DISSERTATION

SCALABLE AND EFFICIENT TOOLS FOR MULTI-LEVEL TILING

Submitted by

Lakshminarayanan Renganarayana

Computer Science Department

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

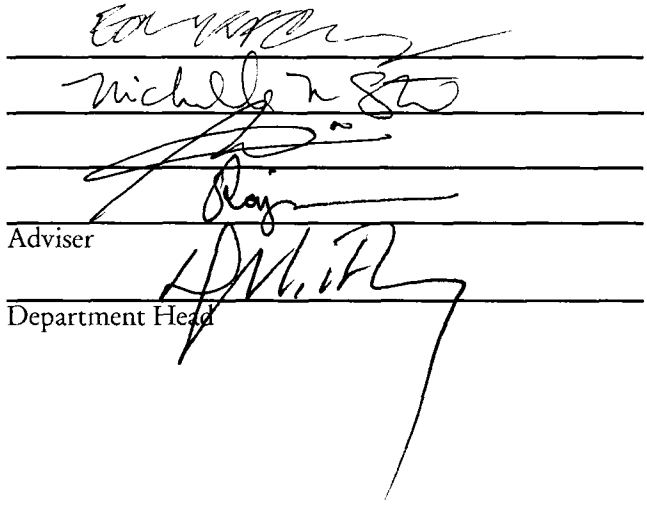
Spring 2008

COLORADO STATE UNIVERSITY

February 29, 2008

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY LAKSHMINARAYANAN RENGANARAYANA ENTITLED SCALABLE AND EFFICIENT TOOLS FOR MULTI-LEVEL TILING BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work



Adviser

Department Head

ABSTRACT OF DISSERTATION

SCALABLE AND EFFICIENT TOOLS FOR MULTI-LEVEL TILING

In the era of many-core systems, application performance will come from parallelism and data locality. Effective exploitation of these require explicit (re)structuring of the applications. Multi-level (or hierarchical) tiling is one such structuring technique used in almost all high-performance implementations. Lack of tool support has limited the use of multi-level tiling to program optimization experts. We present solutions to two fundamental problems in multi-level tiling, viz., optimal tile size selection and parameterized tiled loop generation. Our solutions provide scalable and efficient tools for multi-level tiling.

Parameterized tiled code refers to tiled loops where the tile sizes are not (fixed) compile-time constants but are left as symbolic parameters. It can enable selection and adaptation of tile sizes across a spectrum of stages through compilation to run-time. We define a parametric version of the loop tiling transformation and present a symbolic extension of the Fourier-Motzkin elimination technique for generating parameterized tiled code. To overcome the efficiency and scalability problems of this technique, we introduce two polyhedral sets, viz., inset and outset, and use them to develop a variety of scalable and efficient multi-level tiled loop generation algorithms. The generation efficiency and code quality are demonstrated on a variety of benchmarks such as stencil computations and matrix subroutines from BLAS. Our technique can generate tiled loop nests with parameterized, fixed or mixed tile sizes, thereby providing a one-size-fits all solution ideal for inclusion in production compilers.

Optimal tile size selection (TSS) refers to the selection of tile sizes that optimize some cost (e.g., execution time) model. We show that these cost models share a fundamental mathematical property, viz., positivity, that allows us to reduce optimal TSS to convex optimization problems.

Almost all TSS models proposed in the literature for parallelism, caches, and registers, lend themselves to this reduction. We present the reduction of five different TSS models proposed in the literature by different authors in a variety of tiling contexts. We also present three case studies that illustrate the potential of convex optimization based TSS methods in solving a wider class of loop optimization problems. Our convex optimization based TSS framework is the first one to provide a solution that is both efficient and scalable to multiple levels of tiling.

Lakshminarayanan Renganarayana
Computer Science Department
Colorado State University
Fort Collins, Colorado 80523
Spring 2008

Acknowledgments

I have been very lucky to work under the supervision of Dr. Sanjay Rajopadhye. Thanks to his courageous "Yes, I will take you as my PhD student" (when I knew nothing about polyhedra or parallel computation). Thanks to him for teaching me polyhedra and patience; parallel computation and perseverance; matrices and mathematical rigor; paper writing and proof techniques; and much more. Thanks Sanjay for being a constant source of inspiration to me in both research and personal life.

