fact, consider $\mathbf{F}=\mathbf{CA}$, then $f_{i,j} = \sum_{k=0}^{n-1} c_{i,k} * a_{i,j} = c_{i,j} * a_{j,j} + \sum_{k \neq j} c_{i,k} * a_{i,j}$; because $a_{j,j}=0$, we have $f_{i,j} = c_{i,j} + \sum_{k \neq j} c_{i,k} * a_{i,j} = c_{i,j} + \sum_{k=0}^{n-1} c_{i,k} * a_{i,j}$, so $\mathbf{CA} \equiv \mathbf{C}+\mathbf{CA}$. We conclude by noticing that $\mathbf{D}+=(\mathbf{C}+\mathbf{CA})\mathbf{A}(\mathbf{B}+\mathbf{AB})$ is equivalent to $\mathbf{D}+=(\mathbf{CA}+\mathbf{CAA})(\mathbf{AB}+\mathbf{AAB}) = (\mathbf{CA})(\mathbf{AB}) = (\mathbf{C}+\mathbf{CA})(\mathbf{B}+\mathbf{AB})$.

4. Similarly, we obtain the simplified computation $\mathbf{D}+=\mathbf{CB}$ (i.e., line 4 in Figure 4.20.(b)).

5. Moreover following similar reasoning, we may postpone the computation of line 7 in Figure 4.20.(a), as last on line 8 in Figure 4.20.(b),

6. The last step is to apply the idea recursively on $\mathbf{A}$ and $\mathbf{D}$.

If we look at the algorithm in Figure 4.20.(b), ultimately, this is similar to the recursive algorithm for MM, Figure 4.20.(c). Though, Kleene's algorithm and the algorithm proposed by Park et al. [19], which is basically the algorithm in Figure 4.20.(c), impose a strict order to the function calls, however MM algorithms (in general) and our algorithm do not require. In fact, we shall explain shortly that our algorithm is bound loosely to the function call order.

It is evident that the computations on line 2 and 3 in Figure 4.20.(b) can be executed in any order and in parallel. We now show that the MMs involved in the computation have no restrictions and they may exploit a different order than the one specified in Figure 4.20.(c). This is important because certain low-level-optimizations for the solution of MM need to rearrange the computation order, in general, it is quite different from the algorithm in Figure 4.20.(c), so as to exploit better locality and performance. Indeed, we show in

Theorem 2 that we may choose any order and, thus, the order that exploits data reuse in registers.

Theorem 2 *In a closed semiring the matrix multiplications* $\mathbf{B}+ = \mathbf{AB}$ *or* $\mathbf{B}+ = \mathbf{BA}$ *can be done in place and in any evaluation order if* $\mathbf{A}$ *is a matrix closure.*

A formal proof follows, however we explain the first case $\mathbf{B}+ = \mathbf{AB}$ only. Consider two elements in $\mathbf{B}$ during the in-place computation. Without loss of generality, consider the entries $b_{0,j}$ and $b_{1,j}$, which belong to the same column of $\mathbf{B}$ and necessarily are needed for the computation of each other. Assume we compute first $b_{0,j} = a_{0,m} * b_{m,j}$, and then, we compute $b_{1,j} = a_{1,n} * b_{n,j}$. If we assume that $b_{1,j}$ affects the previous shortest path, then we must recompute $b_{0,j}$, which should be $\dot{b}_{0,j} = a_{0,1} * b_{1,j}$. So we unfold the expression and we obtain $\dot{b}_{0,j} = a_{0,1} * a_{1,n} * b_{n,j}$; because $\mathbf{A}$ is matrix closure, we have $a_{0,1} * a_{1,n} = a_{0,n}$, thus, in the first evaluation we had $b_{0,j} = a_{0,m} * b_{m,j} \leq a_{0,n} * b_{n,1} = \dot{b}_{0,j}$. The following computation of $b_{1,j}$ does not affect the computation of $b_{0,j}$, therefore we may perform the computation in any order and in-place.

Notice that this observation assures that, as long as all terms are computed and written into the destination matrix without *write races*, the matrix multiplication can be successfully parallelized as it would be when source and destination operands are all non-overlapping matrices.

Intuitively, we can see that our algorithm inherits the properties of MM, and thus, it is **cache oblivious** achieving optimal data cache utilization at every cache level (asymptotically). We briefly repeat here the major results about the locality property of Kleene's algorithms as previously investigated in [64]. In fact, Kleene's algorithm is the first cache-aware algorithm with access complexity $O(\frac{n^3}{\sqrt{S}})$ (also I/O complexity), where S is the cache size in number of elements. Matrix Multiply has the same lower and upper bound [11].

To prove that R-Kleene is asymptotically optimal, we determine the upper bound to the

access complexity as follows. Suppose that $S = s^2$, matrix J has size $n \times n$ and (n **mod** s) $= 0$. (The case for general square matrices is similar.) Recursively, R-Kleene divides the problem $n \times n$ in smaller problems and it may compute the solution directly when the problem has size no larger than $s \times s$. Thus, the algorithm solves $8^{\log n/s} = (\frac{n}{s})^3$ problems directly, each requiring up to $2s^2$ memory accesses (i.e., s^2 reads from memory to cache and s^2 writes from cache to memory), so we achieve a total of $O(\frac{n^3}{\sqrt{S}})$ memory accesses, which is optimal asymptotically. Notice that because the algorithm accesses tiles of the adjacency matrix, a cache-aware layout can store such tiles continuously in memory improving the cache behavior of the algorithm. Such a layout reduces self/inter interference, therefore, cache conflicts further (see also [58, 24, 52]).

Previously [22, 8], we developed techniques to improve the **leaf computation** of MM (where the recursion stops). In fact, we may exploit data reuse in registers and, therefore, we may achieve a sensible reduction of loads/stores for matrix multiplication. Indeed, we may reduce the memory accesses, for matrix multiply of matrices of size $m \times m$, from $3m^3 + o(m^2)$ to $\frac{2}{r}m^3 + o(m^2)$, with $1 \leq r^2 \leq R$ where R is the number of registers available. The fewer memory loads/stores in MM are, the higher the overall performance is, because MM is basic kernel.

4.4.2 Experimental Results

In this section, we discuss briefly the characteristics of the algorithms implemented, and how we measured overall performance. We conclude this section presenting the experimental results of the algorithms for every system separately. In practice, we compare R-Kleene versus three other algorithms on five systems using (very different) processors for dense adjacent matrices with random entries uniformly generated in an interval. In all tests and for all architectures here presented, we checked the correctness of the matrix closure by direct comparison with the one generated by FW.

R-Kleene is our recursive algorithm in Figure 4.20.(b). The basic MM operation is a recursive algorithm that exploits data locality and aggressive data reuse in registers at the leaf computations so as to achieve near optimal performance. The algorithm assumes that the adjacent matrix is stored in a row-major format.

Floyd-Warshall (**FW**) is the classic algorithm based on a single loop nest. The algorithm assumes that the adjacent matrix is stored in a row-major format. In general, this algorithm is efficient only for small problem sizes and its performance degrades quickly as the problem size increases.

Simple Recursive (**Z-SR**) is the recursive algorithm in Figure 4.20.(c). This is the algorithm presented by Park et al. [19]– which these authors proposed for power of two matrices only – and, in fact, the performance presented in this work coincides with the performance previously published. The algorithm assumes that the adjacent matrix is stored in a generalized Z-Morton format [14]; that is, logical submatrices of the adjacent matrix are stored continuously in memory in a recursive fashion.

ZR-Kleene is the R-Kleene algorithm, however it assumes that the adjacent matrix is stored in a generalized Z-Morton format [14]. This algorithm should have a performance advantage for very large problems and memory hierarchies with high latency and low associativity.

Our goal is to show the performance improvements obtained by register management only (R-Kleene), by memory layout optimization only (Z-SR) and by the synergy of both (ZR-Kleene). We measure performance as **millions of integer instructions per second** (MIPS) determined as $n^3/(\textit{execution of the algorithm in seconds})$. This format is consistent with the one used for classical linear algebra applications (e.g., MM). However, there is a major difference between APSP and MM. In MM, highly pipelined functional units execute the basic operation (*madd*) using a separate register file and the performance measure is MFLOPS. In contrast, in APSP algorithms, the basic operation is based on a conditional

branch, that is, a **comparison-addition** (*compadd*). This constraint affects the final performance as a function of the values in the adjacent matrix, even if only by a constant factor. [1] In practice, some processors have available branch prediction units - e.g., R12K - allowing instruction speculation on either branch of a conditional jump. However, because the unpredictable nature of the adjacent matrix, branch predictor may be ineffective. Nevertheless, we assume that the reference peak performance of any algorithm is the number of cycles per second.

In the following we discuss our results. For the Fujitsu HAL 100 system based on a SPARC64 processor (one level of split caches, 128KB 4-way data and instruction caches), we present the performance results in Figure 4.21. The Z-SR algorithm performs 1 *compadd* every 5 cycles while R-Kleene and ZR-Kleene perform 1 *compadd* every 4 cycles; that is a 20% performance improvement. Notice that the memory layout has no significative effects on the overall performance of the algorithms. (We used the native compiler, *hcc*, to generate the executables.)

For the SGI O2 system based on a MIPS R12K 300MHz processor (with two-level cache: first level, 32KB 2-way distinct data and instruction cache; second level, 512K 2-way unified cache), we can achieve the best relative performance as presented in Figure 4.22. The Z-SR algorithm performs 1 *compadd* every 5 cycles, while R-Kleene and ZR-Kleene perform 1 *compadd* every 2 and 3 cycles. R-Kleene and ZR-Kleene achieve a two-fold speed up with respect to Z-SR. Notice that the memory layout has significative effects on the overall performance of the algorithms. (We used the native compiler, SGI compiler, to generate the executables.)

For the Sun microsystems Sun Blade system based on an UltraSparc IIe 500MHz processor (two-level cache: first level, 16KB direct mapped distinct data and instruction cache; second level, 512KB 2-way unified cache), we present the performance results in Figure 4.23.

[1] But not as much as for algorithms on sparse adjacent matrices

The Z-SR algorithm performs 1 *compadd* every 11 cycles. In contrast, R-Kleene and ZR-Kleene perform 1 *compadd* every 7 cycles, which is a 30% performance improvement. Notice that the memory layout has no significative effects on the overall performance of the algorithms. (We used two compilers to generate the executables, *gcc/3.0.4* and *cc-forte-6*, we present the best performance.)

For the FOSA 3240 system based on a Pentium III 800MHz (two-level cache: first level, 16KB direct mapped distinct data and instruction cache; second level, 256KB direct mapped unified cache), we present the performance results in Figure 4.24. The Z-SR algorithm performs 1 *compadd* every 9 cycles. In contrast, R-Kleene and ZR-Kleene perform 1 *compadd* every 6 cycles, which is a 35% performance improvement. Notice that the memory layout has significative effects on the overall performance of the algorithms. (We used *gcc/3.1* compiler to generate the executables.)

For ASUS system based on an Athlon-XP 2800 (two-level cache: first level, 64KB 2-way distinct data and instruction cache; second level, 256KB 16-way unified cache), we present the performance results in Figure 4.25. The Z-SR algorithm performs 1 *compadd* every 13 cycles. In contrast, R-Kleene and ZR-Kleene perform 1 *compadd* every 6 cycles. Notice that the memory layout has no significative effects on the overall performance of the algorithms. (We used *gcc/3.3* compiler to generate the executables.)

4.4.3 APSP Conclusions and Future Work

We presented R-Kleene: a novel D&C algorithm for the solution of APSP; we also presented a quantitative measure for its performance across five systems. We conclude that an efficient register allocation is an important feature of any APSP algorithms; we also notice that non-standard layouts are beneficial for very low associative caches but otherwise row-major layouts are quite adequate.

We have started the design and implementation of a preliminary parallel R-Kleene al-

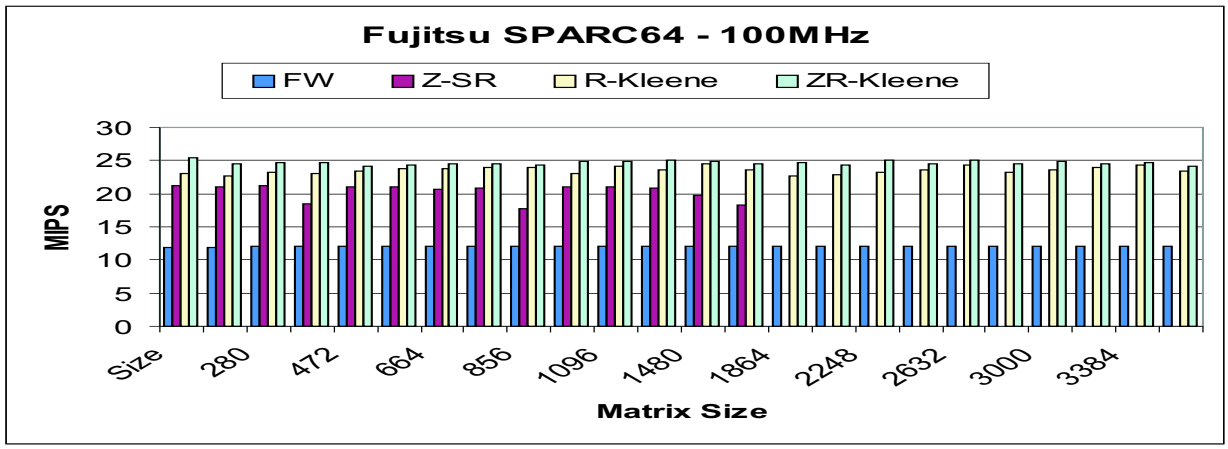

Figure 4.21: Fujitsu HAL 100: best performance 4 cycles per *compadd*.

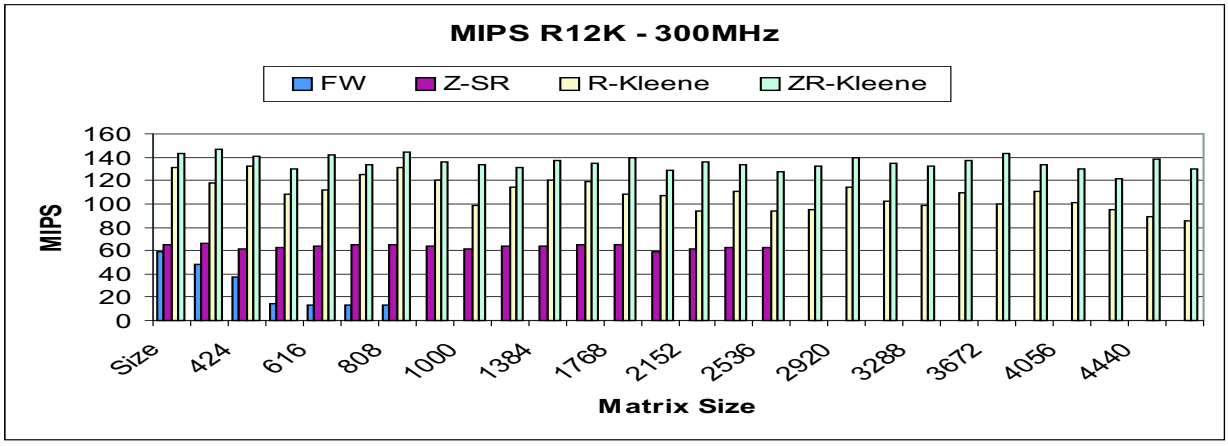

Figure 4.22: SGI O2: best performance 2 cycles per *compadd*.

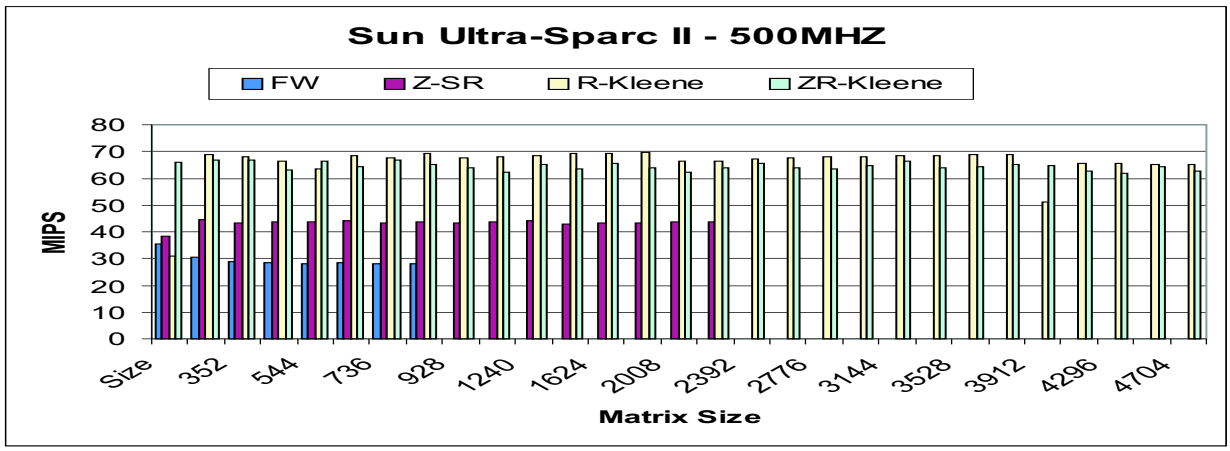

Figure 4.23: Sun Blade 100: best performance 7 cycles per *compadd*.

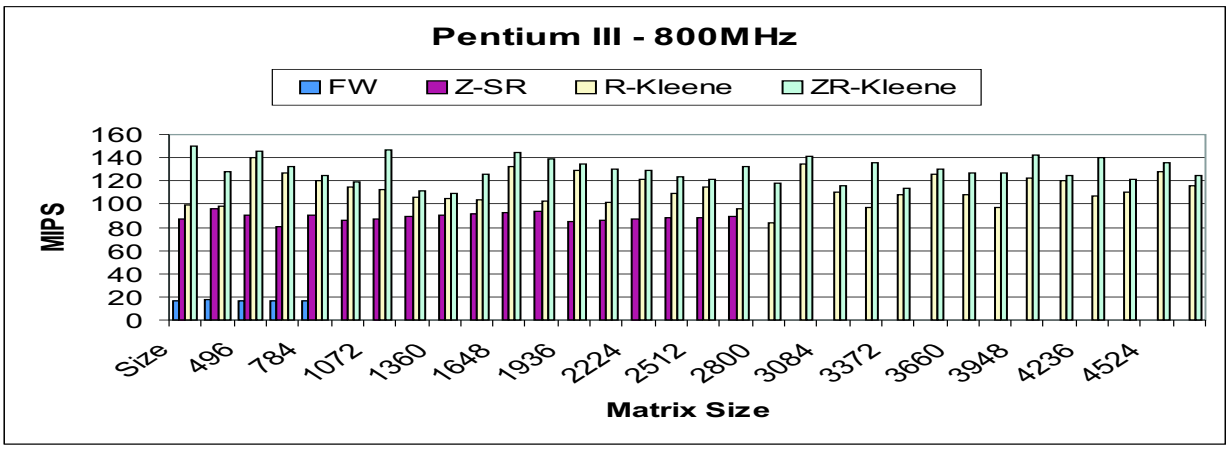

Figure 4.24: FOSA 3240: best performance 6 cycles per *compadd*.

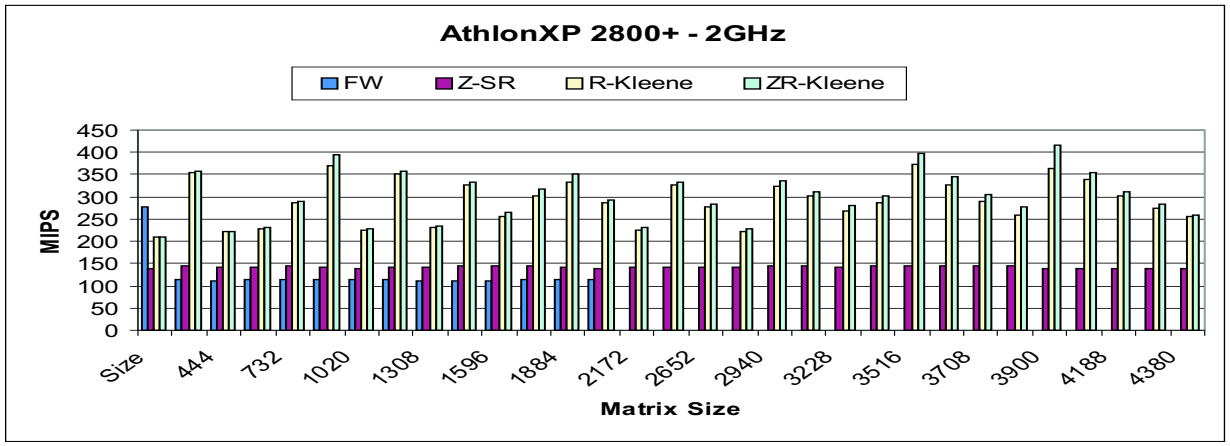

Figure 4.25: ASUS: best performance 5 cycles per *compadd*.

gorithm for a two-processor system achieving speed-ups ranging from 1.41 to 1.74. In the future, we intend to import ATLAS (also multi-thread) routines to improve performance further and exploit parallelism at processor and thread level.

4.5 Conclusive Remarks

We considered four applications where we apply our approach based on the recursion-DAG manually.

We presented an evaluation for cache and runtime performance for LU-factorization with partial pivoting, and therefore for the solution of triangular systems. By simulation, we show the good cache behavior of Toledo's algorithm on several memory hierarchies and we show running time against other LU-factorization algorithms. We have achieved the following results: the new framework and optimizations are suitable for LU-factorization (in general, we have obtained the fastest factorization); the cache behavior is extremely good and predictable; high performance is achieved for different architectures. Our LU-factorization does not perform well for PentiumII because of our poor register allocation for stack-register file.

We have developed a careful study of matrix multiplication implementations, showing that suitable algorithms can efficiently exploit the cache hierarchy without taking cache parameters into account, thus ensuring portability of cache performance. Clearly, performance itself does depend on cache parameters and we have provided a reasonable estimator for it. We have also experimentally shown that, with a careful implementation of recursion, high performance is achievable.

We applied the register allocation developed for matrix multiplication on the all-pair shortest path algorithms. We have collected experimental results for six recursive algorithms across five different architectures. We have shown that the number of parameters used by a recursive algorithm do not affect performance significantly. We have shown that the choice

of a recursive layout offers high performance. At the same time, we have shown that for some architectures, the complexity of the layout does not offer higher performance than the common row major format. We have found that the myth of ASP being much slower than matrix multiplication is not justified and we present experimental results. The performance is comparable to matrix multiply and very close to peak performance.

Eventually, we present implementation for our Fast Fourier Transform using a recursion-DAG. We present just a proof of concept in this chapter, but we present in Section 6.3.4 a performance evaluation where we may find a comparison with FFTW.

CHAPTER 5

STAMINA: Static Modeling of Interference And Reuse

Caches are crucial components of modern processors; they allow high-performance processors to access data fast and, due to their small sizes, they enable low-power processors to save energy - circumventing memory accesses. A high-performance compiler must address the problem of caches utilization and it must drive code-architecture adaptation, reducing memory accesses and energy per access.

We examine efficient utilization of data caches in an adaptive memory hierarchy. We exploit data reuse through the static analysis of cache-line size adaptivity. We present a framework that enables the quantification of data misses with respect to cache-line size at compile-time using (parametric) equations modeling interference.

Our approach aims at the analysis of perfect loop nests in scientific applications; it is an extension and generalization of the Cache Miss Equation (CME) proposed by Ghosh et al 1999 [79], and it is applied to direct mapped cache.

We show evidence of both expressiveness and practicability of the analysis. Part of this analysis is implemented in a software package STAMINA and we present analytical results in comparison with simulation-based methods.

5.1 Notation and Interference Density

In this section, we introduce the notation and terminology used.

A **perfect loop nest** composed of d loops determines a set of integral points in $\mathbb{N}^d$. Each point is denoted by a column vector: $\mathbf{i} = (i_0, \ldots, i_{d-1})^t$; the first component (i.e., i_0) is associated with the outermost loop and the last component (i.e., i_{d-1}) is associated with the innermost loop. The loop order specifies a **lexicographic order** (as in [79]). In fact, a point $\mathbf{u}$ **precedes** a point $\mathbf{v}$, denoted by $\mathbf{u} \lhd \mathbf{v}$, if there exists an index t, $0 \leq t \leq d-1$, such that $u_n = v_n$ for every $n < t$ and $u_t < v_t$. When $\mathbf{v} = \mathbf{u}$ or $\mathbf{v} \lhd \mathbf{u}$, we use the notation $\mathbf{u} \trianglelefteq \mathbf{v}$. A **partial order** [1] between two points $\mathbf{v}$ and $\mathbf{u}$ is defined as follows: a point $\mathbf{u}$ is **smaller than** a point $\mathbf{v}$, denoted by $\mathbf{u} < \mathbf{v}$, when $v_n \leq u_n$ for every index n, $0 \leq n \leq d-1$, except at least one index k such that $u_k < v_k$. For example, a point $\mathbf{u}$ determines a unique bounded polyhedron: $P = \{\mathbf{v} | \mathbf{0} \leq \mathbf{v} \leq \mathbf{u}\}$. Note that, if $\mathbf{v} < \mathbf{u}$ then $\mathbf{v} \lhd \mathbf{u}$, but not vice versa. (For example, $(1,1) < (2,2)$ and $(1,1) \lhd (2,2)$, and $(1,2) \lhd (2,1)$ but $(1,2) < (2,1)$ is not defined!)