It has been a wonderful experience working with Dr. Michelle Mills Strout. Thanks to her for teaching me how to do experimental validation and how to present them. Thanks Michelle for all the great moments and interesting discussions.

It is always a pleasure to take Dr. Wim Böhm's course—or even just to drop in his office and talk to him. I have been very fortunate to have had the chances to do both. Thank you Dr. Böhm for the thought provoking problems, puzzles and discussions.

Dr. Edwin Chong's course on Optimization Techniques is one of the courses I enjoyed most at CSU. This course not only influenced a good part of my thesis, but also changed the way I looked at problems. Thank you Dr. Chong for introducing me to the exciting world of Optimization Techniques.

In addition to teaching and inspiring me, Dr. Bohm, Dr. Strout, and Dr. Chong, also agreed to serve on my thesis committee and provided me invaluable feedback. Thank you all.

I would like to thank Dr. Ross McConnell for introducing me to the wonderful world of Graph Algorithms.

I would like to thank Dr. Rob Schreiber and Dr. Darren Cronquist for giving me an opportunity to work on the PICO project at HP Labs.

I would like to thank Ramakrishna Upadrasta in whom I found a great friend. Thank you Rama for all the wisdom and spiritual guidance.

I would like to thank Gautam Gupta, Ramakrishna Upadrasta, DaeGon Kim, Rinku Dewri, Ashish Gupta and Manjukumar Harthikote-Matha for providing a great atmosphere for bouncing and discussing (all kinds of) ideas. I would like to thank all the members of the MELANGE group at CSU—all of you made my CSU life interesting and colorful.

I would like to thank Sharon Van Gorder and Caroll Calliham for helping me with all the administrative process.

I would like to thank my parents for supporting me through all my adventures. I would like to thank my brother Krishna Narayanan for his encouragement and faith in me. Thank you Krishna for all your sacrifices and support—if not for them, I would not have done a PhD. I would like to thank my sister and brother-in-law for all the happy moments and timely wisdom.

I would like to thank my wife Mythili for her love, support and understanding. This dissertation would not have been possible without her. I dedicate it to her.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

> "[...] a broad range of optimization techniques are, in essence, tiling. We argue that tiling should consider storage mapping, scheduling, and communication pipelining decisions; that it encompasses inspector/executor methods; that it can facilitate register allocation, storage compaction, instruction cache optimization, fault tolerance, and adaptive computing on heterogeneous platforms; and so on. "
>
> —*Tiling, the Universal Optimization,* Larry Carter [29]

TODAY'S general purpose computers have multi-core processors. As the number of cores on a chip doubles every year, very soon there will be a few hundred cores—called *many cores*—on a single chip. This trend of many-core general purpose processors has changed the primary mode of performance improvement—applications need to be *explicitly restructured to exploit parallelism and memory hierarchy [120].* Such restructuring could be done automatically (by compilers or auto-tuners) or manually (by application/library developers). Program transformation tools that

can aid in this restructuring play a fundamental enabling role in achieving the performance potential of many-core systems. The lack of such tools is evident from the widening gap between peak performance of systems and the attained performance of real applications.

The compute and data intensive parts of several important applications are loop kernels. High-performance implementations of these kernels directly translate to application level high-performance. One of the important loop transformation used in high-performance implementations is tiling [62, 117, 78, 136]. Tiling matches program characteristics (locality, parallelism, etc.) to those of the execution environment (memory hierarchy, registers, number of processors, etc.). Often, multiple levels of tiling are used to account for the hierarchy of resources. Given a loop nest, tiling partitions its iterations into groups called *tiles*. These tiles form the execution units with improved performance. The improvement is through parallel execution and/or better data locality.

Parallel systems include an hierarchy of resources: hundreds or thousands of processors, an interconnection network, an hierarchy of shared and private memories, tens of floating point registers, and pipelined superscalar functional units [30]. Figure 1.1 shows an example parallel system with three levels of resources. The bottom level represents the parallelism induced by a set of processors connected through an interconnection network. Here communication is expensive. Tiling has been used to coarsen the granularity of the computation blocks so that the frequency of communication is reduced. The middle level represents an hierarchy of private and shared memory (or caches). Tiling has been used in this context to improve data locality. The top level consists of registers and pipelined functional units. In this context, register tiling (also known as loop unrolling plus scalar replacement) is used to expose instruction level parallelism (ILP) and to promote array values to registers.