We define an **iteration space** as a bounded polyhedron: $S_{\mathbf{p}} = \{\mathbf{i} | \mathbf{0} \trianglelefteq \mathbf{i} \trianglelefteq \mathbf{n}\}$, where $\mathbf{n}$ is $\mathbf{Ai} + \mathbf{Bk} + \mathbf{Cp}$ with $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are constant matrices of size $d \times d$, $\mathbf{k}$ is a vector of constants and $\mathbf{p}$ is a vector of parameters. The parameter $\mathbf{p}$ does not affect the shape of the iteration space but only its cardinality. For example, consider $S_{\mathbf{p}}$ defined as $\{\mathbf{i} | 0 \trianglelefteq \mathbf{i} \lhd (p,p)^t, \}$. The iteration space $S_{\mathbf{p}}$ has **cardinality** $|S_{\mathbf{p}}| = p^2$, which is a function of p, and it has a square shape in $\mathbb{N}^2$ independently of any value of $\mathbf{p}$.

An **interval** is a set $P^{\mathbf{r}}(\mathbf{s}) = \{\mathbf{v} \in S_{\mathbf{p}} | \mathbf{s} - \mathbf{r} \lhd \mathbf{v} \trianglelefteq \mathbf{s}\}$, where $\mathbf{s}, \mathbf{s} - \mathbf{r} \in S_{\mathbf{p}}$ and $\mathbf{0} \trianglelefteq \mathbf{r}$. The cardinality of an interval is a function of $\mathbf{s}$. The cardinality of an interval represents the number of iterations separating the iteration point $\mathbf{s} - \mathbf{r}$ and the iteration point $\mathbf{s}$. In short, we specify **distance** as the cardinality of an interval, $|P^{\mathbf{r}}(\mathbf{s})|$. When $\mathbf{r} = \mathbf{e}_{d-1} \equiv (0, \ldots, 0, 1)$,

[1] Also known as geometrical order, it does not always define an order between two iteration points.

[2] we have $|P^{\mathbf{r}}(\mathbf{s})| \leq 1$. An interval, as well as an iteration space, is the composition of disjoint elementary rectilinear polyhedra. Note that these polyhedra can be just one point, where the vertices merge into one. This property assures that the determination of the distance of any interval is computable and that an Ehrhart polynomial exists [105, 106, 107, 108].

A reference R in the body of a loop nest has **temporal reuse** if, in different iterations $\mathbf{u}$ and $\mathbf{v}$, the reference accesses the same memory location $A_d[R(\mathbf{u})] = A_d[R(\mathbf{v})]$ [55]. We represent reuse by a vector $\mathbf{r}$ such that we have $A_d[R(\mathbf{u})] = A_d[R(\mathbf{u}+\mathbf{r})]$ for every iteration point $\mathbf{u}$. When the address of a reference is an affine function, that is, $A_d[R(\mathbf{u})] = \mathbf{l}^t(\mathbf{M}\mathbf{u}+\mathbf{b})$, the reuse vector is a point in the **null space** of matrix $\mathbf{M}$ - i.e., $\mathbf{M}\mathbf{r} = \mathbf{0}$.[3] A reference has **spatial reuse**, if the reference accesses –in different iterations– the same cache line. Note that temporal reuse is a particular case of spatial reuse. We have **group temporal** and **group spatial reuse** when different references exploit temporal and spatial locality among each other –during the computation.

For example, consider a matrix $A[100][100]$ stored in row-major format and starting at address 0x0. Consider a reference $R_A = A[u_0 + u_1][u_1]$ in a loop nest composed of 2 loops. We have $A_d[R_A(\mathbf{u})] = (100, 1)^t(\begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix}\mathbf{u} + (0, 0)^t)$. In practice, R_A has spatial reuse and reuse vector $(0, 1)^t$ but it has no temporal reuse because $\begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix}\mathbf{u} = \mathbf{0}$ only when $\mathbf{u} = \mathbf{0}$ ($\mathbf{null}(A)$=$\emptyset$).

While we carry on the computation of the loop nest and one reuse of a reference is accomplished, we have a hit in cache because the same reference is reused successfully. Otherwise, the memory reference may have been evicted from the cache and a miss may happen. The reuse $\mathbf{r}$ of a reference $R_A(\mathbf{u})$ is **prevented**, when either a reference $R_B(\mathbf{s})$ with $\mathbf{s} \in P^r(\mathbf{u})$ interferes with the reference $R_A(\mathbf{u})$, or the iteration $\mathbf{u} - \mathbf{r}$ does not belong to the space.[4]

[2] The vector $\mathbf{e}_i$ the i-th column vector of the identity matrix $I \in \mathbb{N}^d$.
[3] Note that linear parameters do not affect the null space.
[4] Spatial reuse is prevented when a different line is accessed.

In general, the prevention of a reuse by another memory reference does not mean that we have a miss in cache. A reference may have multiple reuse vectors and to have a miss in cache, all reuses must be prevented. We model the prevention of a reuse by an interference equation as follows.

Given two array references R_A –*interferer*– and R_B –*interferee*, we define an **interference equation** as :

$$E_{\mathbf{r}} \equiv \begin{cases} \mathbf{a}^t\mathbf{u} + a_{-1} = \mathbf{b}^t\mathbf{s} + b_{-1} + nC + q + \mathbf{d}^t\mathbf{p} \\ \\ \text{with } \mathbf{u} \in P^{\mathbf{r}}(\mathbf{s}), \mathbf{s} \in S_{\mathbf{p}}, n \neq 0, |q| < L, \\ \text{and with } L \text{ cache-line size} \end{cases} \tag{5.1}$$

where $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{d}$ are constant vectors; the affine function for R_A is $\mathbf{a}^t\mathbf{i} + a_{-1}$ and the affine function for R_B is $\mathbf{b}^t\mathbf{i} + b_{-1}$; the parameter vector is $\mathbf{p}$ and the reuse vector for R_B is $\mathbf{r}$; the cache size is a constant C; the free variable n is not zero; the offset in the cache is $|q| \leq L-1$; the cache-line size is L. The set of constraints is defined as **definition domain**.

```
extern double A[2000][1024],B[100][1024];

void foo(int m, int start) {
  int i,j;
  for (i=0;i<m;i++)                /* 0<=m<100 */
    for (j=0;j<m;j++)
      A[i][j+start] += B[i][j];  /* 0<= start <1024-100 */
}
void update(int start) {
  int start1=0; /* compile time */
  int start2;
  int startin;

  start2 = start+2; /* not really at compile time */
  foo(50,start1);
  foo(50,start2);

  scanf(''\%d'',\&startin); /* run time */
  foo(50,startin);
}
```

Figure 5.1: Parameterized loop bounds and index computation, thus interference.

An interference equation is always represented by an equality constraint –Diophantine

equation– and by a definition domain, in which the unknowns are defined [79, 57]. For example, in Fig. 5.1 – introduced in Fig. 2.2 on page 37, the interference equation for interferer A and interferee B is as follows:

$$E_1 \equiv \begin{cases} b_{-1} + (1024, 8)(s_0, s_1)^t + nC + q + (0, 8)(m, start)^t \\ = a_{-1} + (1024, 8)(u_0, u_1)^t \\ \\ \text{with } \mathbf{u} = \mathbf{s}, \mathbf{s} \in S_{\mathbf{p}}, n \neq 0, |q| < L, \\ \text{and with } L \text{ cache-line size} \end{cases} \tag{5.2}$$

We model a direct mapped cache, so when the interference equation has a solution, we have cache interference and, thus, we have cache misses. Otherwise, if the equation has no solution and the interferee has only one reuse vector, then we have a hit.

When $|P^{\mathbf{r}}(\mathbf{s})| = 1$, we simplify (5.1). When $\mathbf{r} = \mathbf{e}_{d-1} \equiv (0, \ldots, 1)$, $\mathbf{u} = \mathbf{s} - \mathbf{e}_{d-1}$ ($\mathbf{u} = \mathbf{s}$ when $\mathbf{r} = \mathbf{0}$) we isolate the term $nC + q$ as follows:[5]

$$E_{\mathbf{e}_{d-1}} \equiv \begin{cases} c_{-1} + \mathbf{c}^t\mathbf{s} = nC + q \\ \\ \text{with } \mathbf{c} = \mathbf{a} - \mathbf{b}, \mathbf{s} \in S_{\mathbf{p}} \\ \text{and } c_{-1} = a_{-1} + a_{d-1} - b_{-1} - \mathbf{d}^t\mathbf{p} \end{cases} \tag{5.3}$$

For example, we simplify (5.2) as follows

$$E_1 \equiv \begin{cases} -168384000 - 8start = nC + q \end{cases} \tag{5.4}$$

We define the **interference density**, denoted by ρ_E, as the ratio of the number of points in the iteration space, for which the equation E has solution, over the total number of iteration points. For example, in (5.4) ρ_{E_1} is 1 (if $8start < L$).

Property 1 *If in (5.3), $E_{\mathbf{e}_{d-1}}$, a solution exists and $\mathbf{c} = C\mathbf{m}$, then $\rho_{E_{\mathbf{e}_{d-1}}} = 1$.*

Proof: Because solution exists in (5.3), a point $\mathbf{v}$, an integer n_0 and an integer q_0 exist for which $c_{-1} + \mathbf{c}^t\mathbf{v} = n_0C + q_0$. We substitute $\mathbf{c}$ with $C\mathbf{m}$ to obtain $c_{-1} + C\mathbf{m}^t\mathbf{v} = n_0C + q_0$.

[5]Note that we do not repeat the definition of the domains for the unknowns n and q.

For any point $\mathbf{s} \in S_{\mathbf{p}}$, we find an integer g such that $c_{-1} + C\mathbf{m}^t\mathbf{s} = gC + q_0$ (e.g., $g = n_0 + \mathbf{m}^t(\mathbf{v} - \mathbf{s})$). Therefore, a solution exists and $\rho_{E_{\mathbf{e}_{d-1}}} = 1$. □

We can simplify (5.3) further, because the element c_k, which is a multiple of the cache size (i.e., c_k **mod** $C = 0$), does not contribute to the interference density:

$$E_{mod} \equiv \begin{cases} f_{-1} + \mathbf{f}^t\mathbf{s} = nC + q \\ \\ \text{where } f_k = c_k \textbf{ mod } C, \forall k \in [-1, d-1] \\ \text{and } \mathbf{s} \in S_{\mathbf{p}} \end{cases} \tag{5.5}$$

In practice, $\rho_{E_{\mathbf{e}_{d-1}}}$ for (5.3) is equal to $\rho_{E_{mod}}$ for (5.5).

Property 2 *For the general case in (5.1), we have* $\rho_{E_{\mathbf{r}}} < \min(1, \max_{\mathbf{s}\in S_{\mathbf{p}},\mathbf{p}} \rho_{E_{mod}} * |P^{\mathbf{r}}(\mathbf{s})|)$.

Proof: For every $\mathbf{s} \in S_{\mathbf{p}}$, we break the interval $P^{\mathbf{r}}(\mathbf{s})$ in smaller intervals with unit distance. We have up to $\max_{\mathbf{s}\in S_{\mathbf{p}}} |P^{\mathbf{r}}(\mathbf{s})|$ unit intervals. We consider each interval independently and we determine its interference density. Every interval has interference density, $\max_{\mathbf{s},\mathbf{p}} \rho_{E_{mod}}$. □

Property 2 states that we can determine the interference density for a rather complex interval using an estimate based on unit intervals. We shall present in Section 5.2.3 a technique that estimates $\rho_{E_{mod}}$ and it is independent of any parameter $\mathbf{p}$ and any iteration point in the iteration space $\mathbf{s}$. Furthermore, when the reuse vectors are short, the reuse intervals have short distance and therefore we have a simple and tight estimation. McKinley and Temam present strong evidence that short reuse are common in scientific computations [83]. We assume that the target of our analysis are applications with short reuse vectors –mostly spatial reuse.

5.2 Parameterized Loop Analysis

In this section we introduce our approach in a top-down fashion describing the organization of our software package STAMINA, Fig. 5.2. In Section 5.2.1, we introduce the trade-off

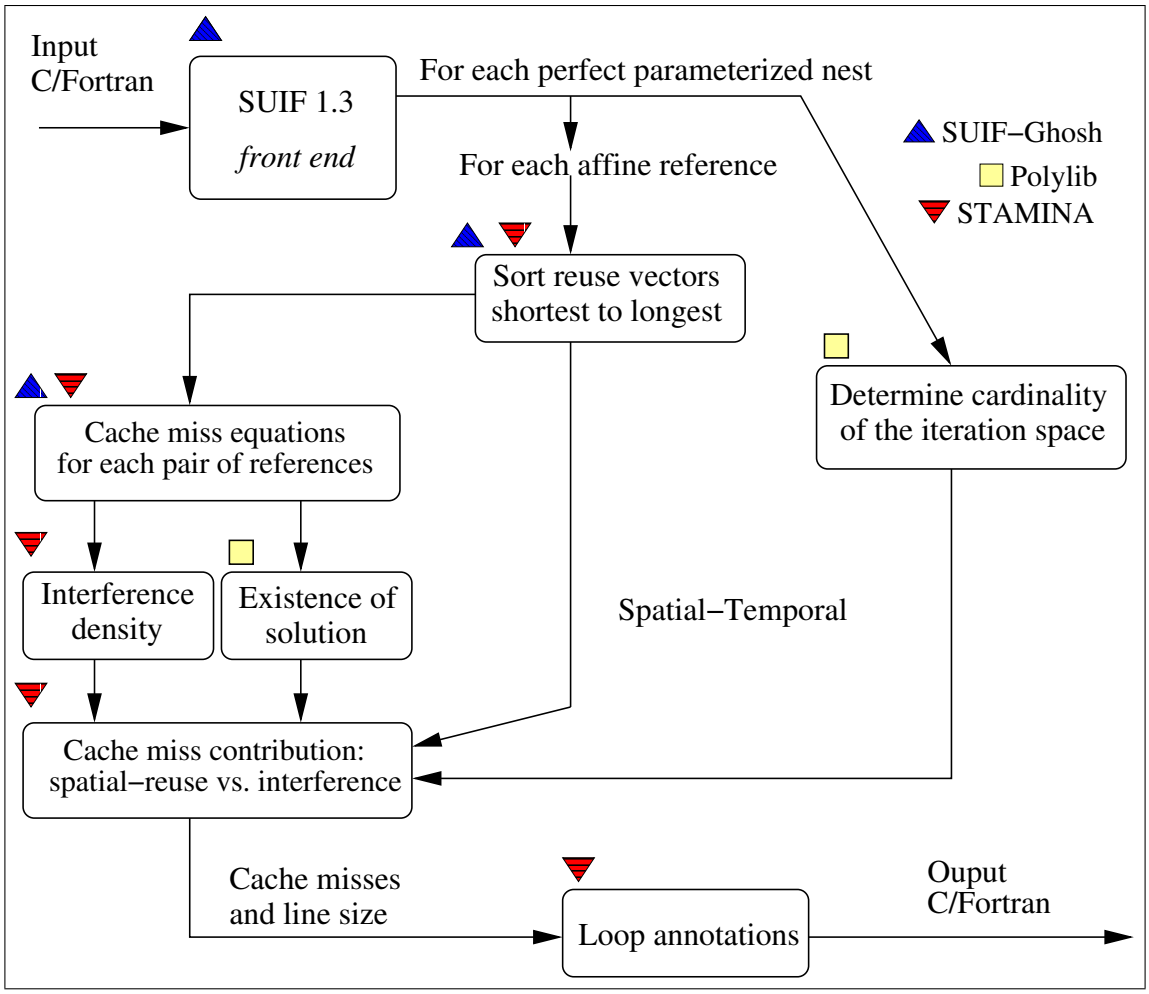

Figure 5.2: STAMINA

between spatial reuse and cache interference, and we propose our model for the representation of cache misses as a function of both cache-line size and interference density. In Section 5.2.2, we present how we model interference as set of interference equations. In Section 5.2.3, we discuss the computation of the interference density based on a simplified analysis of the interference equations. In Section 5.2.3, we introduce a more accurate analysis of the interference density based on the theory of affine equations using unimodular transformations [57].

5.2.1 Spatial Reuse vs. Interference: Optimal Cache-Line Size

Ideally, without cache interference, an application having spatial locality is able to exploit a large cache-line size by reducing cache misses, by virtue of fewer memory accesses. However, a large cache-line size may increase interference, which may impede the spatial locality exploitation and, in the worst case, it may increase cache misses. For some applications, we find it acceptable to have an increase of cache misses due to interference as long as the overall performance improves, due to fewer communications to and from the cache.

For one memory reference R, we estimate the total contribution to the interference density, $\eta_R(L)$, by distinguishing three different cases and considering two contributions.

$$\eta_R(L) = \begin{cases} \min(1, \frac{s}{\ell} + \mu_R(L)) & \text{if } \mu_R(L) < 1 \text{ and} \\ & \text{spatial\&temporal reuse,} \\ \min(1, \mu_R(L)) & \text{if temporal reuse only,} \\ 0 & \text{otherwise.} \end{cases} \tag{5.6}$$

We define $\eta_R(L)$ in (5.6) as the **spatial-temporal interference density per memory reference**.

The memory reference can have **spatial** and **temporal** reuse; that is, a reference has reuse of the same cache line and reuse of the same element located in a cache line. If a reference R has spatial reuse and there is no interference, we estimate a miss every $\frac{\ell}{s}$ access(es) –i.e., iteration(s). The interference density is $\frac{s}{\ell}$, where ℓ is the line size in data elements (i.e., $\ell = L/8$ when an element is a double) and s is the length of the spatial reuse in elements. This contribution to the interference density is due to the spatial reuse only, and, notice that, it is a monotonically decreasing function in L. Spatial reuse is an artificial reuse, which is introduced by the memory architecture configuration. A spatial reuse may be prevented because of the access of a different cache line and not because of interference. In fact, any other (longer) reuse may be satisfied and, instead of a cache miss, we could achieve a cache hit. For spatial reuse, it would be convenient to consider the effect of longer

(temporal) reuse as well (see Ghosh et al. [109]).

If there is interference, part of the reuse can be prevented and we can have a larger contribution to the interference density. The factor $\mu_R(L) \in [0, 1]$ is the estimate of interference density due to cache interference only, **interference density per memory reference**. That is, how other references displace reference R from the cache. When $\mu_R(L) = 1$, it means that interference is so high that no reuse is possible. The factor $\mu_R(L)$ is a monotonically increasing function. We shall see how to determine $\mu_R(L)$ in Section 5.2.2.

If a reference R does not have reuse of any kind, then $\eta_R(L) = 0$. If there is no reuse, there is no interference. If there is a **cold miss**, [6] it is unavoidable in this framework for every cache-line size.

Finally, the estimate of the number of cache misses, due to one memory reference, is simply $|S_{\mathbf{p}}|\eta_R(L)$ (i.e., $|S_{\mathbf{p}}|$ is the number of iterations in the loop nest). We explain shortly, how we use $\eta_R(L)$ to estimate the number of cache misses as a function of the cache-line size. Suppose we have z memory references in a loop nest. We sort the references and we label them by using a unique integer according to the following criterion. Reference R_i, with $0 \leq i < x$, has spatial reuse and reference R_j, with $x \leq j < z$, has temporal reuse. An upper bound on the number of cache misses is given in (5.7).

$$\begin{aligned} |Misses| &\leq |S_{\mathbf{p}}|\epsilon(L) \\ |S_{\mathbf{p}}|\epsilon(L) &= |S_{\mathbf{p}}| \sum_{i=0}^{z-1} \eta_{R_i}(L) \\ &= |S_{\mathbf{p}}|(\sum_{i=0}^{n-1} \eta_{R_i}(L) + \sum_{i=n}^{m-1} \mu_{R_i}(L)). \end{aligned} \tag{5.7}$$

Because the function $|S_{\mathbf{p}}|$ is independent of the cache-line size, the minimum number of cache misses is a function of $\epsilon(L) = \sum_{i=0}^{n-1} \eta_{R_i}(L)$. In practice, we seek for the optimal cache-line size that minimizes $\epsilon(L)$ and we do it by a linear search for increasing values of L (i.e.,

[6] The first time a reference is read, we have a cache miss and it is defined as cold miss

$L = 8, 16, 32, 64, 128, 256$ bytes).

5.2.2 Interference Density per Memory Reference

In this section we introduce two important concepts and estimates: the interference existence and the interference density per memory reference (i.e., $\chi_E(L)$ and $\mu(L)$, respectively).

Consider an interference equation E_{mod} –as in (5.5). We define **interference existence** as a 0-1 function expressing whether or not the equation E_{mod} has integer solutions:

$$\chi_{E_{mod}}(L) = \begin{cases} 1 & \text{if } E_{mod} \text{ has a solution} \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

where L is the cache-line size.

A CME solver as it counts the number of integer solutions of an interference equation, it resolves the existence problem as well. However, a solver may be designed for the existence problem only; in fact, Omega test is an example of such a solver [110, 111]. Note that in the worst case scenario, searching for one solution is as hard as counting all integer solutions.

Currently, we deploy Polylib, which applies a linear search in parameterized polyhedra to find whether or not an integer solution exists. In Section 5.3.3 and 5.3.2, we present an example showing the way we use the interference existence to achieve an accurate estimate of the number of cache misses.

In the following, we present our approach for the determination of the interference density per memory reference. We outline the approach describing the following three possible scenarios:

1. Consider a memory reference R_A with one reuse vector $\mathbf{r}$ and with k interferers R_{B_i}, $0 \leq i < k$. For each pair of references R_A and R_{B_i}, we determine the interference equation E_i, we estimate the interference density and we compute the interference existence (i.e., ρ_{E_i} and $\chi_{E_i}(L)$). Then, we determine the contribution of each interferer

R_{B_i} independently and we add their contributions:

$$\mu(L) = \sum_{i=0}^{k} \rho_{E_i} \chi_{E_i}(L) \tag{5.9}$$

2. Consider a memory reference R_A and one interferer R_B. The reference R_A has m reuse vectors $\{\mathbf{r}_i\}_{0,m-1}$ such that $\mathbf{r}_{m-1} \lhd \ldots \lhd \mathbf{r}_0$. Every reuse vector $\mathbf{r}_i$ is associated with an interval $P^{\mathbf{r}_i}(\mathbf{s})$ and for every $i > j$ we have $P^{\mathbf{r}_i}(\mathbf{s}) \subset P^{\mathbf{r}_j}(\mathbf{s})$. In particular, we have that $\cap_{i=0}^{m} P^{\mathbf{r}_i}(\mathbf{s}) = P^{\mathbf{r}_{m-1}}(\mathbf{s})$.

   We consider the shortest reuse vector only (i.e., $\mathbf{r}_{m-1}$) and, therefore, we consider the shortest interval only (e.g., $P^{\mathbf{r}_{m-1}}(\mathbf{s})$). Because, if the shortest reuse is prevented, all reuses are prevented and there is a cache miss; otherwise, the shortest reuse is exploited and there is no miss in cache –however, other reuse may be prevented. This is equivalent to the first case with one interferer: $\mu(L)$ is $\rho_E \chi_E(L)$ (e.g., in (5.9) with $k = 1$).

3. Consider a reference R_A with k interferers R_{B_i} and m reuse vectors $\{\mathbf{r}_i\}_{0,m-1}$ such that $\mathbf{r}_{m-1} \lhd \ldots \lhd \mathbf{r}_0$. Every reuse vector $\mathbf{r}_i$ is associated with an interval $P^{r_i}(\mathbf{s})$ and for every $i > j$ we have $P^{\mathbf{r}_i}(\mathbf{s}) \subset P^{\mathbf{r}_j}(\mathbf{s})$. In particular, we have that $\cap_{i=0}^{m} P^{\mathbf{r}_i}(\mathbf{s}) = P^{\mathbf{r}_{m-1}}(\mathbf{s})$.

   For each pair of references R_A and R_{B_i} we determine the interference equation E_i for the shortest reuse only, therefore for the shortest interval (e.g., $P^{\mathbf{r}_{m-1}}(\mathbf{s})$). Because, if the shortest reuse is prevented, all reuses are prevented and there is a cache miss; otherwise, the shortest reuse is exploited and there is no miss in cache. In fact, in (5.9) we model this case as well.

The number of cache misses for a direct mapped cache is up to $|S_{\mathbf{p}}|\mu(L)$. For a k-way associative cache, we may estimate the number of cache misses as $|S_{\mathbf{p}}| \left\lfloor \frac{\mu(L)}{k} \right\rfloor$. In practice, our estimate/approach is independent of the approach proposed by Chatterjee et al. [112]

for associative caches. However, there are three common features we summarize in the following. First, both approaches model cache misses using polyhedra, thus they do not convey any information on the temporal distribution of the cache interference; second, both are approximations – not upper bounds; third, we may use these estimations to determine statically –i.e., at compile time– the minimum associativity that circumvent cache interference altogether.

The interference equations model cache interference in an interval. This interval must be a valid interval in the iteration space. Otherwise, no analysis is performed. For example, given a reuse vector $\mathbf{r}$, our approach does not analyze the set of iterations:

$$P_B^{\mathbf{r}} = \{\mathbf{j} | \mathbf{j} \in S_{\mathbf{p}} \cap (\mathbf{j} - \mathbf{r}) \notin S_{\mathbf{p}}\}. \tag{5.10}$$

We can rewrite (5.10) as the union of non-intersecting elementary rectangular sets, therefore we may use Polylib to compute its cardinality. If we count the number of iterations in this set, we determine a **confidence index**, which is used separately to assess whether the analysis has any contribution. In fact, the smaller the reuse vector is, the larger is the iteration space investigated by our approach, therefore the larger the number of cache misses we can determine through the interference density. For the examples we present in Section 5.3, we analyze 99.9% of the iterations.

5.2.3 Interference Density Analysis, STAMINA

In this section, we describe our approach to determine the interference density only from the equality of an interference equation –as in (5.5)– that we repeat here:

$$E_{mod} \equiv \begin{cases} f_{-1} + \mathbf{f}^t\mathbf{s} = nC + q \\ \\ \text{where } f_k = c_k \textbf{ mod } C \ \forall k \in [-1, d-1], \\ c_k = a_k - a_k,\ d_{-1} = a_{-1} - b_{-1} - \mathbf{d}^t\mathbf{p}, \\ \text{and with } \mathbf{s} \in S_{\mathbf{p}} \end{cases} \tag{5.11}$$

(i.e., we consider $f_{-1} + \mathbf{f}^t\mathbf{s} = nC + q$ only).

Theorem 3 states the main result of this section – the interference density is simply a function of the cache size and cache-line size: $\rho_E \leq \frac{1}{2^{d-1}} \frac{2L}{C}$. To prove this result, we start by showing that the **solutions space** –e.g., the iteration points where an interference equation has a solution– is a regular structure in a rational domain. This structure envelopes all integer solutions and it has an extremely regular organization in cells –or tiles. We determine the interference density by computing the ratio of volumes; that is, we determine the *volume* of the solutions over the *volume* of a solution cell.

We begin with the definition of inner product and inverse vector. The **inner product** of two vectors $\mathbf{u}$ and $\mathbf{v}$ is the vector $\mathbf{s} = \mathbf{u} \cdot \mathbf{v}$, such that $s_k = u_k v_k$. The vector $\mathbf{1} = (1, \cdots, 1)^t$ is the identity vector for the inner product (i.e., $\mathbf{v} \cdot \mathbf{1} = \mathbf{1} \cdot \mathbf{v} = \mathbf{v}$). For every nonzero rational vector $\mathbf{v} \in \mathbb{Q}^d$ –i.e., $v_k \neq 0$– there is one and only one **inverse** vector, denoted by $\mathbf{v}^{-1} \in \mathbb{Q}^d$, such that $\mathbf{v}^{-1} \cdot \mathbf{v} = \mathbf{1}$.

From here on, we denote by $\mathbf{i}_0$ the smallest rational solution for equation E_{mod}. We now describe a regular structure that models the solution space. We define a **grid** as a set of points:

$$\mathcal{G}(\mathbf{i}_0) = \{\mathbf{j} | \mathbf{j} = \mathbf{i}_0 + C\mathbf{f}^{-1} \cdot \mathbf{s}, \text{ such that } \mathbf{s} \in \mathbb{N}^d\}. \tag{5.12}$$

We define a **grid cell** as a d-dimensional rectangle determined by the $2 + d$ vertices $\mathbf{i}_0 + C\mathbf{f}^{-1} \cdot \mathbf{s}$, $\mathbf{i}_0 + C\mathbf{f}^{-1} \cdot (\mathbf{s} + \mathbf{e}_0)$,..., $\mathbf{i}_0 + C\mathbf{f}^{-1} \cdot (\mathbf{s} + \mathbf{e}_{d-1})$ and $\mathbf{i}_0 + C\mathbf{f}^{-1} \cdot (\mathbf{s} + \sum_{j=0}^{d-1} \mathbf{e}_j)$, for any $\mathbf{s} \in \mathcal{G}(\mathbf{i}_0)$. Given an integer u, we define a **band** as the following set of rational points:

$$\begin{aligned} \mathcal{B}(u) \equiv \{\mathbf{b} | -L < u + \mathbf{f}^t \mathbf{b} < L \\ \text{with } \mathbf{b} \in \mathbb{Q}^d \text{ and } u \in \mathbb{N}\}. \end{aligned} \tag{5.13}$$

We define a **band cell** as the set of rational points:

$$\begin{aligned} \mathcal{BC}(u) \equiv \{\mathbf{b} | (-L = u + \mathbf{f}^t \mathbf{b} \cup L = u + \mathbf{f}^t \mathbf{b}) \cap \\ \cap (\forall k \neq j, b_k = 0 \text{ and } j \in [0, d-1])\}. \end{aligned} \tag{5.14}$$

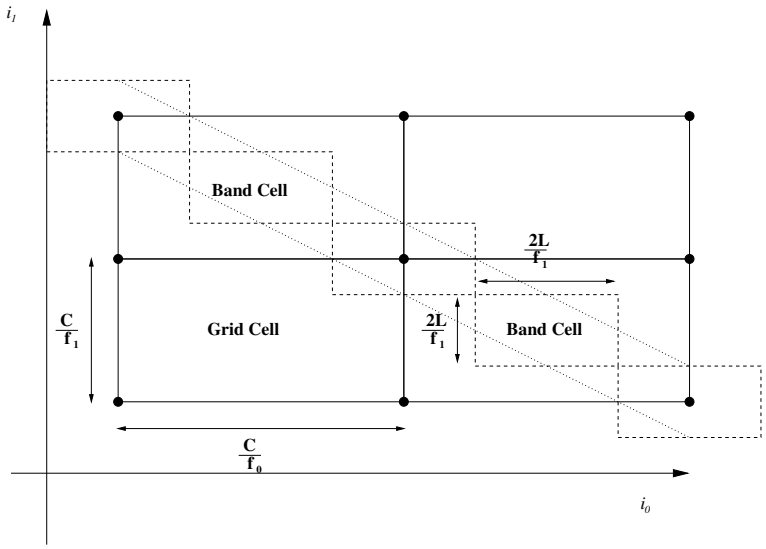

Figure 5.3: Grid cells and band cells in a plane. In a 2-dimensional space grid cells and band cells are rectangles. Note that three bands pass through a grid cell.

For every grid point in $\mathcal{G}(\mathbf{i}_0)$, we determine a band; that is, $\mathcal{B}(n_0C + f_{-1})$. Every point in the band is a solution and, in particular, it has the same value for the variable n. The grid and the bands represent a regular structure (see Fig. 5.3, for a 2-dimensional example). We use the band cells to express the volume of the bands, therefore of the solution number in a grid cell; we eventually determine the interference density for a single grid cell as representative for the entire space determining their volume ratio.

Considering an example as in Fig. 5.3, a grid cell is a rectangle and the band is a line crossing the grid cell on only two grid points. Two different bands are crossing the remaining two vertices. In a 2-dimensional space, the grid cell has an area, in a 3-dimensional space it has a volume. In general, we use the term **volume** to indicate the same *quantity-concept* for any dimension.[7]

Property 3 *Every grid cell has volume* $\frac{C^d}{\prod_{k=0}^{d-1} f_k}$.

We now determine how many bands cross a grid cell and then we determine their volumes. A band is determined by two $(d-1)$-dimensional spaces and, by construction, it passes through

[7]We avoid the use of the term *space* because we use it in another context; that is, iteration space.

grid points. For any grid cell there is only one band splitting the cell in two, so that two vertices are apart. We have three bands crossing a grid cell.

In the following property, we state how many band cells we may find in a grid cell, therefore we have an estimate of the volume of a band intersecting a grid cell.

Property 4 *Every grid cell intersects three bands and up to* $\frac{1}{2^{d-1}}(\frac{\mathcal{C}}{2L})^{d-1}$ *band cells.*

Proof: Consider a grid cell with size $\frac{\mathcal{C}}{f_k}$, $0 \leq k < d$, in a d-dimensional space (i.e., in a 3-dimensional space, it is a cube). The projection of a band on any (d-1)-dimensional space has a number of band cells as $\frac{1}{2^{d-1}} \prod_{k \neq j} \frac{\mathcal{C}}{f_k} / \frac{2L}{f_k}$ (i.e., in a 3-dimensional space, we have three projections on three planes; on each plane, the band cell projections are $\frac{1}{2}(\frac{\mathcal{C}}{f_k} / \frac{2L}{f_k}) * (\frac{\mathcal{C}}{f_j} / \frac{2L}{f_j})$ with $j \neq k$). □

Property 5 *Every band cell has volume at most* $\frac{(2L)^d}{\prod_{k=0}^{d-1} f_k}$.

When we have an estimate of the volume of a band cell and we have the number of bands cells, we have an estimate of the volume of a band. The last step is to show that this regular structure, made of a grid and bands, is dense, as it contains all integer solutions.

Lemma 1 *For any integer solution* $\mathbf{z}$ *of equation* E_{mod}*, there is a grid point in the band passing through* $\mathbf{z}$*.*

Proof: By definition, $Cn_0 + q_0 = \mathbf{f}^t \mathbf{i}_0$ and $Cn_1 + q_1 = \mathbf{f}^t \mathbf{z}$, without loss of generality consider $n_1 > n_0$. A band is a space for which each rational point is a solution for the equation with same value of n, we prove the lemma as soon as we show that $\mathbf{p}$ exists so that $Cn_2 + q_2 = \mathbf{f}^t(\mathbf{i}_0 + C\mathbf{f}^{-1} \cdot \mathbf{p})$ and $n_2 = n_1$.

We have $Cn_2 + q_2 = \mathbf{f}^t \mathbf{i}_0 + \mathbf{f}^t C \mathbf{f}^{-1} \cdot \mathbf{p}$; that is, $Cn_2 + q_2 = \mathbf{f}^t \mathbf{i}_0 + C\mathbf{1}^t \mathbf{p}$. We determine n_2: $n_2 = \left\lfloor \frac{\mathbf{f}^t \mathbf{i}_0 + C\mathbf{1}^t \mathbf{p}}{C} \right\rfloor$. We obtain $n_2 = \left\lfloor \frac{\mathbf{f}^t \mathbf{i}_0}{C} + \mathbf{1}^t \mathbf{p} \right\rfloor = \left\lfloor \frac{\mathbf{f}^t \mathbf{i}_0}{C} \right\rfloor + \mathbf{1}^t \mathbf{p} = n_0 + \mathbf{1}^t \mathbf{p}$.

So, $n_2 = n_1$ when $\mathbf{1}^t \mathbf{p} = n_1 - n_2$. There is always such a vector $\mathbf{p}$. □

Finally, we state and prove our estimate for the interference density.

Theorem 3 *If in (5.5), E_{mod}, solution exists and f_k* **mod** $\mathcal{C} \neq 0$, $\forall k \in [0, d-1]$ *and $C \geq 2L$, then $\rho_E \leq \frac{1}{2^{d-1}} \frac{2L}{C}$.*

Proof: By Lemma 1, the grid and the bands on the grid constitute a dense solution space. Every integer solution is in it. The density is computed on a grid cell as the ratio of the volume of a band intersecting a cell over the volume of a grid cell. By Property 4 and 5, there are $\frac{1}{2^{d-1}}(\frac{\mathcal{C}}{2L})^{d-1}$ band cells of volume $\frac{(2L)^d}{\prod_{k=0}^{d-1} f_k}$ in a grid cell. By Property 3 a grid cell has volume $\frac{\mathcal{C}^d}{\prod_{k=0}^{d-1} f_k}$. Then, we have $\rho_E \leq \frac{1}{2^{d-1}} \frac{2L}{C}$. □

Interference Density Analysis, Refined

In this section we present a more detailed analysis of the interference density; we refine the analysis for the integer domain. First, we present some considerations on the approach to solve Diophantine equations [57]. Second, we apply this approach when both the variable n and l are assigned to values, therefore they are constant terms of the equation. We then consider the case when only the variable l is assigned to a value. At last, we present the final result of this section in Theorem 6.

Consider the coefficients in (5.5). In fact, E_{mod} is a Diophantine equation. For the solution of Diophantine equations, we may use the **GCD test** [57]. We determine the great common divisor of the coefficients of the equations –$g = gcd(d_{-1}, d_0, \ldots, d_{d-1}, C, 1)$– and we verify whether the constant factor of the equation, for example ζ, is evenly divided by g. If ζ**mod**$g = 0$ the equation may have solution –we need to check the domain, otherwise the equation has no solution.

Because the free variable l has 1 as coefficient, the $gcd(d_{-1}, d_0, \ldots, d_{d-1}, C, 1) = 1$, the **GCD test** is inconclusive: we cannot conclude whether or not there is any solution. We need to solve the system and verify the constraints on the definition domain.

Banerjee presents a general approach to determine all integer solutions for Diophantine equations –without parameters, using the theory of unimodular matrices. We consider whether or not there is solution for arbitrary values of $n = n_0$ and $l = l_0$. The constant value will be $\zeta = f_{-1} + n_0 C + l_0$, which includes the parameters as well. We can rewrite (5.5) as follows:

$$E_b \equiv \Big\{ \zeta = \mathbf{f}^t \mathbf{s}. \tag{5.15}$$

If g is $gcd_{k\in[0,d-1]}(f_k)$ and $gcd(g, \zeta)$ is not 1, then there exists an unimodular matrix $\mathbf{U}$, so that all the solutions are determined by the following expression:

$$E_{sol} \equiv \begin{cases} \mathbf{i} = \mathbf{U}^t \mathbf{s} \\ \\ \text{where } \mathbf{s} = (\zeta/g, s_1, s_2, \ldots, s_{d-1}) \\ \text{and } s_k \in \mathbb{N}, \forall k \in [1, d-1] \\ \text{and where } \mathbf{U} \in \mathbb{R}^{d\times d} \text{ is unimodular} \\ \text{and} \mathbf{U}\mathbf{f} = (g, 0, \ldots, 0)^t, \end{cases} \tag{5.16}$$

A matrix $\mathbf{U}$ is **unimodular** when it is an upper triangular matrix and it has $|det(\mathbf{U})| = 1$.[8] A unimodular matrix $\mathbf{U}$ is a linear transformation, it has always an inverse matrix $\mathbf{U}^{-1}$ (i.e., such that $\mathbf{U}^{-1}\mathbf{U} = \mathbf{I}$, $\mathbf{I}$ the identity matrix) and $\mathbf{U}^{-1}$ is unimodular as well. Banerjee presents an effective technique in Algorithm 2.1 for the determination of $\mathbf{U}$.

The matrix $\mathbf{U}$ is a 1-1 mapping between the iteration space S and a space $\mathbb{T}$, where $S, \mathbb{T} \subset \mathbb{N}^d$. In $\mathbb{T}$, the solution space is the plane $s_0 = \zeta/g$; the number of integer solutions in $\mathbb{T}$ are as many as in S and all solutions in S are in a plane.

For example, if $\mathbb{T} = \mathbb{N}^2$, the solution space is a line $\mathbf{s} = (\zeta/g, s_0)^t$ and the minimum distance between any two points in $\mathbb{T}$ is 1; that is, $(\zeta/g, 1) - (\zeta/g, 0) = (0, 1)$. Consider the equation $6i_0 - 4i_1 = 10$.[9] The solutions are:

$$\mathbf{i}^t = (5, s_0) \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix}. \tag{5.17}$$

[8] Where $det(\mathbf{U})$ is the determinant of matrix $\mathbf{U}$.
[9] Example 3.5 [57]

Two solutions in $\mathbb{T}$, such as $(5,0)$ and $(5,1)$, are mapped on two solutions in S, $(5,5)$ and $(7,3)$ respectively. We notice that in $\mathbb{T}$ the distance is 1 but in S the distance is more than 1. We want to determine the interference density in the original space –i.e., S– using some of the properties of matrix $\mathbf{U}$.[10] Indeed, we estimate the distance computing the volume of the d-dimensional rectangle that has the two solutions as opposite vertices. To do so, we estimate the size of the rectangle as follows.

We define

$$h_i = \left| \max_{j \in [1,d-1]} u_{j,i} - \min_{j \in [1,d-1]} u_{j,i} \right| \tag{5.18}$$

where $u_{j,i}$ is the element in the i-th column and j-th row in $\mathbf{U}$. Intuitively, the product $\prod_{i=0}^{d-1} h_i$ is a lower bound to the the distance between two solutions in S.

Theorem 4 *If equation $E_b \equiv \zeta = \mathbf{f}^t\mathbf{s}$ has a solution, then the interference density is at most $\rho_{E_b} \leq 1/(\prod_{i=0}^{d-1} h_i)$.*

Proof: Consider the solutions $\mathbf{s}+\mathbf{e}_1, \ldots, \mathbf{s}+\mathbf{e}_{d-1}$. These solutions are mapped to $\mathbf{U}^t\mathbf{s} + \mathbf{U}^t\mathbf{e}_1, \ldots, \mathbf{U}^t\mathbf{s}+\mathbf{U}^t\mathbf{e}_{d-1}$; that is, $\mathbf{U}^t\mathbf{s}+\mathbf{u}_1, \ldots, \mathbf{U}^t\mathbf{s}+\mathbf{u}_{d-1}$. The solutions are the $d-1$ vertices of a bounded region and $\prod_{i=0}^{d-1} h_i$ is a lower bound to the number of integer points in the region. □

When n is not an arbitrary value but it is a variable, the equation may have solutions for different values of n (i.e., the equation is $\zeta = Cn + \mathbf{f}^t\mathbf{s}$). For each solution of n, there is a different parallel plane in $\mathbb{T}$. As long as the planes are far apart, Theorem 4 holds. Otherwise the interference density may be reevaluated as the following theorem states.

Theorem 5 *If an integer $j \in [0, d-1]$ exists such that $h_j > \frac{C}{f_j}$, then $\rho_{E_b^d} \leq \max_k \frac{2h_k f_k}{C} \rho_{E_b}$.*

Proof: For any n, the solution space is a set of parallel planes. When we determine the image of the planes and of the set of points $\mathbf{s}+\mathbf{e}_1, \ldots, \mathbf{s}+\mathbf{e}_{d-1}$ in S. We note that in one dimension,

[10] Because $\mathbf{U}$ is unimodular, we have that $|\mathbf{det}(\mathbf{U})| = |\mathbf{det}(\mathbf{U}^{-1})| = 1$ and therefore we cannot use the determinant to achieve any estimation for the interference density –at least directly.

the distance between any two planes is (asymptotically) C/f_i and in $\mathbf{U}^t\mathbf{s}+\mathbf{u}_1, \ldots, \mathbf{U}^t\mathbf{s}+\mathbf{u}_{d-1}$ there can be at most $\max_{k\in[0,d-1]} \frac{2h_k f_k}{C}$ planes intersecting. Each plane contributes with just one integral solution. By Theorem 4, the proof follows. □

The last case is when for every solution of n there are different solutions of l. The following theorem estimates the interference density in this scenario.

Theorem 6 *If we have a set J so that $h_j < 2L$ with $j \in J$, then* $\rho_E \leq ((\frac{2L}{C})^{|J|} \prod_{j\in J} \frac{1}{h_j}) * \rho_{E_b^{(k=d-|J|)}}$

Proof: In this case, each variable in J satisfies the equation in an interval of size C at most $2L/h_i$ (with $i \in J$) times, therefore with density $\frac{2L}{Ch_i}$. We restrict the investigation on the other $d - |K|$ variables, and we apply Theorem 5. □

5.3 STAMINA Implementation Results

The reuse and interference analysis is implemented in the software package STAMINA (abbreviation for Static Modeling of Interference And reuse). It is built on top of SUIF 1.3 compiler adapting the analysis developed by Ghosh et al. [79] and using *Polylib* [106, 105, 108]. In this section, we consider three cases to explore three important aspects of our analysis. We analyze loop nests presenting: first, parameterized loop bounds; second, only self interference among memory accesses; and last, parameterized loop bounds, parameterized memory accesses and tiling.

STAMINA presents the result of the analysis in two forms (or types): a numeric and a symbolic form.

Numeric form: the output is a table with two contributions –two rows:

- A row is the contribution at **compile time**. It presents the estimation of interference as a function of the cache-line size, at compile time only. We identify the entries in such a row by $\epsilon_{ct}(L)$.

- A row is the contribution at **run time**. It presents the estimation of the interference as a function of the numeric value of the parameters. We identify the entries in such a row by $\epsilon_{rt}(L)$

This distinction between compile time and run time is extremely helpful for an optimizing compiler: a compiler may use the quantitative measure and decide whether or not any adaptation is worthwhile to pursue. In other words, if the contribution at run time is overall negligible, we can set the optimal line size at compile time; otherwise, we may introduce annotations to the original code and drive adaptation at run time.

Symbolic form: we represent the effect of the cache-line size by a symbolic function. We insert code computing the symbolic function as header of the loop nest and we evaluate it at run time, before the loop nest execution.

We assume that the scheduling of the references (i.e., loads and stores) follows two criteria. First, the computation is performed so that to minimize the number of temporaries [113] for each statement. Second, a reference may be loaded once or more in the inner loop. We assume the final scheduling from the source code only, because it is very difficult to retain high level information from the source code to the assembly code and vice versa (e.g. after all optimizations such as scheduling and register allocation). For example, we label each reference with an integer and we assume a possible reference schedule. This schedule is automatically determined and it is used for the interference analysis. (Note, this is not a limitation of the approach but of the implementation.) We assume the data cache is a direct mapped of size 16KB.

5.3.1 Case A: SWIM-SPEC 2000

The first application is `swim` from SPEC 2000. It has a main loop with four function calls. Each function has a loop nest for which the loop bounds are parameters introduced at run time. We present results for two of these loop nests.

```
#define N1 1335
#define N2 1335

extern  double U[N1][N2], V[N1][N2], P[N1][N2],UNEW[N1][N2], VNEW[N1][N2],
        PNEW[N1][N2], UOLD[N1][N2],  VOLD[N1][N2], POLD[N1][N2],
        CU[N1][N2], CV[N1][N2], Z[N1][N2], H[N1][N2], PSI[N1][N2];

extern double D0, DX, DY;



void calc1(int M, int N) {


  int i,j;
  double FSDX,FSDY;

  for (i=0;i<M;i++)
    for (j=0;j<N;j++) {
      //      RN 0          =        1         2        3
      CU[i+1][j] = D0*(P[i+1][j]+P[i][j])*U[i+1][j];
      //C      #  1 2 3 0
      //C      RN 4                  5         2        6
      CV[i][j+1] = D0*(P[i][j+1]+P[i][j])*V[i][j+1];
      //C      # 5 2  6 4
      //C      RN 7                      8           6                9
      Z[i+1][j+1] = (FSDX*(V[i+1][j+1]-V[i][j+1])-FSDY*(U[i+1][j+1]
      /* C      RN        3            2       1          10          5   */
                    -U[i+1][j]))/(P[i][j]+P[i+1][j]+P[i+1][j+1]+P[i][j+1]);
      //      # 8 6 9 3 2 1 10 5 7
      //         RN 11        2             3         3         12      12
      H[i][j] = P[i][j]+D0*(U[i+1][j]*U[i+1][j]+U[i][j]*U[i][j]
     //      RN         9          9          13       13
                          +V[i][j+1]*V[i][j+1]+V[i][j]*V[i][j]);
      //      #  3 12  9 13 2 11
    }
}
```

Figure 5.4: SWIM: calc1() in C code. The comment above an instruction presents the reference numbers (RN), and the comment below an instruction presents the order in which the memory references are issued.

In Fig. 5.4, we present one of the loop nests in C language. We analyze the interference for two different matrix sizes, the **reference** size 1335×1335 and the power of two 1024×1024. Our analysis states that for the former there is no interference for any cache-line size, but for the latter there is interference among all references and all cache-line sizes.

Due to the number of equations, it is very difficult to verify the accuracy of the analysis by hand. We simulate 10 of the 800 calls to the *calc*1 routine using **cachesim5** from **Shade** [114]. The routine is compiled with *gcc*/3.1 with $-O2$ flag on. The simulation results for matrix size 1335×1335 confirm our analysis as follows:

Table 5.1: Simulation of the data-cache misses due to 10 calls to *calc1()*; L = cache-line size in Bytes, DCMR = Data Cache Miss Rate. Spatial locality is fully exploited in *calc1()*.

L	32	64	128	256
DCMR	11.916 %	5.960%	2.982%	1.493%

The case for power of two matrices is confirmed as well –not reported here.

A more interesting case is in procedure calc2(), see Fig. 5.5. STAMINA determines that reference 16, $CU(I+1,J)$, interferes with reference 8, $H(I+1,J)$, when the line size is larger than 128B. The software determines the trade off between spatial locality exploitation and interference, but even though only two references are interfering, the optimal line size proposed is 128B. Note that the analysis is able to indicate which references are involved and when there is interference.

Using Shade simulator, we validate our the analysis as follows:

Table 5.2: Simulation of the data-cache misses due to 10 calls to *calc2()*; L = cache-line size in Bytes, DCMR = Data Cache Miss Rate. Optimal cache-line size is 128B.

L	32	64	128	256
DCMR	11.968 %	6.739%	**3.371**%	4.091%

The execution of SWIM with reference input (matrices of size 1335×1335) takes 1 hour on a Sun ultra 5, 450MHz. Any full simulation takes at least 50 times more. In contrast, our analysis takes less than one minute for each loop nest whether or not there is interference (i.e., for SWIM our analysis takes less than 5 minutes).