High-performance implementations of loop programs typically employ multiple levels of tiling [30]. For example, the highly tuned matrix multiplication implementation generated by ATLAS or PHiPAC [126, 16] uses two levels of tiling: one for caches and another for registers and ILP. Furthermore, with the advent of multi-core processors in general purpose computers, an additional level of tiling for parallelism has become necessary. Multi-level tiling has almost become a design pattern for high performance implementations. Whenever a programmer is faced with the problem of deriving a high-performance implementation from a sequential specification

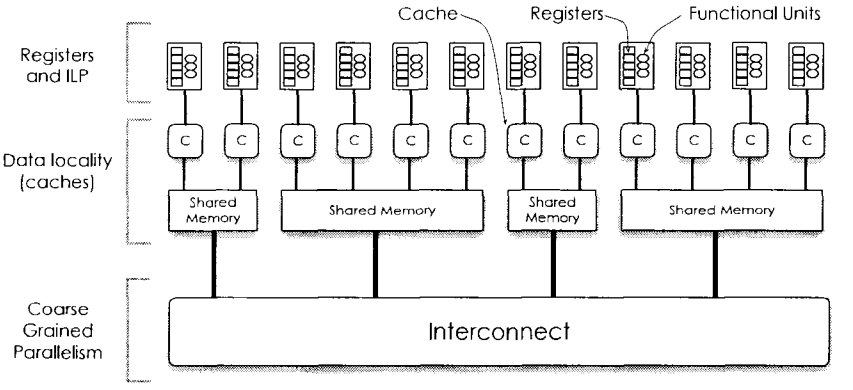


Figure 1.1.
Tiling at various levels of a resource hierarchy. Top layer represents registers and functional units. Middle layer represents private or shared memories. The bottom layer represents the network that connects multiple processors.

of an algorithm, multi-level tiling guides the structuring of the implementation. Language level abstractions such as hierarchical tiled arrays (HTA) [15] reify tiles as first class objects and directly support the use of multi-level tiling as a design pattern.

To summarize, multi-level tiling is emerging as a standard structuring technique for high-performance implementations. Effective use of it requires efficient and scalable tools for *tiled code generation* and *tile shape/size selection.* Tiled code generation involves the generation of the transformed or tiled loop nest and the loop body. The shape and size of the tiles are selected such that the resulting execution time is minimized. In this thesis, we focus on tile size selection and tiled loop generation.

The rest of the chapter is organized as follows. The next section introduces the tile size selection problem, describes the limitations of the current approaches and presents an outline of our solution. Section 1.2 introduces the problem of tiled loop generation, describes the limitations of the current approaches and presents our technique for multi-level tiled loop generation. The chapter closes with an overview of the dissertation.

1.1 Tile Size Selection

Tile size selection has been studied for almost two decades now. As early as 1969, McKellar and Coffman [84] studied how to match the organization of matrices and their operations to paged memory systems. Early studies of such matching, in the context of program transformation, were done by Abu-Sufah *et al.* [3] and Wolfe [130]. Solutions ranging from closed form solutions [4, 20, 26, 56, 91, 98, 12, 10, 134, 137, 138, 117] to heuristic algorithms [78, 60, 33, 36, 99, 48, 115, 63, 77, 116, 85] to exhaustive search [126, 16, 72] have been proposed. Cost models that characterize the performance of a tiled loop nest in terms of tile sizes are used for selecting the best tile sizes. These cost models are closely tied to the execution platform (architecture, communication network, run time libraries, etc.). The two primary limitations of current tile size selection methods are (i) non-extensibility to newer architectures and program classes and (ii) non-scalability to multiple levels of tiling. Given the rapidly changing landscape of multi-core systems, there will be considerable variation in processor architectures, and memory hierarchies will probably be deep and user managed. In such a scenario, effective use of tiling requires tile size selection frameworks which (i) allow extensions and adaptations of cost models and (ii) scale to multiple levels of tiling.