5.3.2 Case B: Self Interference

We now consider the case when an application has self interference. Self interference happens when two references of the same array, or the same reference in different iterations, interfere in cache. The example, Fig. 5.6, is the composition of six loops with only one memory reference in each.

```
      SUBROUTINE CALC2
C
C        COMPUTE NEW VALUES OF U,V,P
C
      IMPLICIT REAL*8   (A-H, O-Z)
      PARAMETER (N1=1335, N2=1335)

      COMMON  U(N1,N2), V(N1,N2), P(N1,N2),
     *        UNEW(N1,N2), VNEW(N1,N2),
     1        PNEW(N1,N2), UOLD(N1,N2),
     *        VOLD(N1,N2), POLD(N1,N2),
     2        CU(N1,N2), CV(N1,N2),
     *        Z(N1,N2), H(N1,N2), PSI(N1,N2)
C
      COMMON /CONS/ DT,TDT,DX,DY,A,ALPHA,ITMAX,MPRINT,M,N,MP1,
     1              NP1,EL,PI,TPI,DI,DJ,PCF
      TDTS8 = TDT/8.D0
      TDTSDX = TDT/DX
      TDTSDY = TDT/DY

C SPEC removed CCMIC$ DO GLOBAL
      DO 200 J=1,N
      DO 200 I=1,M
C           0          1
         UNEW(I+1,J) = UOLD(I+1,J)+
C                      2          3          4            5
     1        TDTS8*(Z(I+1,J+1)+Z(I+1,J))*(CV(I+1,J+1)+CV(I,J+1)
C               6       7                  8         9
     2        +CV(I,J)+CV(I+1,J))-TDTSDX*(H(I+1,J)-H(I,J))
C    #   2 3 4 5 6 7 8 9 1 0
C       10            11                  2           12
         VNEW(I,J+1) = VOLD(I,J+1)-TDTS8*(Z(I+1,J+1)+Z(I,J+1))
C                13          14        15      16
     1        *(CU(I+1,J+1)+CU(I,J+1)+CU(I,J)+CU(I+1,J))
C                      17       9
     2        -TDTSDY*(H(I,J+1)-H(I,J))
C    # 2 12 13 14 15 16 17 9 11 10
C        18          19                16        15
         PNEW(I,J) = POLD(I,J)-TDTSDX*(CU(I+1,J)-CU(I,J))
C                      5         6
     1        -TDTSDY*(CV(I,J+1)-CV(I,J))
C     #  16 15 5 6 19 18
 200  CONTINUE
      RETURN
      END
```

Figure 5.5: SWIM: calc2() in FORTRAN. The comment above an instruction presents the reference numbers (RN), and the comment below an instruction presents the order in which the memory references are issued.

```
#define CACHE_SIZE 16384
int A[CACHE_SIZE /16][(CACHE_SIZE+16)/4];
int B[CACHE_SIZE / 32][(CACHE_SIZE+32)/4];
int C[CACHE_SIZE / 64][(CACHE_SIZE+64)/4];
int D[CACHE_SIZE / 128][(CACHE_SIZE+128)/4];
int E[CACHE_SIZE / 256][(CACHE_SIZE+256)/4];
int F[CACHE_SIZE / 512][(CACHE_SIZE+512)/4];

int main ()
{
  int i,j,k,l;
  int step;
  l = 0;

  for (j=0;j<4;j++)   // LOOP 0
    for (k = 0; k < CACHE_SIZE / 16 k++)
      A[k][j]++;

  for (j=0;j<8;j++)   // LOOP 1
    for (k = 0; k < CACHE_SIZE / 32; k++)
      B[k][j]++;

  for (j=0;j<16;j++)  // LOOP 2
    for (k = 0; k < CACHE_SIZE / 64; k++)
      C[k][j]++;

  for (j=0;j<32;j++)  // LOOP 3
    for (k = 0; k < CACHE_SIZE / 128; k++)
      D[k][j]++;

  for (j=0;j<64;j++)  // LOOP 4
    for (k = 0; k < CACHE_SIZE / 256; k++)
      E[k][j]++;

  for (j=0;j<128;j++) // LOOP 5
    for (k = 0; k < CACHE_SIZE / 512; k++)
      F[k][j]++;

}
```

Figure 5.6: Case B: Self Interference

Each memory reference has a different spatial reuse and the reuse vector is *long*. Each loop accesses a matrix by row and updates a small part of it. Even though the matrix access is done by row, instead of by column, spatial locality may be exploited because of the matrix size. In practice, the number of columns for each matrix is chosen so that each loop has a different optimal line size.

For example, in LOOP 0 a cache-line size of 8 Bytes does not have any self interference, and a cache-line size of 16B has spatial reuse; for larger line size there is always interference because elements in two contiguous rows share the same cache line.

STAMINA recognizes that the spatial reuse goes across one iteration of the outermost loop. In the current implementation, it fixes the value of the interference density at $\rho = 1$ (STAMINA assumes that there is a capacity miss, because in general the distance is not a constant and it cannot be compared to the cache size). For this particular case, we achieve a tight estimation. In general we achieve an over estimation. Notice that the existence of interference plays the main role, it discriminates when there is interference and when to count the interferences. In Table 5.3, we report the results of the analysis.

Table 5.3: STAMINA's result for Self interference example. Loop 4 and 5 have no interference for any line size, the output is set to zero. In bold face, we present the optimal ϵ per cache-line size and loop.

	Line	8	16	32	64	128	256
Loop 0	$\epsilon_{ct}(L)$	0.50	**0.25**	1.00	1.00	1.00	1.00
Loop 1	$\epsilon_{ct}(L)$	0.50	0.25	**0.12**	1.00	1.00	1.00
Loop 2	$\epsilon_{ct}(L)$	0.50	0.25	0.12	**0.06**	1.00	1.00
Loop 3	$\epsilon_{ct}(L)$	0.50	0.25	0.12	0.06	**0.03**	1.00
Loop 4	$\epsilon_{ct}(L)$	0.00	0.00	0.00	0.00	0.00	**0.00**
Loop 5	$\epsilon_{ct}(L)$	0.00	0.00	0.00	0.00	0.00	**0.00**

5.3.3 Case C: Matrix Multiply

In the previous cases (Section 5.3.1 and 5.3.2), the optimal cache-line size is set at compile time and therefore the analysis returns a numeric-form result. In this section, we present a case where the analysis returns a symbolic-form result, to comply with the dynamic behavior of the application.

The examples are simple and we check the accuracy of the analysis manually. At the same time, the problem size is large and it is not practical an exhaustive collection of simulations.

We analyze a variation of the common ikj-matrix-multiply algorithm [2], see Fig. 5.7. Matrix A, B and C are square matrices, and in particular matrix B and C are power of two. We choose the size of the matrices so that if there is interference, due the reference

```
/* B[0][0] 120000000
   C[0][0]
*/
#define MAX 4000
#define MAXCOL 2048


double A[MAX][MAX],  B[MAXCOL][MAXCOL],  C[MAXCOL][MAXCOL];
void ijk_matrix_multiply( int n, int m) {

  int i,j,k;


  for(i=0;i<n;i++)
    for(k=0;k<n;k++)
      for(j=0;j<n;j++)
        C[i][j+3] += A[i][k+m] * B[k][j];


}
```

Figure 5.7: Matrix Multiply. There are two parameters: n and m. The first affects the loop bounds and the latter affects the access offset on matrix A. We assume that $0 < n, m < 64$

to A, it is rare. The index computation for A is parameterized ($0 \leq n \leq 64$). [11] Due to the upper bounds of the parameters, A does not interfere with any other matrix. Even if it could, the interference density would be small. We distinguish two different contributions: at compile time, $\epsilon_{ct}(L)$; at run time, $\epsilon_{rt}(L)$. We have $\epsilon_{rt}(L) = 0$ for any L, and $\epsilon_{ct}(\{8, 16, 32, 64, 128, 256\}) = \{2.00, \mathbf{1.00}, 2.00, 2.00, 2.00, 2.00\}$.

The reference to A does not interfere with the references to C and to B for $0 \leq n, m \leq 64$. It would, only if we use lager values for the parameters. The suggested optimal cache-line size is 16 Bytes. We simulate the number of cache misses for some values of m, n (only a subset of the possible 64^2 pairs is presented) and for different cache-line sizes. The experimental results are in Table 5.4.

STAMINA proposes 16B as optimal cache-line size because it currently assumes the interference density as $\rho = 1$ for every cache-line size. However ρ is 1/2 for L=32B and the

[11] Note that we can handle larger cases; this is to yield a clearer example.

Table 5.4: Data cache misses for Matrix Multiply, Fig. 5.7, using *shade* cache simulator. We present cache misses only for cache-line size 16B, 32B and 64B (cache-line size 8B and 128B are omitted).

m	n	miss L=16B	miss L=32B	miss L=64B
4	4	5670	3740	2856
8	8	6107	4160	**3531**
12	12	7304	5330	**5011**
16	16	9645	**7632**	8800
20	20	13532	**11507**	13818
24	24	19355	**17309**	23355
28	28	27480	**25397**	33922
32	32	38283	**36159**	51881
36	36	52373	**50493**	70606
40	40	69782	**68011**	99709
44	44	91390	**90106**	128561
48	48	116546	**115064**	170124
52	52	146286	**144598**	209300
56	56	181488	**180018**	267866
60	60	221808	**220361**	321279
63	63	260740	**260418**	380464

two cache-line sizes (16B and 32B) are equally good, and a larger cache line may improve overall performance. In practice, simulation results suggest that a cache-line size of 32B is optimal for a negligible difference (Table 5.4). A solution to this problem is presented shortly in the next example, where we represent the cache misses as a symbolic expression of the cache-line size.

We analyze the blocked version of matrix multiplication, see Fig. 5.8. We analyze only the loop nest in the procedure *ikj_mm*, and we find that $\epsilon_{ct}(L) = 0$ for any L and $\epsilon_{rt}(\{8, 16, 32, 64, 128, 256\}) = \{2.00, 2.00, 2.00, 2.00, 2.01, 2.03\}$. Every reference interferes with every other reference. The interference due to matrix A is negligible since the matrix access is an invariant for the inner loop. The interference between C and B can be at every iteration point. There is no interference whenever $|m - n| \textbf{mod } \mathcal{C} = L$. This example is very peculiar because the cache-line size is not set once per loop nest, it is determined at

```
#define MAX 2048
double A[MAX][MAX],  B[MAX][MAX],  C[MAX][MAX];

void ikj_matrix_multiply_4( int x,int y, int z, int m, int n, int p )
{
  int i,j,k;
  for(i=0;i<x;i++)
    for(k=0;k<y;k++)
      for(j=0;j<z;j++)
        C[i][j+m] += A[i][k+n] * B[k][j+p];
}

void matrix_multiply_new_tiling() {
  int ii,jj,kk;
  for(kk=0;k<MAX/b;kk++)
    for(ii=0;i<MAX/b;ii++)
      for(jj=0;j<MAX/b;jj++)
        ikj_matrix_multiply_4(min(b,MAX-ii*b), min(b,MAX-jj*b),
                              min(b,MAX-kk*b), (ii*MAX+jj)*b,
                              (ii*MAX+kk)*b,(kk*MAX+jj));
}
```

Figure 5.8: Tiling Matrix Multiplication. We have 6 parameters: x, i and k are used to specify the loop bounds, m, n and p are used to modify the access to matrix C, A and B respectively.

run time.

We expect to have a symbolic form of the type $\epsilon(L) = \eta_{R_C}(L) + \eta_{R_B}(L) + \eta_{R_A}(L)$. We know that in this particular case $\eta_{R_A} < L/16384 * 2 \sim 0$. STAMINA produces a symbolic output where C_0 is 16384, Δ is $|8n - 8m|\%C_0$ and $\mathbb{1}(x)$ is 1 is $x \geq 0$ and 0 otherwise (where % is the C-language remain operator):

$$\epsilon(L) = 2\min(1, \mathbb{1}(\Delta - L)\frac{8}{L} + \mathbb{1}(8 - \Delta)\frac{8 - \Delta}{L}) \tag{5.19}$$

which has minimum when L is 16B and 32B. In fact, reference C has spatial reuse and it may interfere with B, mainly: $\eta_{R_C}(L) = \mathbb{1}(L - \Delta)\frac{L-\Delta}{L} + \frac{8}{L}$. For example, when $n = m = 0$, references R_C and R_B interfere at any iteration and no optimal line size exists; otherwise, if $m = 3$ and $n = 0$ (notice that this is the example in Fig. 5.7) the optimal line size is 32B. Automatically, the symbolic form and the numerical form are used to insert a function driving adaptation in the source code, before the loop nest.

For the example in Fig. 5.7, the analysis takes up to two minutes. For the blocked matrix

multiplication in Fig. 5.8, the analysis takes more than 8 hours, on a Sun ultra 5 450MHz. The difference of the execution times is expected. For the former case, the existence test has to investigate a relatively small iteration space. For the latter case, the search for the existence of the integer solution is extremely time consuming, because we need to search a space of 2048^9 points. [12]

5.4 Stamina Conclusions

We present a fast approach to determine statically the effect of the data cache-line size on the performance of scientific applications. We use the static cache model introduced by Ghosh et al. [79] and we present an approach to analyze parameterized loop bounds and memory references. The approach is designed to investigate the trade-off between spatial reuse and interferences of perfect loop nests on direct mapped cache. Experimental results demonstrate the accuracy and efficiency of our approach.

[12] The deployment of watch-dogs may be advised at this time, arguably if a solution is difficult to find, then the interference density should be small and negligible.

CHAPTER 6

A Cache with Dynamic Mapping

Dynamic Mapping is an approach to cope with the loss of performance due to cache interference and to improve performance predictability of blocked algorithms for modern architectures. A classic example of blocked algorithm where cache interference may disrupt performance is matrix multiply: in practice, if we tile the matrix multiply 3-level loop for a data cache of size 16KB by using an optimal tile size, we achieve an average data-cache miss rate of 3%. The average measure does not show that there are data-cache miss peaks of 16% and they are due to data interference only.

Dynamic Mapping is a software-hardware approach where the mapping in cache is determined at compile time, by either manipulating the address used by the data cache or by a simple register allocation. This approach makes possible a sensible reduction of cache misses, which translates into twofold and more speed-ups by eliminating data-cache miss spikes altogether.

Dynamic mapping has the same goal as other approaches proposed in the literature, but its main contribution is in the way it achieves the data cache miss reduction. In fact, dynamic mapping determines a cache mapping statically and, thus, before issuing a load/store with little hardware support. Dynamic mapping uses the computational power of the processor –instead of the memory controller or the data cache mapping– and it has no effect on the data-access time in memory and cache. We may describe dynamic mapping succinctly as

an approach combining several concepts, such as non-standard cache mapping functions and data layout reorganization however, potentially, without any of the overheads common in these approaches.

This chapter is organized in two major sections. First, in Section 6.1, we introduce a code transformation, more specifically, a register allocation and instruction scheduling, to reduce the effects of data-cache interference. We start the chapter presenting optimizations that transform the original code so as to exploit the architecture.

Second, from Section 6.2, we present compiler-driven data cache adaptations, especially cache mapping. We present a general data-cache mapping and its properties in Section 6.2.1. Then, we present a dynamic data-cache mapping by hardware only in Section 6.2.2, and by software only in Section 6.3.

6.1 Spatial Register Allocation

In this section, we introduce a rather simple idea to cope with data cache interference by using carefully data allocation to the **register file** (**RF**), which is the ultimate recipient for the manipulation of data coming/going from/to memory. Consider the following scenario: if we know that cache interference will prevent the full reuse of a cache line, we may load the data, which are in a cache line, into (continuous) registers all at once. We may then perform the computation and write the data back in cache and memory when needed and all at once. Such an approach will reduce the effect of cache interference and it will speed up the performance of the computation.

Conventionally, the scheduling of the instructions in a basic block of a loop nest is determined by the parallelism available – i.e., based on number of functional units, the number of registers, the data dependency and the ability to hide as much as possible the cache/memory latency. Note that the objectives of hiding data access latency and the efficient utilization of registers are often in contrast, because the former increases the register pressure and often

forcing the compiler to let go the data stored in register back in memory (i.e., in the stack) by the introduction of specific instructions called **register spills**.

In this section, we show that a data-cache interference analysis can be used to decide the trade off between cache-latency hiding and data allocation to registers and, in practice, drive the instruction scheduling by a cache aware approach.

Consider the following example, where the functional unit adder is composed of three stages (i.e., the result of an addition is available after three cycles since the operation has been issued). If we access data in memory, a load takes T cycles. If we access data in cache, a load takes one cycle. The cache line can allocate four array elements.

```
for (i=0;i<n;i++)
    C[i] += A[i]+B[i];
```

This example is vectorizable and we may unroll it three times to exploit more parallelism.

```
for (i=0;i<n;i+=3) {
    C[i  ] += A[i  ]+B[i  ];
    C[i+1] += A[i+1]+B[i+1];
    C[i+2] += A[i+1]+B[i+2];
}
```

Loop unrolling is used to expose more (independent) instructions, which may increase the throughput because of a more efficient utilization of parallel and pipelined functional units. However, parallelism demands registers to load and write data independently, this pressures the register file. (Other optimizations can be applied as well; for examples on loop transformations see [115]).

Here, we assume that we have only one adder and one blocking load unit with parallel write and load port. We use scalar replacement to represent load/write instructions and we align the code to represent parallelism (as `gcc` represents a possible loop nest instruction schedule).

Example 6.1.1 ASAP scheduling*:*

```
i=0;
a0 = A[i  ];
```

```
b0 = B[i  ];
a1 = A[i+1];   b0 = a0+b0;
b1 = B[i+1];
a2 = A[i+2];   b1 = a1+b1;
for (;i<n-3;i+=3) {
    b2 = B[i+2];                  C[i  ] = b0;
    a0 = A[i+4];   b2 = a2+b2;
    b0 = B[i+4];                  C[i+1] = b1;
    a1 = A[i+5];   b0 = a0+b0;
    b1 = B[i+5];                  C[i+2] = b2;
    a2 = A[i+5];   b1 = a1+b1;
}
 ..... (tail) ....
```

The static scheduling has an efficient utilization of the adder and load unit. Indeed, it issues a load instruction at any cycle and an addition every other cycle. The loop has inherent data spatial locality and the computation exploits such a locality by using data stored in the cache line fully, at least, as long as no interference in cache occurs. We assume that, on a data miss, the cache stalls for T cycles to retrieve the missing data on higher levels of the memory hierarchy. Otherwise, on a cache hit, the cache deliver the data in one cycle.

In practice, without cache interference, the execution of the loop nest takes $\frac{n}{3}2(T+3)$ (cycles) if we assume that the array A and B are not in cache already. However, if we have that the references $A[i]$ and $B[i]$, for any i, interfere in cache, the execution will take longer, up to $\frac{n}{3}2(3T) = 2nT$, and the load unit is the performance bottleneck.

In general, if we identify as $\Delta = |A - B| \mathbf{mod} C$ (i.e., the difference of the address starting point of matrix **A** and **B** modulo C, where C is size of the cache), there are $\max(4-\Delta, 1)$ misses every 4 accesses (4 is the cache line size in matrix element). We can see that the execution time is $2n[\frac{\max(4-\Delta,1)}{4}T + \frac{\min(\Delta,4)}{4}]$. We may deploy a scheduling, which is aware of the interference and prevents the latency penalty of a cache miss as follows:

Example 6.1.2 Spatial scheduling*:*

```
i=0;
a0 = A[i  ];
a1 = A[i+1];
a2 = A[i+2];
```

```
a3 = A[i+3];
for (;i<n-4;i+=4)
    b0 = B[i];
    b1 = B[i+1];   b0 = a0+b0;
    b2 = B[i+2];   b1 = a1+b1;
    b3 = B[i+3];   b2 = a2+b2;
    a0 = A[i+4];   b3 = a3+b3;  C[i  ] = b0;
    a1 = A[i+5];                C[i+1] = b1;
    a2 = A[i+6];                C[i+2] = b2;
    a3 = B[i+7];                C[i+3] = b3;

 .... (tail) ....
```

The execution of this loop nest takes $\frac{n}{4}(2T + 6)$ cycles.

Note, in case there is interference: we do not use minimum number of registers but we have optimal execution times; When there is no interference in cache the two scheduling have should achieve the same performance, however they require a different number of registers.

Notice that the instruction scheduling in Example 6.1.1 *shuffles* the loads of matrix A with loads of matrix B so to issue the addition (of the two operands) as soon as possible. In contrast, the instruction scheduling in Example 6.1.2 exploits the cache spatial locality storing the cache line in the register file before the beginning of any computation.

These examples are somehow didactic however they present a scenario for which the optimal scheduling may depend on whether or not there is cache interference. In fact, cache spatial locality can be exploited at register level to reduce interference. Furthermore, data-cache analysis is becoming a key feature for embedded compiler and, thanks to new compiler techniques, practical for general-purpose compilers. Data-cache analysis allows the driving of a tailored scheduling that may circumvent the performance bottleneck by exploiting cache locality.

6.1.1 The Algorithm

In the following, we present a brief description of our approach:

1. We analyze every perfect loop nests and every array access. We determine spatial

reuse, spatial reuse vector and cache interference for each memory reference.

2. We estimate the effects of cache interference w.r.t. the case with full cache line utilization. We apply any optimization only when beneficial.

3. We unroll the inner loop (or inner loops) as many times as the cache line size in matrix elements (ℓ) divided by the length of the shortest spatial reuse vector ($s_{\min}$). We unroll $k = \ell/s_{\min}$ times. We determine the number of scalar variables/registers to hold each unique memory reference.

4. For each memory reference in the original loop, we have now up to k new references. We load them into continuous scalar variables-register. We have cliques of up to k memory references. The scheduling of the instructions is based on any ASAP scheduling approach, but when a variable of the clique is loaded, all of them are loaded.

   We apply a scalar assignment, removing redundant assignment. If the number of scalar variables are more than the number of registers, we can tolerate spills. Data are stored continuously in the stack (no interference and high reuse in the lower level in the memory hierarchy).

5. When a scalar variable is not used, it is freed and can be used by another clique.

In the following, we apply our approach on SWIM.

6.1.2 Real Case: SWIM from SPEC 2000

SWIM is highly vectorizable application, where there is temporal reuse but mostly spatial locality. There are four basic loop nests in three different procedures. We present them one at a time in the following from Example 6.1.3 to Example 6.1.6.

Example 6.1.3 *The following loop nest is used to prepare the output matrices –of the whole computation in SWIM.*

```
      DO 3500 ICHECK = 1, MNMIN
       DO 4500 JCHECK = 1, MNMIN
       PCHECK = PCHECK + ABS(PNEW(ICHECK,JCHECK))
       UCHECK = UCHECK + ABS(UNEW(ICHECK,JCHECK))
       VCHECK = VCHECK + ABS(VNEW(ICHECK,JCHECK))
4500   CONTINUE
       UNEW(ICHECK,ICHECK) = UNEW(ICHECK,ICHECK)
    1  * ( MOD (ICHECK, 100) /100.)
3500   CONTINUE
```

The inner loop in Example 6.1.3 does not exploit spatial locality, because the matrices are accessed in row major (in FORTRAN matrices are stored in column major). The loops can be safely interchanged without change the meaning of the computation and improving performance.

Example 6.1.4 *The following loop nest is from the routine* `CALC1` *and it exploits spatial locality - as well as some temporal locality.*

```
      DO 100 J=1,N
      DO 100 I=1,M
      CU(I+1,J) = .5D0*(P(I+1,J)+P(I,J))*U(I+1,J)
      CV(I,J+1) = .5D0*(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)
     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)+.25D0*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
     1               +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
  100 CONTINUE
```

In practice, in the inner loop in Example 6.1.4, the computation accesses matrix **P** in two different columns with leading references `P(I,J)` and `P(I,J+1)`, respectively. In similar fashion, the computation accesses matrices **V** and **U**.

Example 6.1.5 *The following loop nest is from the routine* `CALC2`.

```
      DO 200 J=1,N
      DO 200 I=1,M
         UNEW(I+1,J)=UOLD(I+1,J) + TDTS8*(Z(I+1,J+1)+Z(I+1,J))*(CV(I+1,J
   1           +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        +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*(CV(I ,J
   1          +1)-CV(I,J))
   200 CONTINUE
```

In practice, the computation of the loop nest in Example 6.1.5 accesses the operand matrices exploiting mostly spatial reuse.

Example 6.1.6 *The last loop nest is from routine* `CALC3`.

```
      DO 300 J=1,N
      DO 300 I=1,M
      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)
  300 CONTINUE
```

The loop nest in Example 6.1.6 has exactly the format of a matrix update, as our introductory example and for which we presented two possible instruction schedules in Example 6.1.1 and Example 6.1.2.

We apply loop unrolling and scalar replacement to the Examples 6.1.4-6.1.6. We apply by hand scalar replacement and, then, the code is used as input to the native compiler.

SWIM is an interesting application. We may notice that for power of two matrices, every matrix access conflicts with every other matrix access. In the following, we present the optimizations for each example with and without spatial reuse. Example 6.1.3 can be unrolled and optimized as in Figure 6.1. In the loop nest on the left of Figure 6.1, the loads of the same matrix elements are gathered together so that four consecutive elements are read in registers at once - before other data are brought in. The sequence of additions starts as soon as the first element in the other matrix is in a register. Four registers are allocated for the matrix elements and, they may be reused as temporaries (if optimal performance is the goal) to hide the adder latency. In the right loop nest the loads are shuffled, but no registers are used as temporaries.

For completeness, we present all transformations in Section 6.1.4, from Figure 6.3 to Figure 6.7. For all loop nests, note there is a very contained code expansion because of the loop unrolling, see Figure 6.2 and 6.3.

```
      DO 3500  JCHECK= 1, MNMIN
         DO 4500 ICHECK = 1, MNMIN-4,4
            PNEW0 = PNEW(ICHECK,JCHECK)
            PNEW1 = PNEW(ICHECK+1,JCHECK)
            PNEW2 = PNEW(ICHECK+2,JCHECK)
            PNEW3 = PNEW(ICHECK+3,JCHECK)
            PCHECK = PCHECK+ABS(PNEW0)+ABS(PNEW1)+ABS(PNEW2)+ABS(PNEW3)
            UNEW0 = UNEW(ICHECK,JCHECK)
            UNEW1 = UNEW(ICHECK+1,JCHECK)
            UNEW2 = UNEW(ICHECK+2,JCHECK)
            UNEW3 = UNEW(ICHECK+3,JCHECK)
            UCHECK = UCHECK+ABS(UNEW0)+ABS(UNEW1)+ABS(UNEW2)+ABS(UNEW3)
            VNEW0 = VNEW(ICHECK,JCHECK)
            VNEW1 = VNEW(ICHECK+1,JCHECK)
            VNEW2 = VNEW(ICHECK+2,JCHECK)
            VNEW3 = VNEW(ICHECK+3,JCHECK)
            VCHECK = VCHECK+ABS(VNEW0)+ABS(VNEW1)+ABS(VNEW2)+ABS(VNEW3)
 4500    CONTINUE
         DO 4600 ICHECK = MNMIN-3,MNMIN
c     DO 4600 ICHECK = 1,MNMIN
            PCHECK = PCHECK + ABS(PNEW(ICHECK,JCHECK))
            UCHECK = UCHECK + ABS(UNEW(ICHECK,JCHECK))
            VCHECK = VCHECK + ABS(VNEW(ICHECK,JCHECK))
 4600    CONTINUE
         UNEW(ICHECK,ICHECK) = UNEW(ICHECK,ICHECK)
     1       * ( MOD (ICHECK, 100) /100.)
 3500 CONTINUE
```

(a)

```
      DO 3500 JCHECK = 1, MNMIN
       DO 4505 ICHECK = 1, MNMIN-4,4
          PCHECK = PCHECK + ABS(PNEW(ICHECK  ,JCHECK))
          UCHECK = UCHECK + ABS(UNEW(ICHECK  ,JCHECK))
          VCHECK = VCHECK + ABS(VNEW(ICHECK  ,JCHECK))
          PCHECK = PCHECK + ABS(PNEW(ICHECK+1,JCHECK))
          UCHECK = UCHECK + ABS(UNEW(ICHECK+1,JCHECK))
          VCHECK = VCHECK + ABS(VNEW(ICHECK+1,JCHECK))
          PCHECK = PCHECK + ABS(PNEW(ICHECK+2,JCHECK))
          UCHECK = UCHECK + ABS(UNEW(ICHECK+2,JCHECK))
          VCHECK = VCHECK + ABS(VNEW(ICHECK+2,JCHECK))
          PCHECK = PCHECK + ABS(PNEW(ICHECK+3,JCHECK))
          UCHECK = UCHECK + ABS(UNEW(ICHECK+3,JCHECK))
          VCHECK = VCHECK + ABS(VNEW(ICHECK+3,JCHECK))
 4505    CONTINUE
       DO 4500 ICHECK = MNMIN-3,MNMIN
          PCHECK = PCHECK + ABS(PNEW(ICHECK,JCHECK))
          UCHECK = UCHECK + ABS(UNEW(ICHECK,JCHECK))
          VCHECK = VCHECK + ABS(VNEW(ICHECK,JCHECK))
 4500    CONTINUE
      UNEW(ICHECK,ICHECK) = UNEW(ICHECK,ICHECK)
     1  * ( MOD (ICHECK, 100) /100.)
 3500   CONTINUE
```

(b)

Figure 6.1: (a) spatial reuse: each line is fully read before its use. (b) only unrolling

6.1.3 Experimental Results

Spatial Scheduling is applied to SWIM's loops and experimental results are collected on three different systems: Sun Blade 100, Sun Ultra 5 and Silicon Graphics O2. In practice, the optimizations are implemented at source level –i.e., rewriting the loop body of the loop nests– and, a native compiler takes the source code as input and it produces the final executable codes.