1.1.1 Limitations of Current Approaches

We first describe the design process used by current methods and identify their limitations. *Optimal Tile Size Selection* (TSS) is the problem of selecting the tile sizes that are *optimal* with respect to a given cost model. For example, in the use of tiling to improve cache locality, consider the selection of sizes x and y which form the sides of a 2D tile. A widely used cost function is the number of cache misses. This cost function is used, together with the constraint that the data accessed by a given tile—*tile footprint*—fits in the cache. The corresponding optimal TSS problem can be stated as follows:

$$\begin{aligned} &\text{select } x,y \quad \text{which minimize} \quad Misses(x,y) \\ &\text{subject to} \quad FootPrint(x,y) \leq CacheCapacity \end{aligned} \tag{1.1}$$

where, $Misses(x,y)$ estimates the number of misses experienced with a tile of size $x \times y$, $FootPrint(x,y)$ estimates the number of cache lines touched by a tile of size $x \times y$, and $CacheCapacity$ is the capacity of the cache in number of lines. The cost function together with the constraint is called the *cost model.* One can view the optimal TSS problem as a constrained optimization problem and in such a view the cost function is also referred to as the *objective function.*

All TSS solutions proposed currently in the literature follow a design process that can be summarized as follows:

1. *Design a cost model.* This includes the design of a cost metric (objective function) that estimates a desired quantity as a function of tile sizes and constraints that qualify tile sizes as valid or not. The cost models seek to estimate quantities that are related to the execution characteristics and hence are inherently strongly tied to the class of programs and architectural features for which they are designed.

2. *Reason about the structure of the cost functions.* For example, one can check whether the objective function is linear or quadratic in terms of the tile size variables.

3. *Exploit the properties of functions to derive a closed form solution or a heuristic/search algorithm.*

As an illustration, consider the optimal tiling problem proposed by Andonov et al. [11]. They study the problem of tiling 2D iteration spaces with uniform dependencies for parallel SPMD style execution on distributed memory machines. They come up with a cost model, after a detailed study of the class programs they want to tile, the architectural parameters, and the execution characteristics. The objective function $T(x,y)$ estimates the total (parallel) execution time of the tile program and the goal is to pick the tile sizes that minimize this metric. The objective function and the constraints can be abstractly viewed as

$$\begin{aligned} \text{min. } T(x,y) &= \frac{A}{xy} + Bxy + Cx + \frac{D}{y} + E \\ \text{subject to } x,y &\geq 1,\ x,y \in \mathbb{Z} \end{aligned}$$

where x,y are the tile size variables and A,B,C,D,E are constants. Then they use the following

reasoning to obtain a closed form solution: for $xy = K, K \in \mathbb{R}$, the function $T(x,y)$ monotonically decreases with x. As a result, the optimal solution is on certain boundaries of the feasible space, and using this information, one of the variables can be eliminated, yielding a closed form solution for x and y.

A subtle but important feature of the above process is the following: the cost model is strongly coupled to the program class/architectural features and the solution (method) is derived by exploiting the properties of the functions used in the cost model. Any extensions of the cost model to a different architecture, richer program class, or to multiple levels of tiling, change the structure of the functions used in the cost model, and hence leave the solution (method) inapplicable. For example, an extension of the Andonov et al.'s model to a richer program class, viz., 3D iteration spaces requires the solution of a completely different problem [12].

All TSS solutions proposed in the literature are cost model specific and do not lend themselves to extensions. Any non-trivial extension typically requires an effort equal to or more than the earlier one, and are often publishable results (e.g., extension from direct mapped caches to set associative caches, from 2D to nD iteration spaces, etc.). Typically, one wants to use a TSS solution for a program class or architecture that is slightly different than the one considered by the author of the solution. But accounting for the differences lead to changes in the cost model, which leaves the solution inapplicable. This is in fact an important reason for the popularity of exhaustive search (run the program for different tile sizes and pick the best).