Though we need to make sure the compiler does not reorganize the load/store sequence disrupting spatial locality, however we may allow the compiler to apply other aggressive optimizations. The code optimized to exploit spatial locality at register file is compiled with $-O1$ (i.e. gcc) flag on (i.e., local optimizations only). We measure the execution time and we adjust the optimization flags as long as we achieve performance and, by direct inspection of the assembly code, the sequence of load follows our suggestions. The process is repeated for every architecture and every compiler.

We present the execution time for two different input sets: first, for matrices of size 509×509, for which there is no interference; and second, for matrices of size 512×512, for which there is always interference. We modified the original number of iterations in SWIM

so to have a shorter-but-representative execution time of the full application. In fact, now, the main loop iterates for 20 times and we report the time in seconds.

Table 6.1: Notations: **swim** m is the execution time of the original application applied to input matrices of size $m \times m$; **spatialswim** m is the execution time of the application optimized to exploit spatial locality; and, **unrolled** m is the execution time for the application with the basic loop nest simply unrolled four times.

Architecture	**Compiler**	**swim** 509	**spatialswim** 509	**unrolled** 509
Sun Blade 100	g77	8.47	7.29	7.59
Sun ULTRA 5	g77	11.43	7.97	8.56
	f90	11.26	11.23	8.32
SGI	f90	16.62	17.44	17.22

Architecture	**Compiler**	**swim** 512	**spatialswim** 512	**unrolled** 512
Sun Blade 100	g77	47.99	20.70	40.61
Sun ULTRA 5	g77	14.94	9.11	10.50
	f90	13.91	10.91	10.77
SG O2	f90 pad	20.16	20.76	26.35
	f90 no pad	230.24	106.79	145.12

We analyze first the case without interference (matrix of size 509×509). Independently from the compiler and the architecture, we can say that exploiting spatial locality at register level does not harm performance. We can achieve comparable (sometimes better) performance w.r.t. the original SWIM and the unrolled SWIM.

Observing the assembly code generated by the different versions of compiler, only the FORTRAN compiler for SGI reorganizes heavily the code. In this environment, our register allocation is completely neglected and arrays are padded. To compare the effect of padding and code reorganization, we turned off the padding optimization and collected the experimental results (see Table 6.1, where the compiler is indicated as "f90 no pad"). Otherwise the assembly code generated by g77 on Sun architectures, follows our register allocation and it performs a good job on local optimizations.

The input set with interference is more interesting though. In general our approach

outperforms the others (but on SGI). On Sun ULTRA 5, the performance improvement is significant, but not really high. On Sun Blade the improvement is impressive. The main difference between Blade 100 and ULTRA 5 is the size and the associativity of unified second level of cache.

To have a better understanding of the effect of cache misses on the overall performance, we simulated the cache behavior and in particular the reads and the misses on reads

Table 6.2: Read misses on Sun BLADE 100.

Application	**Data cache misses L1**	%	**Data cache miss L2**
swim	46982382	57.53	34736729
unrolled	31372840	48.89	33328515
spatial	19947847	30.87	20119575

In practice, there is a data-cache miss reduction between 36 to 40%, which represents a good utilization of the data-cache line.

6.1.4 Code Transformations (Appendix)

```
      DO 100 J=1,N
         DO 105 I=1,M-4,4
C     Reading a line of P
            P00 = P(I,J)
            P10 = P(I+1,J)
            P20 = P(I+2,J)
            P30 = P(I+3,J)
C     This is a miss because it accesses a new line, we hide latency
C     issuing useful computations
            P40 = P(I+4,J)
C     Reading a line of U
            U00 = U(I,J)
            U10 = U(I+1,J)
            U20 = U(I+2,J)
            U30 = U(I+3,J)
C     This is a miss because it accesses a new line, we hide latency
C     issuing useful computations
            U40 = U(I+4,J)
            T1 = P10  +P00
            T2 = P20  +P10
            T3 = P30  +P20
            T1 = .5D0*T1
            T2 = .5D0*T2
            T3 = .5D0*T3
            T1 = T1*U10
            T2 = T2*U20
            T3 = T3*U30
C     Reading a line of CU
            CU(I+1,J)   = T1
            CU(I+2,J)   = T2
            CU(I+3,J)   = T3
C     Reading a new line of P
            P01 = P(I,J+1)
            P11 = P(I+1,J+1)
            P21 = P(I+2,J+1)
            P31 = P(I+3,J+1)
            P41 = P(I+4,J+1)
C     Reading a new line of V
            V01 = V(I,J+1)
            V11 = V(I+1,J+1)
            V21 = V(I+2,J+1)
            V31 = V(I+3,J+1)
C     These are misses because they accesse new lines, we hide latency
C     issuing useful computations
            V41 = V(I+4,J+1)
            T1 = P01  +P00
            T2 = P11  +P10
            T3 = P21  +P20
            T4 = P31  +P30
            T1 = .5D0*T1
            T2 = .5D0*T2
            T3 = .5D0*T3
            T4 = .5D0*T4
            T1 = T1*V01
            T2 = T2*V11
            T3 = T3*V21
            T4 = T4*V31
C     Reading a line of CV
            CV(I,J+1)   = T1
            CV(I+1,J+1) = T2
            CV(I+2,J+1) = T3
            CV(I+3,J+1) = T4
C     Reading a line of U
            U11 = U(I+1,J+1)
            U21 = U(I+2,J+1)
            U31 = U(I+3,J+1)
C     This is a miss because it accesses a new line, we hide latency
C     issuing useful computations
            U41 = U(I+4,J+1)
            T1 = V11-V01
            T2 = V21-V11
            T3 = V31-V21
            T4 = V41-V31
            T1 = FSDX*T1
            T2 = FSDX*T2
            T3 = FSDX*T3
            T4 = FSDX*T4
c     Reuse of V*0 because they will be read again later, to help the
c     compiler
            V00 = U11-U10
            V10 = U21-U20
            V20 = U31-U30
            V30 = U41-U40
            V00 = FSDY*V00
            V10 = FSDY*V10
            V20 = FSDY*V20
            V30 = FSDY*V30
            T1 = T1-V00
            P01 = P00+P10+P11+P01
C     Here comes the floating division, usually not pipelined and with
C     very high latency, we shuffle some operation to avoid to stall too
C     badly
            Z(I+1,J+1)= T1/P01
            T2 = T2-V10
            P11 = P10+P20+P21+P11
            Z(I+2,J+1)= T2/P11
            T3 = T3-V20
            P21 = P20+P30+P31+P21
            Z(I+3,J+1)= T3/P21
            P31 = P30+P40+P41+P31
            T4 = T4-V30
C     We access different lines
            Z(I+4,J+1)= T4/P31
            CU(I+4,J)   = .5*(P40  +P30)  *U40
C     Reading a new line of V
            V00 = V(I,J)
            V10 = V(I+1,J)
            V20 = V(I+2,J)
            V30 = V(I+3,J)
            U10=U10*U10
            U00=U00*U00
            V01=V01*V01
            V00=V00*V00
            U20=U20*U20
            U10=U10*U10
            V11=V11*V11
            V10=V10*V10
            U30=U30*U30
            U20=U20*U20
            V21=V21*V21
            V20=V20*V20
            U40=U40*U40
            U30=U30*U30
            V31=V31*V31
            V30=V30*V30
            U10 = U10 + U00 + V01 + V00
            U20 = U20 + U10 + V11 + V10
            U30 = U30 + U20 + V21 + V20
            U40 = U40 + U30 + V31 + V30
C     write 1 line of H
            H(I,J)      = P00+.25D0*U10
            H(I+1,J)    = P10+.25D0*U20
            H(I+2,J)    = P20+.25D0*U30
            H(I+3,J)    = P30+.25D0*U40
 105     CONTINUE
         DO 106 I=M-3,M
 1          CU(I+1,J) = .5D0*(P(I+1,J)+P(I,J))*U(I+1,J)
            CV(I,J+1) = .5D0*(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)+.25D0*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
     1           +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
 106     CONTINUE
 100  CONTINUE
```

Figure 6.2: Loop nest in procedure calc1 using spatial register allocation

```
      DO 100 J=1,N
         DO 102 I=1,M-4,4
C     first iteration
            P10 = P(I+1,J)
            P00 = P(I,J)
            U10 = U(I+1,J)
            CU(I+1,J) = .5D0*(P10+P00)*U10
            P01 = P(I,J+1)
            V01 = V(I,J+1)
            U11 = U(I+1,J+1)
            CV(I,J+1) = .5D0*(P01+P00)*V01
            V11 = V(I+1,J+1)
            T1 = FSDX*(V11-V01)
            T2 = FSDY*(U11 -U10)
            P11 = P(I+1,J+1)
C            P01 = P(I,J+1)
            Z(I+1,J+1) = (T1-T2)/(P00+P10+P11+P01)
            U00 = U(I,J)
            V00 = V(I,J)
            H(I,J) = P00+.25D0*(U10*U10+U00*U00 +V01*V01+V00*V00)

C     second iteration
            P20 = P(I+2,J)
C            P10 = P(I+1,J)
            U20 = U(I+2,J)
            CU(I+2,J) = .5D0*(P20+P10)*U20
C            P11 = P(I+1,J+1)
C            V11 = V(I+1,J+1)
            U21 = U(I+2,J+1)
            CV(I+1,J+1) = .5D0*(P11+P10)*V11
            V21 = V(I+2,J+1)
            T1 = FSDX*(V21-V11)
            T2 = FSDY*(U21 -U20)
            P21 = P(I+2,J+1)
C            P11 = P(I+1,J+1)
            Z(I+2,J+1) = (T1-T2)/(P10+P20+P21+P11)
C            U10 = U(I+1,J)
            V10 = V(I+1,J)
            H(I+1,J) = P10+.25D0*(U20*U20+U10*U10 +V11*V11+V10*V10)

C     third iteration
            P30 = P(I+3,J)
C            P20 = P(I+2,J)
            U30 = U(I+3,J)
            CU(I+3,J) = .5D0*(P30+P20)*U30
C            P21 = P(I+2,J+1)
C            V21 = V(I+2,J+1)
            U31 = U(I+3,J+1)
            CV(I+2,J+1) = .5D0*(P21+P20)*V21
            V31 = V(I+3,J+1)
            T1 = FSDX*(V31-V21)
            T2 = FSDY*(U31 -U30)
            P31 = P(I+3,J+1)
C            P21 = P(I+2,J+1)
            Z(I+3,J+1) = (T1-T2)/(P20+P30+P31+P21)
C            U20 = U(I+2,J)
            V20 = V(I+2,J)
            H(I+2,J) = P20+.25D0*(U30*U30+U20*U20 +V21*V21+V20*V20)

C     fourth iteration
            P40 = P(I+4,J)
C            P30 = P(I+3,J)
            U40 = U(I+4,J)
            CU(I+4,J) = .5D0*(P40+P30)*U40
C            P31 = P(I+3,J+1)
C            V31 = V(I+3,J+1)
            U41 = U(I+4,J+1)
            CV(I+3,J+1) = .5D0*(P31+P30)*V31
            V41 = V(I+4,J+1)
            T1 = FSDX*(V41-V31)
            T2 = FSDY*(U41 -U40)
            P41 = P(I+4,J+1)
C            P31 = P(I+3,J+1)
            Z(I+4,J+1) = (T1-T2)/(P30+P40+P41+P31)
C            U30 = U(I+3,J)
            V30 = V(I+3,J)
            H(I+3,J) = P30+.25D0*(U40*U40+U30*U30 +V31*V31+V30*V30)

 102     CONTINUE
         DO 103 I=M-3,M
            CU(I+1,J) = .5D0*(P(I+1,J)+P(I,J))*U(I+1,J)
            CV(I,J+1) = .5D0*(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)+.25D0*(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))
 103     CONTINUE
 100  CONTINUE
```

Figure 6.3: Loop nest in procedure calc1 using loop unrolling

```
      DO 200 J=1,N
         DO 203 I=1,M-3,3
C     7 registers for Z
            Z10 = Z(I+1,J)
            Z20 = Z(I+2,J)
            Z30 = Z(I+3,J)
            Z01 = Z(I,J+1)
            Z11 = Z(I+1,J+1)
            Z21 = Z(I+2,J+1)
            Z31 = Z(I+3,J+1)

C     Reuse of registers for Z**
            Z10 = Z11+Z10
            Z20 = Z21+Z20
            Z30 = Z31+Z30
            Z01 = Z11+Z01
            Z11 = Z21+Z11
            Z21 = Z31+Z21
            Z10 = TDTS8*Z10
            Z20 = TDTS8*Z20
            Z30 = TDTS8*Z30
            Z01 = TDTS8*Z01
            Z11 = TDTS8*Z11
            Z21 = TDTS8*Z21

C     8 Registers for CV
            CV00 = CV(I ,J)
            CV10 = CV(I+1,J)
            CV20 = CV(I+2,J)
            CV30 = CV(I+3,J)
            CV01 = CV(I,J+1)
            CV11 = CV(I+1,J+1)
            CV21 = CV(I+2,J+1)
            CV31 = CV(I+3,J+1)

C    7 Registers for H
            H00 = H(I ,J)
            H10 = H(I+1,J)
            H20 = H(I+2,J)
            H30 = H(I+3,J)
            H01 = H(I ,J+1)
            H11 = H(I+1,J+1)
            H21 = H(I+2,J+1)

C     Reuse of registers for H**
            H30 = (H30-H20)
            H21 = (H21-H20)
            H20 = (H20-H10)
            H11 = (H11-H10)
            H10 = (H10-H00)
            H01 = (H01-H00)
            H30 = TDTSDX*H30
            H21 = TDTSDY*H21
            H20 = TDTSDX*H20
            H11 = TDTSDY*H11
            H10 = TDTSDX*H10
            H01 = TDTSDY*H01

            UOLD1 = UOLD(I+1,J)
            UOLD2 = UOLD(I+2,J)
            UOLD3 = UOLD(I+3,J)

C     Reuse of registers TO HELP THE COMPILER TO SCHEDULE THE INSTRUCTIONS
            T1 = CV11+CV01+CV00+CV10
            T2 = CV21+CV11+CV10+CV20
            CV31 = CV31+CV21+CV20+CV30
            Z10 = Z10*T1
            Z20 = Z20*T2
            Z30 = Z30*CV31

C     BASIC COMPUTATION !!!!!
            UNEW(I+1,J) = UOLD1+Z10-H10
            UNEW(I+2,J) = UOLD2+Z20-H20
            UNEW(I+3,J) = UOLD3+Z30-H30

C     REUSE OF REGISTERS USED AT THE LAST BASIC COMPUTATION
            CV01 = CV01 - CV00
            CV11 = CV11 - CV10
            CV21 = CV21 - CV20
            CV01 = TDTSDY*CV01
            CV11 = TDTSDY*CV11
            CV21 = TDTSDY*CV21
            H10 = CV01
            H20 = CV11
            H30 = CV21

C     HERE ARE ALIVE H* AND Z*, IF WE WANT WE CAN REUSE COMPLETELY THE
C     CV*, BUT TO HAVE A READABLE CODE WE USE DIFFERENT LABELING, LET'S
C     THE COMPILER TO THE REST

            CU00 = CU(I ,J)
            CU10 = CU(I+1,J)
            CU20 = CU(I+2,J)
            CU30 = CU(I+3,J)
            CU01 = CU(I,J+1)
            CU11 = CU(I+1,J+1)
            CU21 = CU(I+2,J+1)
            CU31 = CU(I+3,J+1)

            VOLD0 = VOLD(I,J+1)
            VOLD1 = VOLD(I+1,J+1)
            VOLD2 = VOLD(I+2,J+1)

C     REUSE OF REGISTERS
            T1 = CU11+CU01+CU00+CU10
            T2 = CU21+CU11+CU10+CU20
            T3 = CU31+CU21+CU20+CU30
            Z01  = Z01*T1
            Z11 = Z11*T2
            Z21 = Z21*T3

C     BASIC COMPUTATION !!!!!
            VNEW(I,J+1) = VOLD0-Z01-H01
            VNEW(I+1,J+1) = VOLD1-Z11-H11
            VNEW(I+2,J+1) = VOLD2-Z21-H21

C     REUSE OF REGISTERS H*
            H01 = TDTSDX*(CU10-CU00)
            H11 = TDTSDX*(CU20-CU10)
            H21 = TDTSDX*(CU30-CU20)

            POLD0 = POLD(I,J)
            POLD1 = POLD(I+1,J)
            POLD2 = POLD(I+2,J)

C     BASIC COMPUTATION !!!!!
            PNEW(I,J) = POLD0-H01-H10
            PNEW(I+1,J) = POLD1-H11-H20
            PNEW(I+2,J) = POLD2-H21-H30

 203     CONTINUE
         DO 205 I=M-2,M
C     DO 205 I=I,M
            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
     $           *(CV(I,J+1)-CV(I,J))
 205     CONTINUE
 200  CONTINUE
```

Figure 6.4: Loop nest in procedure calc2 using spatial register allocation

```
      DO 200 J=1,N
         DO 203 I=1,M-3,3
            Z11 = Z(I+1,J+1)
            Z10 = Z(I+1,J)
            T1  = TDTS8*(Z11+Z10)
            CV11 = CV(I+1,J+1)
            CV01 = CV(I,J+1)
            CV00 = CV(I,J)
            CV10 = CV(I+1,J)
            T2 = CV11+CV01+CV00+CV10
            H01 = H(I,J+1)
            H00 = H(I,J)
            T3 = TDTSDY*(H01 -H00)
            UNEW(I+1,J)=UOLD(I+1,J) + T1*T2-T3
            Z01 = Z(I,J+1)
            T1 = TDTS8*(Z11+Z01)
            CU11 = CU(I+1,J+1)
            CU01 = CU(I,J+1)
            CU00 = CU(I,J)
            CU10 = CU(I+1,J)
            T2 = CU11+CU01+CU00+CU10
            H01 = H(I,J+1)
            T3 = TDTSDY*(H01 -H00)
            VNEW(I,J+1)=VOLD(I,J+1)-T1 *T2 -T3
            T1 = TDTSDX*(CU10-CU00)
            T2 = TDTSDY*(CV01-CV00)
            PNEW(I,J)=POLD(I,J)-T1- T2


            Z21 = Z(I+2,J+1)
            Z20 = Z(I+2,J)
            T1  = TDTS8*(Z21+Z20)
            CV21 = CV(I+2,J+1)
C            CV11 = CV(I+1,J+1)
C            CV10 = CV(I+1,J)
            CV20 = CV(I+2,J)
            T2 = CV21+CV11+CV10+CV20
            H11 = H(I+1,J+1)
            H10 = H(I+1,J)
            T3 = TDTSDY*(H11 -H10)
            UNEW(I+2,J)=UOLD(I+2,J) + T1*T2-T3
C            Z11 = Z(I+1,J+1)
            T1 = TDTS8*(Z21+Z11)
            CU21 = CU(I+2,J+1)
C            CU11 = CU(I+1,J+1)
C            CU10 = CU(I+1,J)
            CU20 = CU(I+2,J)
            T2 = CU21+CU11+CU10+CU20
C            H11 = H(I+1,J+1)
            T3 = TDTSDY*(H11 -H10)
            VNEW(I+1,J+1)=VOLD(I+1,J+1)-T1 *T2 -T3
            T1 = TDTSDX*(CU20-CU10)
            T2 = TDTSDY*(CV11-CV10)
            PNEW(I+1,J)=POLD(I+1,J)-T1- T2

            Z31 = Z(I+3,J+1)
            Z30 = Z(I+3,J)
            T1  = TDTS8*(Z31+Z30)
            CV31 = CV(I+3,J+1)
C            CV21 = CV(I+2,J+1)
C            CV20 = CV(I+2,J)
            CV30 = CV(I+3,J)
            T2 = CV31+CV21+CV20+CV30
            H21 = H(I+2,J+1)
            H20 = H(I+2,J)
            T3 = TDTSDY*(H21 -H20)
            UNEW(I+3,J)=UOLD(I+3,J) + T1*T2-T3
C            Z21 = Z(I+2,J+1)
            T1 = TDTS8*(Z31+Z21)
            CU31 = CU(I+3,J+1)
C            CU21 = CU(I+2,J+1)
C            CU20 = CU(I+2,J)
            CU30 = CU(I+3,J)
            T2 = CU31+CU21+CU20+CU30
C            H21 = H(I+2,J+1)
            T3 = TDTSDY*(H21 -H20)
            VNEW(I+2,J+1)=VOLD(I+2,J+1)-T1 *T2 -T3
            T1 = TDTSDX*(CU30-CU20)
            T2 = TDTSDY*(CV21-CV20)
            PNEW(I+2,J)=POLD(I+2,J)-T1- T2

 203     CONTINUE
         DO 204 I=M-2,M
            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*(CV(I
     $           ,J+1)-CV(I,J))
 204     CONTINUE
 200  CONTINUE
```

Figure 6.5: Loop nest in procedure calc2 using loop unrolling

```
      DO 300 J=1,N
         DO 305 I=1,M-4,4
            U0 = U(I  ,J)
            U1 = U(I+1,J)
            U2 = U(I+2,J)
            U3 = U(I+3,J)
            UOLD0 = UOLD(I  ,J)
            UOLD1 = UOLD(I+1,J)
            UOLD2 = UOLD(I+2,J)
            UOLD3 = UOLD(I+3,J)
            UNEW0 = UNEW(I  ,J)
            UNEW1 = UNEW(I+1,J)
            UNEW2 = UNEW(I+2,J)
            UNEW3 = UNEW(I+3,J)


            UOLD(I  ,J) = U0+ALPHA*(UNEW0-2.*U0+UOLD0)
            UOLD(I+1,J) = U1+ALPHA*(UNEW1-2.*U1+UOLD1)
            UOLD(I+2,J) = U2+ALPHA*(UNEW2-2.*U2+UOLD2)
            UOLD(I+3,J) = U3+ALPHA*(UNEW3-2.*U3+UOLD3)

            U0 = V(I  ,J)
            U1 = V(I+1,J)
            U2 = V(I+2,J)
            U3 = V(I+3,J)
            UOLD0 = VOLD(I  ,J)
            UOLD1 = VOLD(I+1,J)
            UOLD2 = VOLD(I+2,J)
            UOLD3 = VOLD(I+3,J)
            UNEW0 = VNEW(I  ,J)
            UNEW1 = VNEW(I+1,J)
            UNEW2 = VNEW(I+2,J)
            UNEW3 = VNEW(I+3,J)

            VOLD(I  ,J) = U0+ALPHA*(UNEW0-2.*U0+UOLD0)
            VOLD(I+1,J) = U1+ALPHA*(UNEW1-2.*U1+UOLD1)
            VOLD(I+2,J) = U2+ALPHA*(UNEW2-2.*U2+UOLD2)
            VOLD(I+3,J) = U3+ALPHA*(UNEW3-2.*U3+UOLD3)

            U0 = P(I  ,J)
            U1 = P(I+1,J)
            U2 = P(I+2,J)
            U3 = P(I+3,J)
            UOLD0 = POLD(I  ,J)
            UOLD1 = POLD(I+1,J)
            UOLD2 = POLD(I+2,J)
            UOLD3 = POLD(I+3,J)
            UNEW0 = PNEW(I  ,J)
            UNEW1 = PNEW(I+1,J)
            UNEW2 = PNEW(I+2,J)
            UNEW3 = PNEW(I+3,J)

            POLD(I  ,J) =  U0+ALPHA*(UNEW0-2.*U0+UOLD0)
            POLD(I+1,J) =  U1+ALPHA*(UNEW1-2.*U1+UOLD1)
            POLD(I+2,J) =  U2+ALPHA*(UNEW2-2.*U2+UOLD2)
            POLD(I+3,J) =  U3+ALPHA*(UNEW3-2.*U3+UOLD3)

            U0 = UNEW(I  ,J)
            U1 = UNEW(I+1,J)
            U2 = UNEW(I+2,J)
            U3 = UNEW(I+3,J)
            U(I  ,J) = U0
            U(I+1,J) = U1
            U(I+2,J) = U2
            U(I+3,J) = U3

            U0 = VNEW(I  ,J)
            U1 = VNEW(I+1,J)
            U2 = VNEW(I+2,J)
            U3 = VNEW(I+3,J)
            V(I  ,J) = U0
            V(I+1,J) = U1
            V(I+2,J) = U2
            V(I+3,J) = U3
            U0 = PNEW(I  ,J)
            U1 = PNEW(I+1,J)
            U2 = PNEW(I+2,J)
            U3 = PNEW(I+3,J)
            P(I  ,J) = U0
            P(I+1,J) = U1
            P(I+2,J) = U2
            P(I+3,J) = U3

  305    CONTINUE
         DO 306 I=M-3,M
c         DO 306 I=1,M
            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)
  306    CONTINUE
  300 CONTINUE
```

Figure 6.6: Loop nest in procedure calc3 using spatial register allocation

```
      DO 300 J=1,N
         DO 303 I=1,M-4,4
            U00 = U(I,J)
            UOLD(I,J) = U00+ALPHA*(UNEW(I,J)-2.*U00+UOLD(I,J))
            V00 = V(I,J)
            VOLD(I,J) = V00+ALPHA*(VNEW(I,J)-2.*V00+VOLD(I,J))
            P00 =  P(I,J)
            POLD(I,J) = P00+ALPHA*(PNEW(I,J)-2.*P00+POLD(I,J))
            U(I,J) = UNEW(I,J)
            V(I,J) = VNEW(I,J)
            P(I,J) = PNEW(I,J)

            U10 = U(I+1,J)
            UOLD(I+1,J) = U10+ALPHA*(UNEW(I+1,J)-2.*U10+UOLD(I +1,J))
            V10 = V(I+1,J)
            VOLD(I+1,J) = V10+ALPHA*(VNEW(I+1,J)-2.*V10+VOLD(I +1,J))
            P10 = P(I+1,J)
            POLD(I+1,J) = P10+ALPHA*(PNEW(I+1,J)-2.*P10+POLD(I +1,J))
            U(I+1,J) = UNEW(I+1,J)
            V(I+1,J) = VNEW(I+1,J)
            P(I+1,J) = PNEW(I+1,J)

            U20 = U(I+2,J)
            UOLD(I+2,J) = U20+ALPHA*(UNEW(I+2,J)-2.*U20+UOLD(I +2,J))
            V20 = V(I+2,J)
            VOLD(I+2,J) = V20+ALPHA*(VNEW(I+2,J)-2.*V20+VOLD(I +2,J))
            P20 = P(I+2,J)
            POLD(I+2,J) = P20+ALPHA*(PNEW(I+2,J)-2.*P20+POLD(I +2,J))
            U(I+2,J) = UNEW(I+2,J)
            V(I+2,J) = VNEW(I+2,J)
            P(I+2,J) = PNEW(I+2,J)

            U30 = U(I+3,J)
            UOLD(I+3,J) = U30+ALPHA*(UNEW(I+3,J)-2.*U30+UOLD(I +3,J))
            V30 = V(I+3,J)
            VOLD(I+3,J) = V30+ALPHA*(VNEW(I+3,J)-2.*V30+VOLD(I +3,J))
            P30 = P(I+3,J)
            POLD(I+3,J) = P30+ALPHA*(PNEW(I+3,J)-2.*P30+POLD(I +3,J))
            U(I+3,J) = UNEW(I+3,J)
            V(I+3,J) = VNEW(I+3,J)
            P(I+3,J) = PNEW(I+3,J)
303      CONTINUE
         DO 304 I=M-3,M
            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)
304      CONTINUE
300   CONTINUE
```

Figure 6.7: Loop nest in procedure calc3 using loop unrolling

6.2 Dynamically Mapped Cache

In this section, we introduce a novel software-hardware approach where the mapping in a data cache is determined at compile time, by manipulating the address used by the data cache. Our approach has the same goals of other approaches proposed in the literature, but, it is an hybrid approach that specify a data-cache mapping at compile time and the actual application of such a mapping at runtime. Dynamic mapping needs little extra hardware and it uses the computational power of the processor –instead of the memory controller or the data cache mapping– and it has no effect on the access time of memory and cache. It is an approach combining several concepts, such as non-standard cache mapping functions and data layout reorganization and, potentially, without any overhead.

6.2.1 General Hardware Mapping

The main result of this section is stated in Theorem 7, where we show that if in a loop nest there are m memory accesses interfering with each other in cache, there is always a dynamic mapping that takes $O(\log m)$ steps per memory access that reduces to zero cache interference.

Before we introduce the main result, we present the terminology used in this section.

Definition 2 *We define the bit-wise operation $y = shift(x, l, k)$ as the following sequence of bitwise operations (in C-language):*

- $t = x\%(1 << l)$ *(i.e. $t = x \mathbf{mod} (2^l)$)*
- $y = ((x - t) << k) + t$

The effect of the operations *shift* is to introduce k-zero bits starting from bit l to $l + k$ into x. We assume that the bits of x, which are lost shifting to left, are all 0.

Definition 3 *We define the bit-wise $y = set(x, l, b)$ as $y = x\&(b << l)$.*

The effect is to set k (i.e. $k = \log b$) bits of x starting from bit l to $l + k$. Now, we state the main result of this section.

Theorem 7 *If there are m vectors, $V_0 \ldots V_{m-1}$ continuously stored with starting address $A_0 < A_1 \cdots < A_{m-1}$ respectively, then it exists an integer function $f()$ so that*

$$y = set(shift(x, l, \log m), l, f(x))$$

with $2^l = L$ is the line size, it is a dynamic mapping, which nullifies interference among the vectors.

Proof: The proof is constructive. We store in a vector of length m the list of the starting address in increasing order. Each pair of contiguous elements in the vector specifies a bucket and therefore one of the input vectors. Given an address x and doing a binary search on the vector, we can check what is the bin the address falls into. This requires up to $\log m$ steps, the position in the vector is a integer $i \in [0, m-1]$. This is an integer function $i = f(x)$. The dynamic mapping address is $y = set(shift(x, l, \log m), l, i)$ where $2^l = L$ is the line size. □

Theorem 7 states that there is a mapping, but it does not consider any of the characteristics of the vector involved in the computation, i.e. sizes and relative addresses. The number of steps required for the computation of $f(x)$ are up to $\log m$ and the computation requires $O(m)$ space. In the following, we shall present stronger premises and, thus, simpler dynamic mapping.

Lemma 2 *If there are m vectors, $V_0 \ldots V_{m-1}$ continuously stored starting from address $A_0 < A_1 \cdots < A_{m-1}$ respectively, so that $A_i - A_{i-1} = k_i 2^c$ with $i \in [1, m-1]$ and $k_0 = 0$, then we can store in a vector of size m the normalized values $\frac{A_i - A_0}{2^c}$ and we can use a binary search to identify where the normalized input address $z = (x - A_0)/2^c$ belongs to,*

and therefore determine a dynamic mapping, which takes $O(\log m)$ integer operations, $O(m)$ space and nullifies interference among the vectors.

We store in a vector of size m - in increasing order - the normalized starting address $(A_0 - A_0)/2^c$, ...,$(A_{m-1} - A_0)/2^c$. Note that we need $m \log(\sum_i k_i)$ space. We take the address x and we normalize it to $z = (x - A_0)/2^c$. Then we use z for a binary search and we determine the index i in the sorted vector. Then we apply the dynamic mapping $y = set(shift(x, l, \log m), l, i)$ where $2^l = L$ is the line size. We still use a binary search and the complexity of the search is $\Theta(\log m)$. In the following, we consider the particular cases when $k = k_i$ is the same for every i, and discover that no binary search is required.

Lemma 3 *If there are m vectors, $V_0 \ldots V_{m-1}$ continuously stored starting from address $A_0 < A_1 \cdots < A_{m-1}$ respectively, so that $A_i - A_{i-1} = k2^c$ with $i \in [1, m-1]$, then $y = set(shift(x, l, \log m), l, f(x))$ with $f(x) = (x - A_0)/k2^c$ is a programmable dynamic mapping that takes $O(1)$ integer operations and $O(1)$ space, which nullifies interference among the vectors.*

For the last case we need a simpler version of the *shift* and *set* instruction, we may use a *swap*.

Definition 4 *A swap function is a function $y = swap_{[\{i_0,i_1,\ldots,i_k\},\{j_0,j_1,\ldots,j_k\}]}(x)$ where x is represented in a binary base as $x_{n-1}, x_{n-2}, \ldots, x_{i_0}, x_{i_k}, \ldots, x_{i_k}, \ldots, x_{j_0}, x_{j_1}, \ldots, x_{j_k}, \ldots, x_0$ and y is $x_{n-1}, x_{n-2}, \ldots, x_{j_0}, x_{j_1} \ldots, x_{j_k}, \ldots, x_{i_0}, x_{i_1}, \ldots, x_{i_k}, \ldots, x_0$.*

While it is arguable the *feasibility* of such function using regular arithmetic, this can be easily implemented by a switch or cross bar by hardware.

Theorem 8 *If there are m vectors $\{V_i\}_{i\in[0,m-1]}$ continuously stored starting from address $A_0 < A_1 \cdots < A_{m-1}$ respectively, so that $A_i - A_{i-1} = 2^k$ with $2^k > C$ and C the cache size,*

then the mapping

$$g(x) = swap_{[\{k+\log m,\ldots,k\},\{\ell+\log m,\ldots,\ell\}]}(x - A_0)) \tag{6.1}$$

nullifies interference, where $2^\ell = L$ *is the line size.*

Proof:

Table 6.3: Vectors range, schematic representation: "*" are wild.

Vector		**bit-31**		$k+\log m$		$k+1$	k	$k-1$			0
V_0	$\rightarrow$	0	...	0	...	0	0	*	*	...	*
V_1	$\rightarrow$	0	...	0	...	0	1	*	*	...	*
V_2	$\rightarrow$	0	...	0	...	1	0	*	*	...	*
...											
V_m	$\rightarrow$	0	...	1	...	1	1	*	*	...	*

The proof is constructive. The beginning of every vector is normalized to the beginning of the first vector (V_0), therefore to a power of two boundary. In Table 6.3, we show that $\log m$ bits are sufficient to distinguish if any of the accesses belong to a particular vector. This can be used to map the address to some particular lines. We follow the approach to associate every access to vector V_0 to line 0 **mod** m, and in general accesses to vector V_i to line i **mod** m.

Indeed, the mapping presented in Equation 6.1, indirectly splits the cache in m groups, and each vector access is going to lines that cannot be shared with any other vector accesses, therefore there is no interference and the theorem follows. □

Theorem 8 is a particular case of Lemma 3 but the function $f()$ is so simplified that no computations are needed at all. We can state the following lemmas.

Lemma 4 *$g(x)$ is bijective and $g^{-1}(y) = swap_{[\{k+\log m,\ldots,k\},\{\ell+\log m,\ldots,\ell\}]}(x) + A_0$*

Lemma 5 *Given a matrix A of size $2^n \times 2^n$ and any tile of matrix A of size $m \times m$, it exists a unique mapping s.t. there is no self interference.*

6.2.2 Hardware Dynamic Mapping

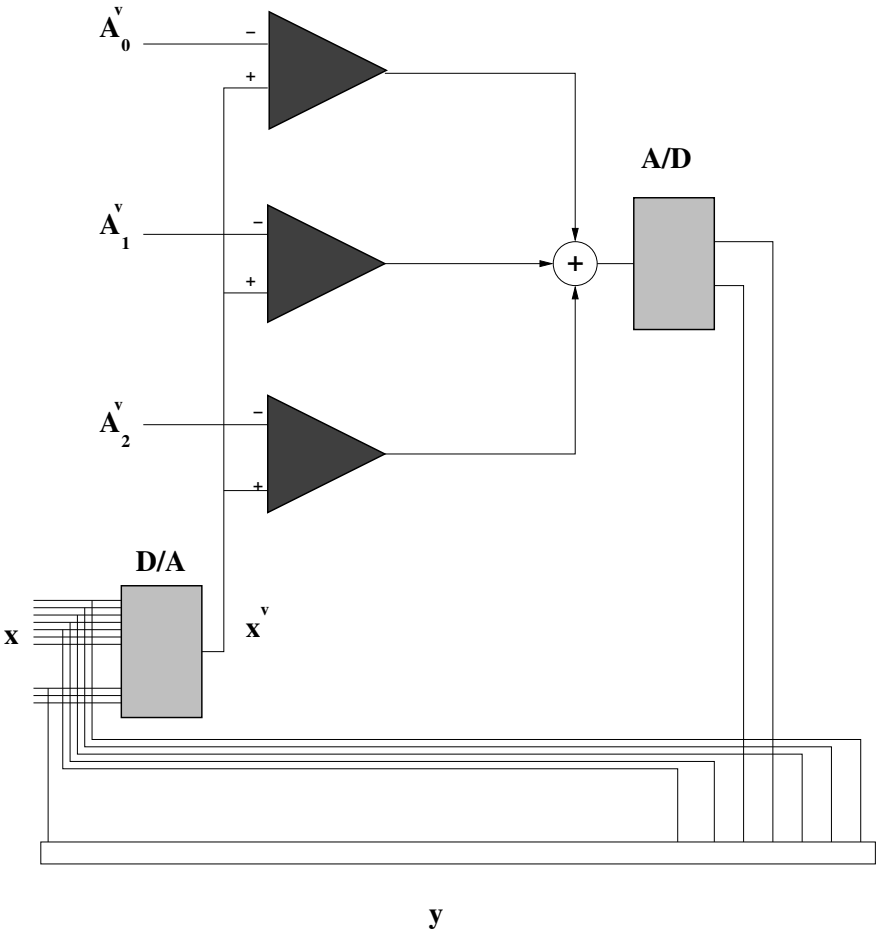

Figure 6.8: Line determination by comparators and decoder.

The main idea in the dynamic mapping is to associate any load on a vector/array with a particular set of lines in the data cache. The association is done by the determination of the bucket in memory from which the load is going to access data. We propose a binary search approach by a comparison of the current load address with the start address of each vector (or the normalized ones). The placement in a bucket of an address can be performed faster when special hardware is deployed. In this short section, we present a solution based on fast components as decoders and comparators. To explain our hardware solution we restrict the number of buckets, alias the number of vectors interfering, to three (however, the approach

can be generalized to any number of vectors).

In Figure 6.8 we can see that the starting address of three vectors have been converted in analog voltage and set as input of one comparator. When a comparator switches off to high voltage (Logic level 1), all the ones above do as well. When a comparator switches off to low voltage (logical zero) all the comparators below do as well. The converter take the outputs of the comparators and return a binary representation between 0 and 2 (with m vectors, $\log m$ bits are enough). The code so determined will be used to determine the cache line.

This implementation takes $O(\log n)$ steps (where n is the number of bits required to represent a line), it requires $O(m \log m)$ space, and it is a mapping as stated in Theorem 7.

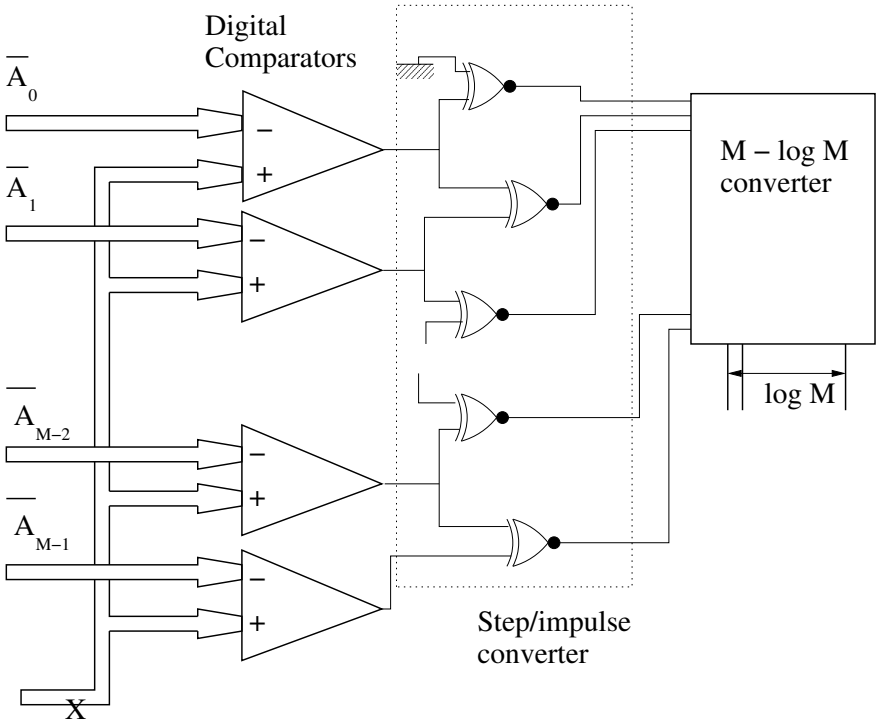

Figure 6.9: Data Cache Line mapping determination using Digital Comparators

The intrinsic *beauty* of the implementation in Figure 6.8 is its simplicity. The bottleneck of the mapping is in the A/D converter (theoretically $O(\log m)$ steps are required). The idea can be implemented using digital hardware as we show in the following, Figure 6.9 and 6.10.

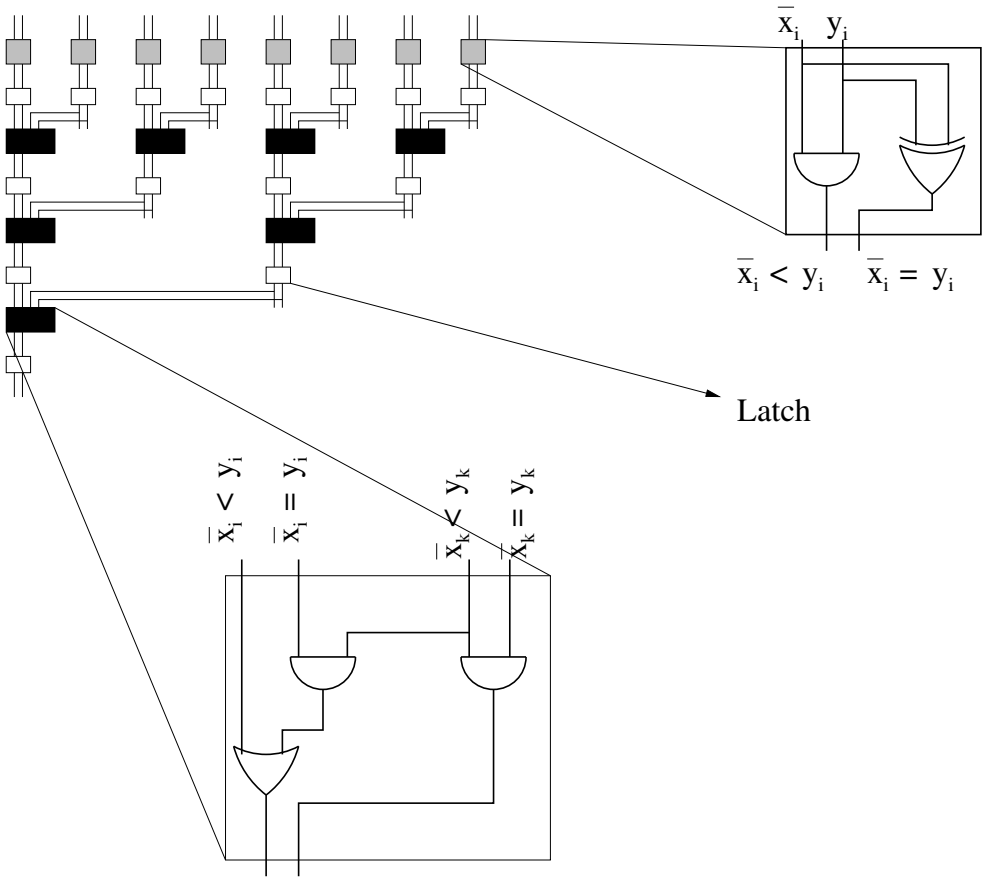

Figure 6.10: Digital comparator implemented as prefix tree

Comparison between two integer numbers can be performed with dedicated hardware. Comparison at first seems simpler than addition. But using logic with max fan in/out of two, the best known algorithm to perform addition of two n bits takes $O(\log_2 n)$ steps, which is the same of any comparison based on the following recursive Equation 6.2, with $x_{up} = x >> (n/2)$, $y_{up} = y >> (n/2)$, $x_r = x - (x_{up} << (n/2))$ and $y_r = y - (y_{up} << (n/2))$:

$$x < y \iff (x_{up} < y_{up}) \textbf{ or } (x_{up} == y_{up} \textbf{ and } x_r < y_r) \tag{6.2}$$

The digital comparator in Figure 6.10 takes $1 + log_2 n$ steps, and each step is composed of at most two level of *simple* logic. It allows a thorough-put on one comparison per cycle. The reference input (i.e. A_i) is bit-wise negated. A digital comparator takes $O(2n)$ space.

When the i-th comparators switches off to 1, i.e. $A_i < x$, then for every $j < i$ $A_j < x$. When the i-th comparators switches off to 0, i.e. $A_i \geq x$, then for every $j > i$ $A_j \geq x$. Therefore the output of the m comparators have only m possible configurations (as supposed to). The comparator outputs determined a step function and the m-EXOR transform the step to an impulse: only one line is set to zero and the others are 0. The encoder is easy to implement and it will not take more than one level of digital logic.

This approach uses M comparators, a step-to-impulse converter and a $m|\log m$ encoder; in fact, this implements a valid mapping with latency $O(\log n)$ and space $O(nm + 2nm + m\log m + 2m)$. Notice that a proper control unit can be deployed to put asleep the part of the circuit not needed reducing energy consumption. Though the structure proposed is complex, however it computes a data cache mapping at any cycle. It also has a moderate latency; for example, a conservative estimation will suggest a latency of up to 7 cycles when $n = 32$, which is independent from the cache deployed.

The work of this parallel architecture is $O(m2n + 2m)$ per access/address. If we consider the algorithm presented in the proof of Theorem 7, the work is $O(n\log m)$ for each address/access. Thus, we can see that the solution in Figure 6.9 is not work optimal. In practice, if we implement the binary search purely in hardware with m comparators, we can improve the work but at the cost of increasing the latency: we can achieve a latency of $O(\log m \log n)$ and work $8n\log m$ per access.

In fact, only two comparators at any time do actual work, one performs a true comparison and the other switches to zero from the previous comparison. The space is $O(m2n\log n)$, thus we need more space due to the latches to store the input address x during the comparison. See Figure 6.11 for an skeleton of the actual implementation.

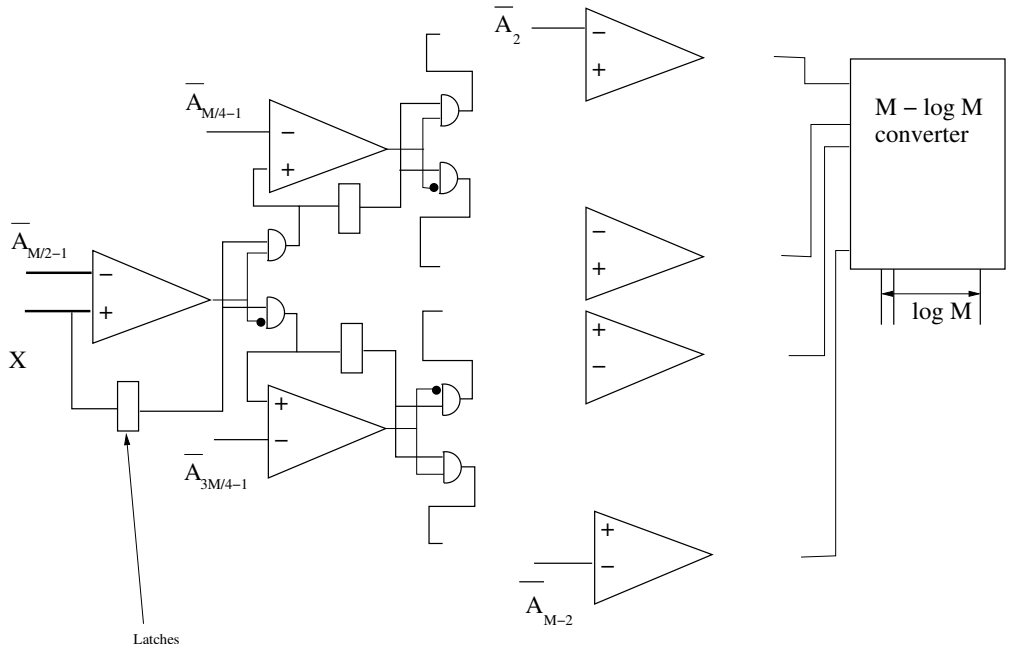

Figure 6.11: Data Cache Line mapping determination using Digital Comparators and binary search

6.3 Dynamic Software Mapping

In this section, we investigate a software-hardware approach to minimize data-cache interferences for perfect loop nest with memory references expressed by affine functions of the loop indexes. This is a common scenario, and other approaches have been presented and powerful analysis techniques can be applied as well.

Briefly this section is organized as follows: in Section 6.3.1, we propose our approach in conjunction with tiling to perfect loop nests, and in Section 6.3.2, we show that the approach can be successfully applied to recursive algorithms as FFT.

6.3.1 Matrix Multiply

In matrix multiply, every memory reference in the loop nest body is determined by an affine function of the loop indexes, for short, **index function** (note: scalar references, if any, are array references with constant index function). An index function determines an address

used to access the memory and the cache at a certain loop iteration. The idea is to compute, in parallel to a regular index function, a **twin function**. The twin function is an affine function of the indexes and it maps a regular address to an alternative address space, or **shadow address**. The index function is used to access the memory; the twin function is used to access the cache. In practice, a compiler can determine the twin functions as a result of an index function analysis and it can tailor the data cache mapping for each load in the inner loop.

Example 6.3.1 *We consider square matrices of size $N \times N$.*

```
for (i=0;i<N;i++)
  for (k=0;k<N;k++)
    for (j=0;j<N;j++)
       C[i][j] += A[i][k]*B[k][j];
```

The reference $A[i][k]$ is a constant in the inner loop and it has index function $A_0 + N * i + 1 * k + 0 * j$. The index function for $C[i][j]$ (respectively, $B[k][j]$) is $C_0 + N * i + 0 * k + 1 * j$ (respectively, $B_0 + 0 * i + N * k + 1 * j$).

Matrix multiply loop nest can be reorganized to exploit temporal locality.

Example 6.3.2 *Let us tile the loop nest by square tiles of size $s \times s$; we assume that N is a multiple of s, and matrices are aligned to the line size, and s is a multiple of the line size L:*

```
for (i=0;i<N/s; i++)
 for (j=0;j<N/s;j++)
  for (k=0;k<N/s;k++)
   for (kk=0;kk<s;kk++)
    for (ii=0;ii<s;ii++)
     for (jj=0;jj<s;jj++)
      C[i*s+ii][j*s+jj] += A[i*s+ii][k*s+kk]*B[k*s+kk][j*s+jj];
```

When $3s^2 < S$, we achieve $\frac{2}{sL}N^3 + \frac{N^2}{L}$ memory accesses (cache misses). Cache misses may be more, due to interference.

When we tile the loop as in Example 6.3.2, the index function can be described concisely by three vectors (or projections onto 1-dimensional space), one for each matrix: $\alpha_1 = [Ns, 0, s, 1, N, 0]$, $\beta_1 = [0, s, Ns, N, 0, 1]$ and $\gamma_1 = [Ns, N, 0, 0, N, 1]$. If we indicate

with $\iota = [i, j, k, kk, ii, jj]^t$ an iteration in the loop nest, the index function for A, B and C are $\alpha_1 * \iota + A_0$, $\beta_1 * \iota + B_0$ and $\gamma_1 * \iota + C_0$.

When matrix $N = 2^n > S$, $S = 2^k$ and all matrices are stored continuously one after the other ($A_0 + N^2 * \ell = B_0$ and $B_0 + N^2 * \ell = C_0$), there is cross interference between references to different tiles (e.g., $\mathbf{C}[i * s + ii][j * s + jj]$ and $\mathbf{B}[k * s + kk][j * s + jj]$) and there is self interference between any two rows in the same tile (e.g., $A[i * s + \mathbf{ii}][k * s + kk]$ and $A[i * s + \mathbf{ii} + \mathbf{1}][k * s + kk]$).

When we tile the loop nest, we tile each matrix as well. Every tile is a square, as the matrix, and it has size $s \times s$. When matrices are aligned to the cache line (i.e.; $A_0 \% L = B_0 \% L = C_0 \% L = 0$) and all tiles are aligned to the cache line (i.e., $s \% L = 0$ and $N \% L = 0$), we can change the data-cache mapping for all memory references safely.

An element in the 6-dimensional space is associated to a **twin element** in a 6-dimensional space. The difference is that we enforce the projection of a twin tile to be a convex space on a 1-dimensional space. The twin tile is stored continuously in memory - but the tile, with which is associated, needs not. Any two twin tiles are spaced at interval of S elements; so different twin tiles will be mapped into the same cache portion.

The twin function uses the following vectors: $\alpha_s = \left[\frac{N}{s}S, 0, S, 1, s, 0\right]$, $\beta_s = \left[0, S, \frac{N}{s}S, s, 0, 1\right]$ and $\gamma_s = \left[\frac{N}{s}S, S, 0, 0, s, 1\right]$. The twin functions for A, B and C are $\alpha_s * \iota$, $\beta_s * \iota + s^2$ and $\gamma_s * \iota + 2 * s^2$.

We consider in details the construction of $\alpha_s = \left[\frac{N}{s}S, 0, S, 1, s, 0\right]$ from $\alpha_1 = [Ns, 0, s, 1, N, 0]$ for matrix A. The components $\alpha_1[0]$ and $\alpha_1[2]$ allow the computation to access different tiles of matrix A: $\alpha_1[0] = Ns$ allows to go from tiles to tiles of size $s^2 = S$ in the same column, and $\alpha_1[2] = s$ allows to go from tiles to tiles in the same row. These two components become $\alpha_s[0] = \frac{N}{s}S$ and $\alpha_s[0] = S$ respectively. The coefficients $\alpha_1[4] = N$ and $\alpha_1[3] = 1$ allow the computation to access elements in a tile: $\alpha_1[4]$ allows to access element in the same column of the tile, and $\alpha_1[3]$ allows to access elements in the same row of the tile - and

stored continuously in memory. They become $\alpha_s[4] = s$ and $\alpha_s[3] = 1$.

The original coefficient that is unitary is left unchanged (e.g., $\alpha_1[3] = 1$ and $\alpha_s[3] = 1$) so a line in memory is a line in cache.

6.3.2 Fast Fourier Transform

A n-point Fourier Transform, ***n-FT***, can be represented as the product of a matrix by a vector: $\mathbf{y} = F_n * \mathbf{x}$ with $\mathbf{x}, \mathbf{y} \in \mathcal{C}^n$ and $F_n \in \mathcal{C}^{n \times n}$. Each component of $\mathbf{y}$ is the following sum: $y_k = \sum_{i=0}^{n-1} x_i \omega_n^{ik}$, where ω_n is called **twiddle factor**.

When n is the product of two factors p and q (i.e., $n = pq$) we can apply Cooley-Tookey's algorithm. The input vectors $\mathbf{x}$ can be seen as $q \times p$ matrix X stored in row major. The output vector $\mathbf{y}$ can be seen as a matrix Y of size $q \times p$ but stored column major. We can write the n-FT algorithm as follows:

1. for every $i \in [0, p-1]$, we compute $X_{[0,q-1],i} = F_q X_{[0,q-1],i}$ –this is a computation on the columns of matrix X;
2. we distribute the twiddle factors, $X_{i,j} = \omega_n^{ij} X_{i,j}$;
3. for every $i \in [0, q-1]$ we compute $X_{i,[0,p-1]} = F_p X_{i,[0,p-1]}$;
4. $Y = X^t$.

Algorithms implementing n-FT on $n = 2^\mu$ points are well known, and attractive, because the designer can reduce the number of computations (twiddle factors reductions). However, they are inefficient when the cache has size $S = 2^k$ due to their intrinsic self interference. Implementations, such as FFTW [12], may exploit temporal locality through copying the input data on a temporary work space. Nonetheless, the spatial locality between the computation of $X_{[0,q-1],i} = F_q X_{[0,q-1],i}$ and $X_{[0,q-1],i+1} = F_q X_{[0,q-1],i+1}$ is not exploited fully; because two elements in the same column of X interfere in cache, preventing the spatial reuse. Even

though the cache interference is responsible for relatively few misses, it effects every level of the memory hierarchy - memory pages too. Any improvement in the number of misses at the first level of cache, even small, is very beneficial for a multilevel cache system.

Sub-problems of n-FT access data in non-convex set, therefore our algorithm cannot be applied as is. We follow a very simple implementation of a recursive algorithm. When we execute the algorithms on the columns of X, the original input matrix X of size $p \times q$ is associated with its twin image in X' of size $p \times (q + \ell)$ where ℓ is the number of matrix elements that can be stored in a cache line. If $q\%\ell \neq 0$, elements in the last and first column of X share the same cache line. If the input vector does not fit the cache, the first and the last column of X are not accessed at the same time; the cache coherence is unaffected.

6.3.3 Architecture

To describe the effects of the dynamic mapping approach on the architecture design we use the block structure of MIPS R10K microprocessor, Figure 6.12. We use MIPS but these modifications can be applied to other processors like SPARC64 processors as well. SPARC64 has a large integer register file (RF), 32 registers directly addressable but a total of 56 for register renaming; it is a true 64 bits architecture, and it has only one level of cache.

The twin function is performed in parallel with the regular index function. The computations share the same resources, integer units and register file. The twin function result will be stored in the register file, but it is not really an address. The address calculation unit (ACU) and TLB do not process the twin function result. The twin functions need not to be valid addresses in memory at all.

The architecture is as follows. The instruction set is augmented with a new *load* instruction with three operands, or registers: the destination register, the index function register and the twin function register. The load instruction becomes like any other instructions, with two source operands and one destination. To improve performance, a load can be is-

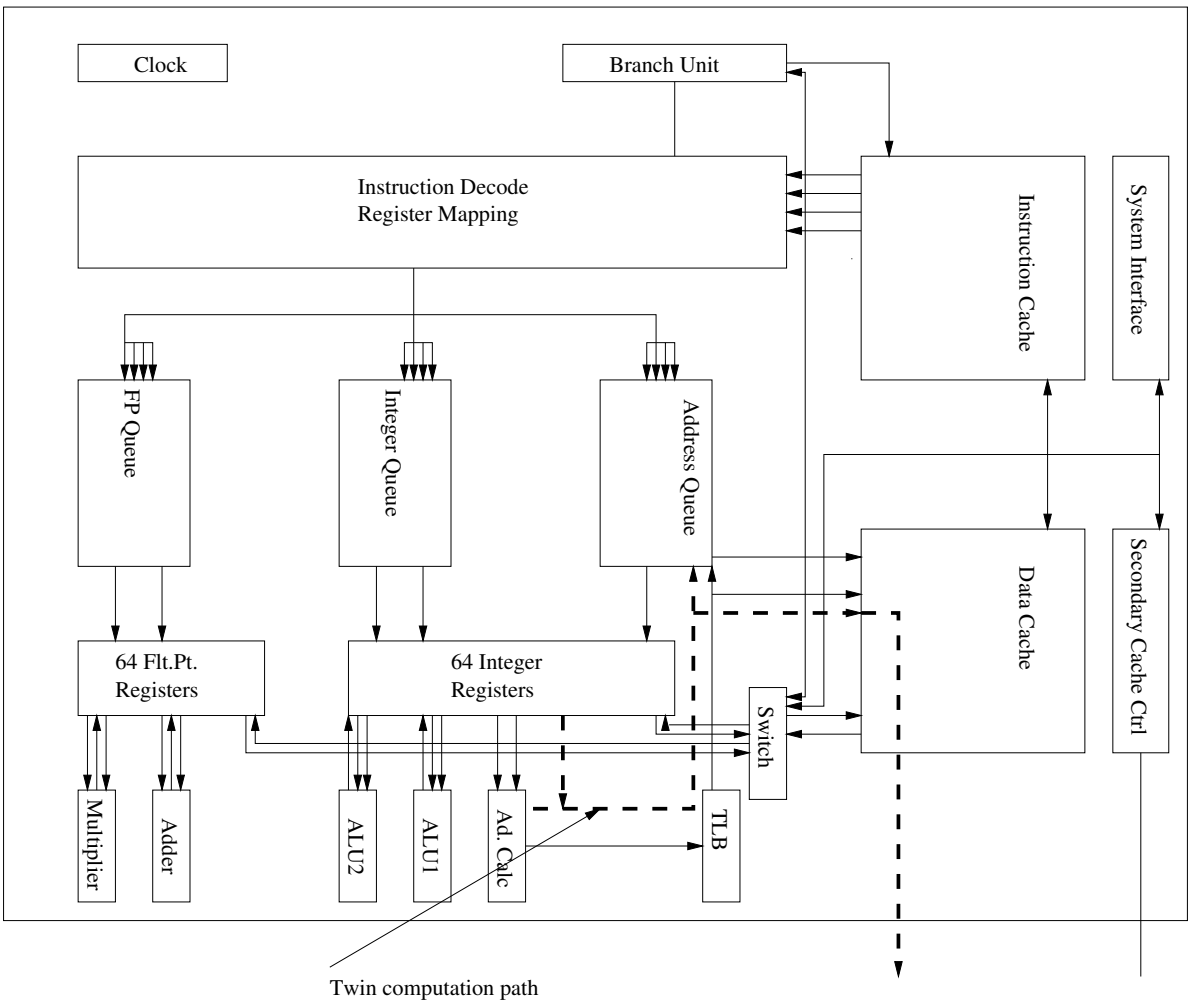

Figure 6.12: Proposed architecture, designed based on MIPS R12K

sued as soon as the twin index function is computed to speed up the access in cache. The twin function can be used directly by the cache without further manipulation. The regular index function is really necessary in case of a miss and ACU and TLB must process it. We can imagine that a possible implementation may decide to execute the regular index function only when a miss happens. We can see there is potential to change cache mapping without increasing the hit latency time in cache. The design remains simple: the functional units communicate with the register files only and the register file with the first level of cache only. The design can be applied even when a cache is multi-ported; that is, when multiple loads and writes can be issued to cache.

The proposed approach increases integer register pressure and it issues more integer instructions because of the index computations. The compiler may introduce register spills on the stack, but in general, they are not misses in cache (having temporal and spatial locality). Index function and twin index function are independent and we can issue them in parallel. If the number of pipelined ALUs does not suffice the parallelism available, the index function can lead to a slow down. The slow-down factor is independent from the number of index functions and it is no larger than 2. (The slow-down factor and the total work can be reduced issuing the index function only in case of a miss; we increase only the cache miss latency.)

To avoid consistency problems the data cache is virtually indexed and physically tagged. The new type of load does not affect how many load instructions can be issued or executed per cycle. We assume that an additional RF output port has a negligible effect on the register file access time. (Otherwise, twin functions can be processed in parallel with the index functions and dedicated RF can be used.)

6.3.4 Experimental Results

Dynamic Mapping is applied to two applications, Matrix Multiply (Section 6.3.4, Example 6.3.2) and FFT (Section 6.3.4). We show the potential performance on 5 different systems. Indeed, the algorithms are performed only using the twin function (no regular index function is performed).

Matrix Multiply

We implemented the matrix multiply as described in Example 6.3.2 in Section 6.3.1. The application has spatial and temporal locality. Our goal is to show the cache improvements due to dynamic mapping, Figure 6.13. We measure the cache performance of matrix multiplication with either standard index function or dynamic mapping –not both. The two algorithms have the same number of operations, memory accesses and most probably the

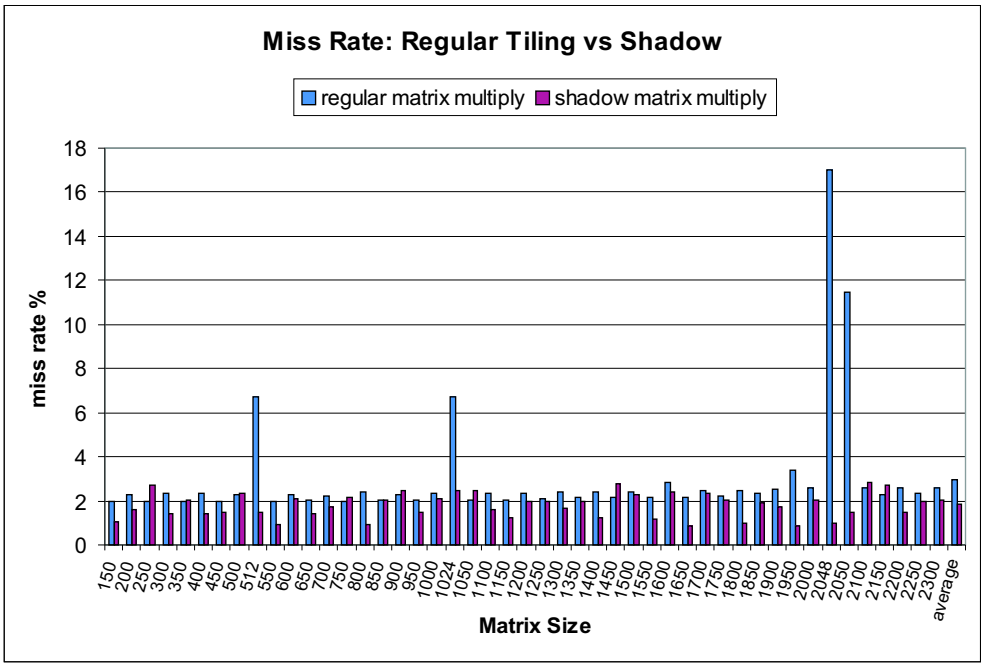

Figure 6.13: Matrix Multiply on Blade 100, miss rate comparison.

same instruction schedule. The only difference is the access pattern. The dynamic mapping allows a stable miss ratio across different input size (2% which is close to the expected, roughly $\frac{1}{\ell\sqrt{C/3}} = \frac{\sqrt{3}}{2\sqrt{2048}}$) removing cache miss spikes altogether.

We always improve cache performance. We can see a cache miss reduction of 30% in average and up to 8 times cache miss reduction for power of two matrices.

In this section, we do not take into account the effects of register allocation, which can improve cache and overall performance. An optimal register allocation, for both tiled and recursive algorithms, can reduce the number of memory accesses and exploits a better data reuse at register level. In general, register allocation is not designed to reduce cache interference and it is machine dependent - different libraries use different register allocations, see for examples of register allocation for matrix multiply kernel [22, 103].

n-FFT

We implemented our variation of Cooley-Tookey algorithm as follows.

Cooley-Tookey algorithm finds a factorization of n in two factors so that $n = p*q$. When two factors are found, the problem is decomposed in two sub-problems. We can represent this divide and conquer algorithm by a binary tree. An internal node represents a problem of size m and its children the factors of m - and the two subproblems. The leaves of the binary tree are the **codelets** from FFTW [12] - codelets are small FFTs of size between 2 and 16 written as a sequence of straight-line code. The binary tree can have a hight between 1 and $O(\log_2 n)$. The tree hight is 1 when n is prime and we need to perform $O(n^2)$ operations. In case one of the two factors is always between 2 and 32, the tree hight is at most $O(\log_2 n)$ - when $n = 2^k$ - and we need to perform $O(n \log_2 n)$ operations.

Our algorithm aims at a balanced decomposition of n in two factors, such as $n = p * q$ and $p \sim q$, that is, the difference $|p - q|$ is as small as possible. The binary tree can have a hight at most $O(\log_2 \log_2 n)$ –when $n = 2^{2^k}$– however, we need to execute always $O(n \log_2 n)$ operations.

We show the performance of our FFT with and without dynamic mapping in Figure 6.14 and 6.15. The bars represent normalized MFLOPS: given an input of size n, we show $n * \log n/(10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT. Since our implementation does not have the same number of floating point instructions of FFTW, we opted to measure the execution time and determine a normalized number of FLOPS –even when n is prime. Every complex point is composed of two float point numbers of 4 byte each (we want to have spatial locality for 16 B cache line size).

We choose to show performance in MFLOPS instead of data-cache miss rate for two reasons. First, the reduction of cache misses using dynamic mapping is small, but the performance improvement is extremely significant. Second, the implementation is one for all architecture, therefore we may compare performance across architectures.

We collected experimental results for four algorithms: we identify with **our FFT**, our implementation of Cooley-Tookey algorithm; **Dynamic Mapping** is **our FFT** when dynamic

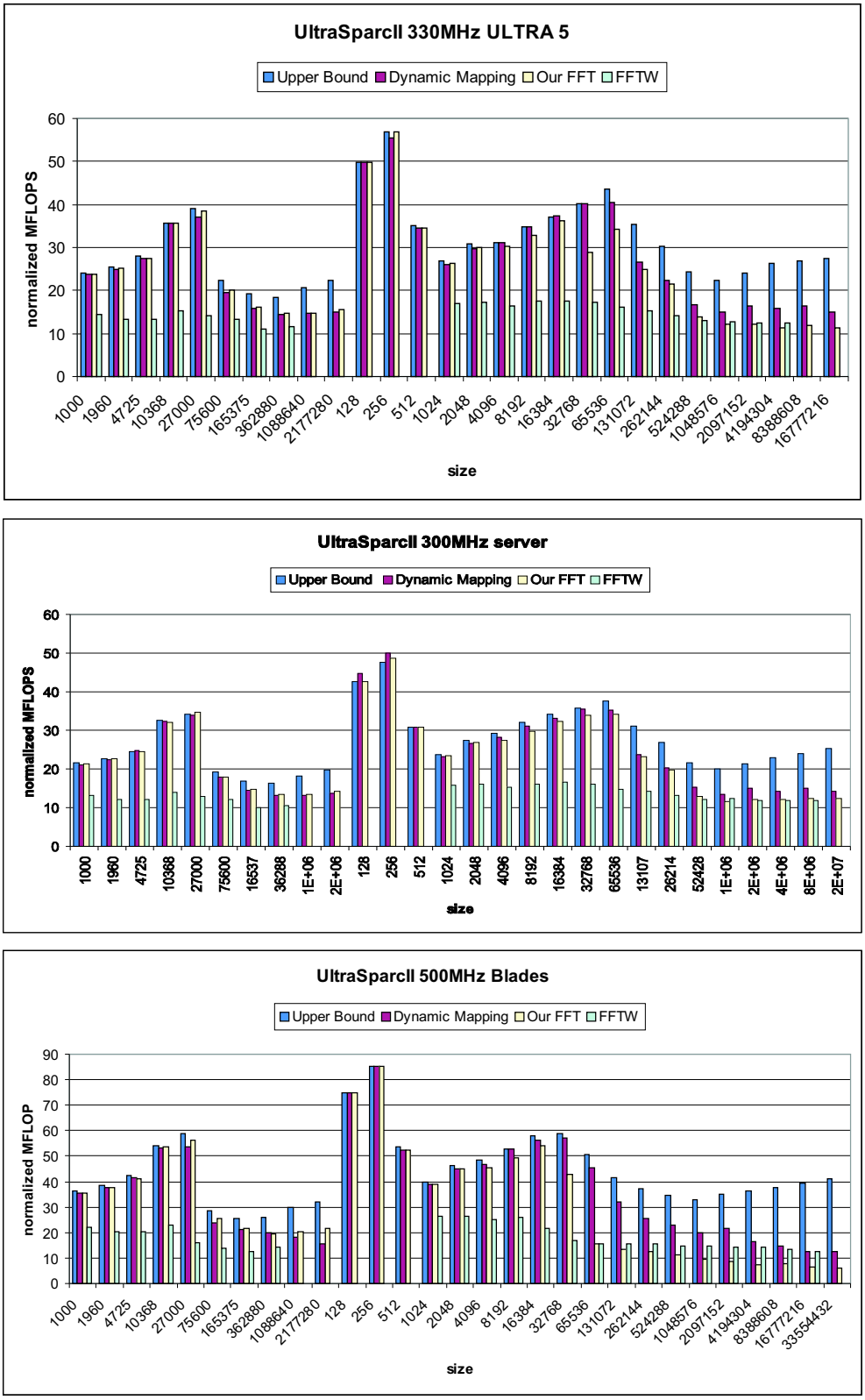

Figure 6.14: FFT Ultra 5 (330MHz), Enterprise 250 (300MHz) and Blade 100 (500MHz). Normalized performance: $n * \log n/(10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT. The bars from left to right:**Upper Bound**, **Dynamic Mapping**,**our FFT** and **FFTW**

mapping is applied; we identify with **Upper Bound** a recursive algorithm for FFT that accesses the cache with an ideal pattern (but invalid). When present, **FFTW** is the Fastest Fourier Transform in the West [12]. FFTW is used as reference to understand the relation between the performance of our implementation, the potential performance of dynamic mapping and the performance of a well known FFT. Dynamic mapping could be applied to the FFTW as well, and the improvements would be proportional to the ones shown in this section.

In Figure 6.14, we show that our implementation is efficient for large power-of-two inputs, but also, it has no steady performance - as FFTW. The performance is a function of the input size. Indeed, the performance is a function of the input decomposition. Large leaves allow fewer computations, therefore better performance. Since the decomposition does not assure that all leaves have the best size, this behavior is expected.

We can notice that for small n, the performance of our implementations may be faster than expected (or slower). This is due to several reasons. One of them is the accuracy problem: the execution of other processes effect the execution-time measure under investigation (e.g., caches, register file, ALU and FPU pipeline trashing). For fairly large to very large problems, where the memory hierarchy is utilized intensely, dynamic mapping lays between its upper bound and our implementation of FFT, as expected.

The characterization of the worst and best performance is twofold: first, it shows the performance we can achieve, the performance we can achieve with dynamic mapping and ideal performance if cache locality is fully exploited; second, when there is no reference for execution time, we are still able to have an estimation of execution time and potential performance.

FFT is an excellent example of application with a high self interference. Even caches with a large cache associativity cannot cope with the loss of performance due to interference for large number of points. In Figure 6.15, we present performance for two systems with

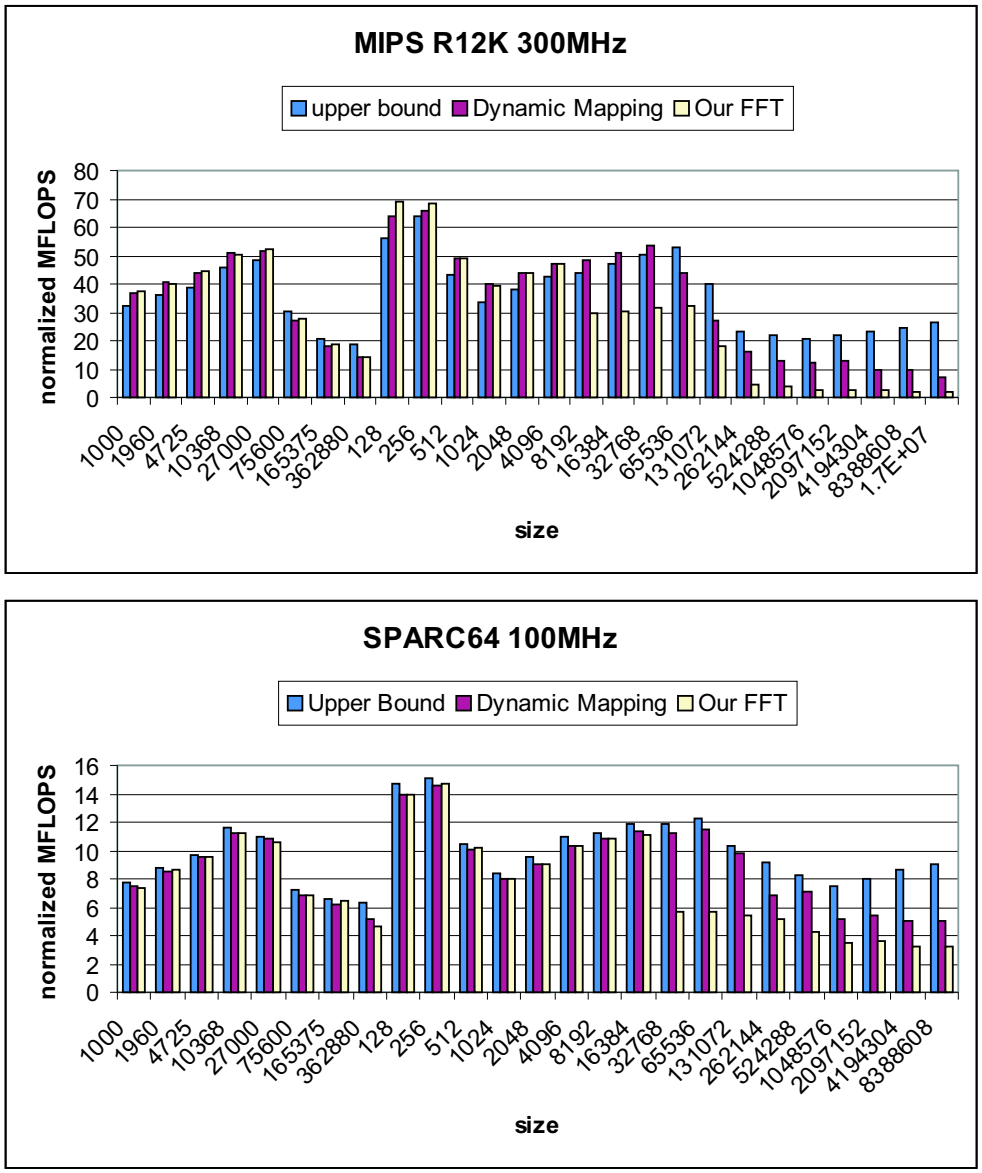

Figure 6.15: FFT Silicon Graphics O2 and Fujitsu HAL 300 Normalized performance: $n * \log n/(10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT. The higher the bar the better the performance. The bars from left to right:**Upper Bound**, **Dynamic Mapping**,**our FFT**

associative caches: SGI O2 and Fujitsu HAL 300.

CHAPTER 7

Conclusions and Future Work

For a student, a thesis is often considered as the presentation of years of research and a major component for the completion of a degree, in this case for a Doctorate in Philosophy. In fact, a thesis is often considered as the final documentation of the successes (and failures) associated with an original work (of a student). However, the organization and production of this thesis has been for me a concrete backbone to drive my curiosity and my research and it has not been just the last push to achieve a degree. As such, being a research backbone, this thesis is self contained and it introduces few novel ideas and approaches, and, thus, their effects; however, for the same reason, this thesis is incomplete and it leaves (unwillingly) open some questions and applications that I shall answer and apply after this work has been submitted and retired. In this last section, we shall summarize the main contributions of this thesis and also draw our conclusions and possible avenues for future work.

The novel contribution of this thesis is the presentation and practice of two basic ideas for the analysis and optimization of recursive D&C algorithms: first, the recursion-DAG as computation model for recursive D&C algorithms; and second, the practical and symbolic data-cache miss analysis for parameterized loop nests.

In fact, the recursion-DAG is a concise structure that can be used as profiling tool for the unfolding of recursive algorithms so to apply high-level optimizations such as function specialization and function cloning for applications that previously could not be optimized

at such level. Though optimizations at compile time are the main topic of this thesis, the recursion-DAG could be used for synthesis purpose and it could extend the capability of current synthesis tools to recursive functions as well. We present the practical implementation of this idea introducing JuliusC.

The cache-miss static analysis based on the application of the Cache Miss Equation (CME) Model introduces new ideas for the solution of a difficult problem such as the cache analysis of loop nests with unknown loop bounds at compile time. The practical application of data cache miss estimation opens new avenues to a compiler for the application of known and new optimizations. In fact, such an analysis allows the compiler to estimate statically whether or not specific optimizations are beneficial previously achievable only by profiling techniques. We have found and presented three different optimizations such as data-cache line size adaptation, dynamic mapping and spatial reuse. These optimizations can be driven by a careful data cache analysis and they aim at either the annihilation of cache interference or the reduction of the effects of cache interference on the application performance.

The recursion-DAG and the cache miss analysis have a natural application for the design and optimization of parallel linear algebra applications. In fact, the recursion-DAG can be used as intermediate representation of the application execution and, with data dependency analysis, it can be used for the spawning of threads and processes so to exploit parallelism and high level of granularity. In practice, the spawning should be driven by the problem size (foot print), which, in turn, can be annotated into the recursion-DAG in advance and statically.

Then, the spawned thread should be optimized so as to exploit the processor characteristics and, especially, the memory hierarchy available at processor level. In fact, a process is spawned so as to have the right problem size for the processor and, thus, an optimal utilization of the processor resources will translate in high and predictable performance of the locally running process and of the entire parallel application.

As an intermediate step for the application of our techniques to parallel application, we would like to investigate how a compiler or an expert developer would choose the division strategy (of a D&C algorithm) to exploit parallelism and performance. For example, matrix multiply has different D&C algorithms such as the classical algorithm introduced in Section 4.2 or Strassen's algorithms. These two algorithms have very different number of floating point computations, the classic algorithm has $O(n^3)$ `madd` operations and, in contrast, Strassen's has only $O(n^{2.86})$. To a closer investigation, the number of basic operation is not the only difference between these two algorithms. In practice, the two algorithms exploit different data locality; for example, the classic algorithm has and presents temporal and spatial data locality; in contrast, Strassen's algorithm has little temporal data locality and mostly spatial locality (because Strassen's requires the application of matrix additions, which have spatial locality only).

The determination of a recursive algorithm guided by a quantitative and concise measure of the performance of both algorithms is a basic milestone towards a quantitative evaluation of the characteristics of the algorithms. This can be used so as the recursive algorithm, without any further directives by the developer, can choose the optimal algorithm for a specific system and input problem. Due to the importance of matrix multiply, this approach should be generalizable to families of applications such as linear algebra routines, where matrix multiplication is a basic kernel. Moreover, this approach can be used by a parallel algorithm as well.

The design of parallel applications for multiprocessor systems is complicated by the interconnection among every processor. In fact, the codes should be implemented keeping in mind the processor interconnections – which is not easy. In practice, the implications of such an adaptive-code design for a parallel application is a work in progress, however it is receiving plenty of attention for uniprocessor systems.

In fact, the processor capabilities and its configurations/variations are evolving as such a

high speed that the software is becoming the **slow-changing component** of a system and often the most expensive. Adaptive codes, such as the one using a mix approach of Strassen's and the classic algorithm for matrix multiply, are receiving more and more attention by the research community because they bring back the software to be the *malleable* and *portable* component of a system. A system where both components aim at achieving the best possible performance.

BIBLIOGRAPHY

[1] J. Dongarra, I. Duff, D. Soransen, and H. van Der Vorst, *Numerical Linear Algebra for Performance Computers*. SIAM, 2000.

[2] G. Golub and C. van Loan, *Matrix Computations*. Ed. The Johns Hopins University Press, 1996.

[3] J. Vitter and E. Shriver, "Algorithms for parallel memory I: Two-level memories," *Algorithmica*, vol. 12, no. 2/3, pp. 110–147, Aug - Sep 1994.

[4] ——, "Algorithms for parallel memory II: Hierarchical multilevel memories," *Algorithmica*, vol. 12, no. 2/3, pp. 148–169, Aug - Sep 1994.

[5] S. Toledo, "Locality of reference in LU-decomposition with partial pivoting," *SIAM J. Matrix Anal. Appl*, vol. 18, no. 4, pp. 1065–1081, Oct 1997.

[6] J. Frens and D. Wise, "Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code," *Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming*, vol. 32, no. 7, pp. 206–216, July 1997.

[7] E. Anderson, Z. Bai, C. Bischof, J. D. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D.Sorensen, *LAPACK User' Guide, Release 2.0*, 2nd ed. SIAM, 1995.

[8] R. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 1998, pp. 1–27.

[9] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," in *Proceedings of the 19-th annual ACM conference on Theory of computing*, 1987, pp. 1–6.

[10] V. Strassen, "Gaussian elimination is not optimal." *Numerische Mathematik*, vol. 14, no. 3, pp. 354–356, 1969.

[11] J. Hong and T. Kung, "I/O complexity, the red-blue pebble game," in *Proceedings of the 13th Ann. ACM Symposium on Theory of Computing*, Oct 1981, pp. 326–333.

[12] M. Frigo and S. Johnson, "The fastest fourier transform in the west," Massachusetts Institute of technology, Tech. Rep. MIT-LCS-TR-728, Sep 1997.

[13] A. Aggarwal, A. Chandra, and M. Snir, "Hierarchical memory with block transfer," in *In 28th Annual Symposium on Foundations of Computer Science*, Los Angeles, California, October 1987, pp. 204–216.

[14] S. Chatterjee, V. Jain, A. Lebeck, and S. Mundhra, "Nonlinear array layouts for hierarchical memory systems," in *Proc. of ACM international Conference on Supercomputing*, Rhodes,Greece, June 1999.

[15] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi, "Recursive array layout and fast parallel matrix multiplication," in *Proc. 11-th ACM SIGPLAN*, June 1999.

[16] P. Flajolet, G. Gonnet, C. Puech, and J. Robson, "The analysis of multidimensional searching in quad-trees," in *Proceeding of the second Annual ACM-SIAM symposium on Discrete Algorithms*, San Francisco, 1991, pp. 100–109.

[17] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau, "Augmenting loop tiling with data alignment for improved cache performance," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 142–9, Feb 1999.

[18] B. Kagstrom, P. Ling, and C. van Loan, "GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark," *ACM Transactions on Mathematical Software*, vol. 24, no. 3, pp. 268–302, Sept 1998.

[19] J. Park, M. Penner, and V. Prasanna, "Optimizing graph algorithms for improved cache performance," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.

[20] I. Jonsson and B. Kagström, "Recursive blocked algorithms for solving triangular systems part I: one-sided and coupled sylvester-type matrix equations," *ACM Trans. Math. Softw.*, vol. 28, no. 4, pp. 392–415, 2002. [Online]. Available: http://doi.acm.org/10.1145/592843.592845

[21] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache oblivious algorithms," in *Proceedings 40th Annual Symposium on Foundations of Computer Science*, 1999.

[22] G. Bilardi, P. D'Alberto, and A. Nicolau, "Fractal matrix multiplication: a case study on portability of cache performance," in *Workshop on Algorithm Engineering 2001*, Aarhus, Denmark, 2001.

[23] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, Ansi C coding methodology," in *International Conference on Supercomputing*, July 1997.

[24] M. Lam, E. Rothberg, and M. Wolfe, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the fourth international conference on architectural support for programming languages and operating system*, Apr 1991, pp. 63–74.

[25] F. Gustavson, A. Henriksson, I. Jonsson, P. Ling, and B. Kagström, "Recursive blocked data formats and BLAS's for dense linear algebra algorithms." in *PARA'98 Proceedings. Lecture Notes in Computing Science, No. 1541*, S. Verlag, Ed., 1998, pp. 195–206.

[26] P. D'Alberto and A.Nicolau, "JuliusC: A practical approach for the analysis of divide-and-conquer algorithms," in *(to be published) Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science. Springer Verlag, Sept 2005.

[27] P. D'Alberto, A.Nicolau, and A. Veidenbaum, "A data cache with dynamic mapping," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, L. Rauchwerger, Ed., vol. 2958. Springer Verlag, Oct 2003.

[28] J. Hummel, L. Hendren, and A. Nicolau, "Abstract description of pointer data structures: an approach for improving the analysis and optimization of imperative programs," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 3, pp. 243–260, 1992. [Online]. Available: http://doi.acm.org/10.1145/151640.151644

[29] R. Rugina and M. Rinard, "Automatic parallelization of divide and conquer algorithms," in *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, 1999, pp. 72–83. [Online]. Available: http://doi.acm.org/10.1145/301104.301111

[30] E. Albert, M. Hanus, and G. Vidal, "Using an abstract representation to specialize functional logic programs," in *Proc. of 7th International Conference on Logic for Programming and Automated Reasoning, LPAR'2000*. Springer LNAI 1955, 2000, pp. 381–398.

[31] C. Gomard, "A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics," *ACM Trans. Program. Lang. Syst.*, vol. 14, no. 2, pp. 147–172, 1992. [Online]. Available: http://doi.acm.org/10.1145/128861.128864

[32] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, June 1993. [Online]. Available: http://www.dina.dk/~sestoft/pebook/

[33] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation.* ACM Press, 1994, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/178243.178256

[34] W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press, 1989, pp. 315–328. [Online]. Available: http://doi.acm.org/10.1145/75277.75305

[35] W. Pugh, "An improved replacement strategy for function caching," in *Proceedings of the 1988 ACM conference on LISP and functional programming.* ACM Press, 1988, pp. 269–276. [Online]. Available: http://doi.acm.org/10.1145/62678.62719

[36] A. Heydon, R. Levin, and Y. Yu, "Caching function calls using precise dependencies," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation.* ACM Press, 2000, pp. 311–320. [Online]. Available: http://doi.acm.org/10.1145/349299.349341

[37] M. Abadi, B. Lampson, and J. Lévy, "Analysis and caching of dependencies," in *Proceedings of the first ACM SIGPLAN international conference on Functional*

programming. ACM Press, 1996, pp. 83–91. [Online]. Available: http://doi.acm.org/10.1145/232627.232638

[38] Y. Liu and S. Stoller, "Dynamic programming via static incrementalization," *Higher Order Symbol. Comput.*, vol. 16, no. 1-2, pp. 37–62, 2003.

[39] ——, "From recursion to iteration: What are the optimizations?" in *Partial Evaluation and Semantic-Based Program Manipulation*, 2000, pp. 73–82. [Online]. Available: citeseer.ist.psu.edu/liu00from.html

[40] Q. Yi, V. Adve, and K. Kennedy, "Transforming loops to recursion for multi-level memory hierarchies," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM Press, 2000, pp. 169–181. [Online]. Available: http://doi.acm.org/10.1145/349299.349323

[41] C. Meyer, *Matrix Analysis and Applied Linear Algebra*. SIAM, 2000.

[42] N. Higham, *Accuracy and stability of numerical algorithms*. SIAM, 1996.

[43] J. Hennesy and D. Patterson, *Computer architecture a quantitative approach*. Morgan Kaufman 2-nd edition, 1996.

[44] N. Eiron, M. Rodeh, and I. Steinwarts, "Matrix multiplication: a case study of algorithm engineering," in *Proceedings WAE'98*, Saarbrücken, Germany, Aug 1998.

[45] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir, "A model for hierarchical memory," in *Proceedings of 19th Annual ACM Symposium on the Theory of Computing*, New York, 1987, pp. 305–314.

[46] J. Savage, "Extending the Hong-Kung model to memory hierachies," in *In Proc. of the 1st Ann. International Conference on Computing and Combinatorics*, ser. Lecture Notes in Computer Science, S. Verlag, Ed., vol. 959, 1995, pp. 270–281.

[47] B. Alpern, L. Carter, E. Feig, and T. Selker, "The uniform memory hierarchy model of computation." *Algorithmica*, vol. 12, pp. 72–129, 1994.

[48] G. Bilardi and F. Preparata, "Processor-time tradeoffs under bounded-speed message propagation. part ii: lower bounds." *Theory of Computing Systems*, vol. 32, no. 5, pp. 531–559, November/December 1999.

[49] G. Bilardi, A. Pietracaprina, and P. D'Alberto, "On the space and access complexity of computation dags," in *26th Workshop on Graph-Theoretic Concepts in Computer Science*, Konstanz, Germany, June 2000.

[50] G. Bilardi and E. Peserico, "A characterization of temporal locality and its portability across memory hierarchies," in *International Colloquium on Automata, Languages, and Programming ICALP 2001*, Crete, July 2001.

[51] B. Kagstrom, P. Ling, and C. van Loan, "Algorithm 784: GEMM-based level 3 BLAS: portability and optimization issues," *ACM Transactions on Mathematical Software*, vol. 24, no. 3, pp. 303–316, Sept 1998.

[52] M. Dayde and I. Duff, "A blocked implementation of level 3 BLAS for RISC processors," CERFACS, Tech. Rep. TR_PA_96_062, 1996. [Online]. Available: http://www.cerfacs.fr/algor/reports/TR_PA_96_06.ps.gz

[53] IBM, "Engineering and scientific subroutine library." 2004. [Online]. Available: http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html

[54] M. Wolfe, "More iteration space tiling," in *Proceedings of Supercomputing*, Nov 1989, pp. 655–665.

[55] M. Wolfe and M. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN'91 conference on programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991.

[56] S. Carr and K. Kennedy, "Compiler blockability of numerical algorithms," in *Proceedings of Supercomputing*, Nov 1992, pp. 114–124.

[57] U. Banerjee, *Loop Transformations for Restructuring Compilers The Foundations*. Kluwer Academic Publishers, 1993.

[58] E. Granston, W. Jalby, and O. Teman, "To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts," in *Proceedings of Supercomputing*, Nov 1993, pp. 410–419.

[59] M. Wolfe, *High performance compilers for parallel computing*. Addison-Wesley, 1995.

[60] F. Gustavson, "Recursion leads to automatic variable blocking for dense linear algebra algorithms," *Journal of Research and Development*, vol. 41, no. 6, November 1997.

[61] M. Thottethodi, S. Chatterjee, and A. Lebeck, "Tuning Strassen's matrix multiplication for memory efficiency." in *Proc. Supercomputing*, Orlando,FL, nov 1998.

[62] D. Wise, "Undulant-block elimination and integer-preserving matrix inversion," Science Department Indiana University, Tech. Rep. Technical Report 418 Computer, August 1995.

[63] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[64] J. Ullman and M. Yannakakis, "The input/output complexity of transitive closure," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, vol. 19, May 1990.

[65] U. Zwick, "All pairs shortest paths using bridging sets and rectangular matrix multiplication," *Journal of the ACM*, vol. 49, no. 3, pp. 289–317, 2002. [Online]. Available: http://doi.acm.org/10.1145/567112.567114

[66] R. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, 1962.

[67] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM*, vol. 9, no. 1, 1962.

[68] U. Zwick, "Exact and approximate distances in graphs - a survey," in *Proceedings of the 9th Annual European Symposium on Algorithms*. Springer-Verlag, 2001, pp. 33–48.

[69] E. Dijkstra, "A note on two problems in connection with graphs," *Numerische Math*, no. 1, pp. 269–271, 1959.

[70] M. Penner and V. Prasanna, "Cache-friendly implementations of transitive closure," in *Proeedings of International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[71] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: theory and experimental evaluation," in *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1994, pp. 516–525.

[72] C. Demetrescu and G. Italiano, "A new approach to dynamic all pairs shortest paths," in *Proceedings of the thirty-fifth ACM symposium on Theory of computing*. ACM Press, 2003, pp. 159–166. [Online]. Available: http://doi.acm.org/10.1145/780542.780567

[73] C. Demetrescu, S. Emiliozzi, and G. Italiano, "Experimental analysis of dynamic all pairs shortest path algorithms," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2004, pp. 369–378.

[74] M. Kamble and K. Ghose, "Analytical energy dissipation models for low-power caches," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1997, pp. 143 – 148.

[75] K. Ghose and M. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation," in *Proceedings 1999 International Symposium on Low Power Electronics and Design*, 1999, pp. 70 – 75.

[76] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adaptive cache line size to application behavior," in *Proceedings of International Conference on Supercomputing (ICS)*, June 1999, pp. 145–154.

[77] P. van Vleet, E. Anderson, L. Brown, J. Baer, and A. Karlin, "Pursuing the performance potential of dynamic cache line sizes," in *Proceedings of the international conference on computer design (ICCS'99)*, 1999.

[78] X. Ji, D. Nicolaescu, A. Veidenbaum, A. Nicolau, and R. Gupta, "Compiler-directed cache assist adaptivity," University of California, Irvine, ICS, Tech. Rep. 00 17, May 2000.

[79] 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*, vol. 21, no. 4, pp. 703–746, July 1999.

[80] ——, "Automated cache optimizations using CME driven diagnosis," in *Proceedings of the international conference on supercomputing*, 2000, pp. 316–326. [Online]. Available: citeseer.nj.nec.com/ghosh00automated.html

[81] X. Vera, N. Bermudo, J. Llosa, and A. González, "A fast and accurate framework to analyze and optimize cache memory behavior," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 2, pp. 263–300, 2004, previously appeared in EUROPAR. [Online]. Available: http://doi.acm.org/10.1145/973097.973099

[82] X. Vera and J. Xue, "Let's study whole-program cache behaviour analytically," in *8th International Symposium on High Performance Computer Architecture (HPCA'02)*, 2002, pp. 176–185. [Online]. Available: citeseer.ist.psu.edu/article/vera02lets.html

[83] K. McKinley and O. Temam, "A quantitative analysis of loop nest locality," in *APLOS VII*, MA, USA, Oct 1996.

[84] K. McKinley and K.Kennedy, "Optimizing for parallelism and data locality," in *The International Conference on Supercomputing*, 1992, pp. 323–333.

[85] F. Catthoor, N. Dutt, and C. E. Kozyrakis, "How to solve the current memory access and data transfer bottlenecks: At the processor architecture or at the compiler level?" in *DATE*, march 2000.

[86] R. Gupta, "Architectural adaptation in AMRM machines," in *Proceedings of IEEE Computer Society Workshop on VLSI 2000*, Los Alamitos, CA, USA, 2000, pp. 75–79.

[87] M. Cabeza, M. Clemente, and M. Rubio, "Cachesim: a cache simulator for teaching memory hierarchy behavior," in *Proceedings of the 4th annual Sigcse/Sigue on Innovation and Technology in Computer Science education*, 1999, p. 181.

[88] K. Gatlin and L. Carter, "Memory hierarchy considerations for fast transpose and bit-reversals," in *HPCA*, 1999, pp. 33–. [Online]. Available: citeseer.nj.nec.com/gatlin99memory.html

[89] Y. Paek, J. Choi, J. Joung, J. Lee, and S. Kim, "Exploiting parallelism in memory operations for code optimization," in *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, Sept. 2004. [Online]. Available: http://www.ecn.purdue.edu/lcpc2004

[90] L. Zhang, Z. Fang, M. Parker, B. Mathew, L. Schaelicke, J. Carter, W. Hsieh, and S. McKee, "The impulse memory controller," *IEEE Transactions on Computers, Special Issue on Advances in High Performance Memory Systems*, pp. 1117–1132, November 2001.

[91] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *In the Proceedings of the Fifth International Symposium on High Performance Computer Architecture (HPCA-5)*, January 1999, pp. 70–79.

[92] T. Johnson and W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[93] ——, "Run-time adaptive cache hierarchy management via reference analysis," in *24th Annual International Symposium on Computer Architecture ISCA'97*, May 1997, pp. 315–326.

[94] A. Seznec, "A case for two-way skewed-associative caches," in *Proc. 20th Annual Symposium on Computer Architecture*, June 1993, pp. 169–178.

[95] A. González, M. Valero, N. Topham, and J. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *Proceedings of the 11th international conference on Supercomputing*. ACM Press, 1997, pp. 76–83. [Online]. Available: http://doi.acm.org/10.1145/263580.263599

[96] Z. Zhang and X. Zhang, "Cache-optimal methods for bit-reversals," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 1999, p. 26. [Online]. Available: http://doi.acm.org/10.1145/331532.331558

[97] P. Wang, H. Wang, J. Collins, E. Grochowski, R. Kling, and J. Shen, "Memory latency-tolerance approaches for itanium processors: Out-of-order execution vs. speculative precomputation," in *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, I. C. Society, Ed., Boston, Massachusettes, Feb. 2002, pp. 187–196. [Online]. Available: http://www.princeton.edu/~echi/dataspec/wang%20-%20itanium%20memory%20-%20speculative%20vs%20outorder.pdf

[98] W. Triebel, *IA-64 Architecture for Software Developers*. Intel Press, 2000.

[99] M. Lam and et al., "SUIF," 1994-current, http://suif.stanford.edu/.

[100] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[101] P. D'Alberto, "Performance evaluation of data locality exploitation," University of Bologna, Dept. of Computer Science, Tech. Rep., 2000. [Online]. Available: citeseer.nj.nec.com/alberto00performance.html

[102] A. Lenstra and H. Lenstra, "The development of the number field sieve," ser. Lecture Notes in Math, vol. 1554. Springer-Verlag, 1993.

[103] R. Whaley and J. Dongarra, "Automatically tuned linear algebra software, Tech. Rep. UT-CS-97-366, 1997. [Online]. Available: citeseer.nj.nec.com/article/whaley98automatically.html

[104] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, 2005.

[105] P. Clauss, "Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs," in *10th ACM Int. Conf. on Supercomputing*, May 1996.

[106] P. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," in *Int. Conf. on Application Specific Array Processors*, IEEE, Ed., Chicago, Illinois, August 1996.

[107] P. Clauss, "Handling memory cache policy with integer points countings," in *Euro-Par'97 LNCS 1300*, August 1997, pp. 285–293.

[108] ——, "Advances in parameterized linear diophantine equations for precise program analysis," Universit Louis Pasteur, Strasbourg, Tech. Rep. [ICPS RR 98-02], September 1998.

[109] S. Ghosh, M. Martonosi, and S. Malik, "Precise miss analysis for program transformations with caches of arbitrary associativity," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. ACM Press, 1998, pp. 228–239. [Online]. Available: http://doi.acm.org/10.1145/291069.291051

[110] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, no. 8, August 1992.

[111] ——, "Counting solutions to presburger formulas: How and why," in *SIGPLAN Programming language issues in software systems*, Orlando, Florida, USA, 1994, pp. 94–6.

[112] S. Chatterjee, E. Parker, P. Hanlon, and A. Lebeck, "Exact analysis of the cache behavior of nested loops," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM Press, 2001, pp. 286–297. [Online]. Available: http://doi.acm.org/10.1145/378795.378859

[113] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techinques and Tools*. Addison-Wesley publisging company, 1986.

[114] R. F. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," Sun Microsystems Laboratories, Tech. Rep., 1993.

[115] D. Bacon, S. Graham, and O. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, 1994. [Online]. Available: citeseer.nj.nec.com/bacon93compiler.html