Given the trend towards multi-core parallel architectures high-performance implementations use two to three levels of tiling [31, 138, 103]. For example, an outer level of tiling for parallelism, another level for cache locality, and another for registers and ILP are used. Mitchell et al. [85] have shown, in three different architectural scenarios, that the tiling parameters from different levels interact with each other and a level-by-level independent selection of the tile sizes will lead to sub-optimal performance. However, due to the non-scalability of the current optimal tiling solutions, such a level-by-level approach is very common. The scalability limitation of current approaches is once again due to their strong dependence on the properties used in the cost model. For example, in a 2D one level tiling, the optimal tiling problem has the two tile sizes as variables and functions used in the cost models are of degree at most two (linear, quadratic, etc.) and are easy to reason about. However, when we move to two levels of tiling there are four variables and

functions of four variables with degree up to four are much harder to reason about.

To summarize, the cost-model specificity of the solution methods lead to their non-extensibility and non-scalability. Our framework overcomes these limitations by providing a cost-model independent solution method. When using our framework one does not have to perform the second and third steps of the traditional optimal TSS design process described above.

Despite the aforementioned limitations, current methods are very efficient, whenever they are applicable. For example, an optimal tiling solution which provides closed form expressions for the optimal tile sizes is very efficient to use when compared to our approach which requires an optimization solver. Unfortunately, reusing such optimal tiling methods require significant extensions and adaptations.

1.1.2 A Unified Tile Size Selection Framework

On a more fundamental note, one might speculate about the existence of a formalism that might allow the formulation and solution of tile size selection problems independent of the specific cost models used. To better understand this quest, consider the analogy of loop transformations. The class of linear transformations serve as foundational formalism for expressing and reasoning about a wide variety of loop transformations independent of what they are used for (parallelism, cache locality, register locality, etc.). We are asking whether we can find one such formalism for tile size selection.

In this thesis we show that there exists one such formalism and that using it for modeling tile size selection leads to extensible models and a scalable solution method. Further, we show how the closure properties of the formalism can be exploited to design multi-level optimal tiling models from single-level models via composition. Even though this is the first time this formalism is proposed as a generic tile size selection method, almost all the optimal tiling models proposed in literature can be directly cast in this formalism and solved efficiently. In fact, many of them were already expressed in this formalism without knowing about it, and hence did not benefit from it earlier.

We identify a fundamental property, viz., *positivity,* that is shared by many mathematical expression and terms used in a wide variety of optimal tiling models. Based on this positivity property, we identify a class of functions called *posynomials* that can serve as a formalism for spec-

ification of optimal tiling problems. By formulating a class of non-linear optimization problems using posynomials, we propose an efficient, scalable and cost-model independent framework for optimal tile size selection. We show that almost all the tiling models proposed in the literature can be cast into our framework. To substantiate this claim, we describe the reduction of five different tiling models (from a wide range of tiling contexts) to this framework. We also show how the closure properties of posynomials can be exploited to extend single level models and/or compose them to form multi-level tiling models. We have implemented a MATLAB based tool for using posynomials to model and solve optimal tiling problems.

To the best of our knowledge this is the first framework that can scale to an arbitrary number of levels of tiling and still be efficient and extensible. Further, it is insightful to find that such a framework can be derived by exploiting a simple but fundamental property shared by all optimal tiling models. Note that the goal of our work is not to prove tiling is useful—several authors have shown this. Our goal is to propose a framework that not only unifies the variety of TSS models proposed in the literature, but also lays the foundations to build more sophisticated models.

We got the insight about the positivity property only after developing posynomial based tile size selection models in three different contexts, viz., (i) multi-level tiling for parallel execution of 3D stencil computations [103]; (ii) tiling for registers and ILP [102]; (iii) multi-level tiling to improve data locality of uniform dependence computations [101]. These three solutions are also included in this thesis.

1.2 Parameterized Tiled Loop Generation

One of the important steps in application of tiling to a loop kernel is the generation of the tiled or transformed loop nest. Tiled loop generation refers to the generation of the bounds of the tiled loop nest. Parameterized tiled loop generation refers to the generation of tiled loop nests in which the tile sizes are not fixed, but left as symbolic parameters, which can be fixed/tuned at a later stage. First we motivate the need for parameterized tiled code and then present the limitations of current approaches.

The optimal tile sizes are very sensitive to characteristics of the execution environment such as available cache size, processor work load, network latency, etc. Traditionally loop tiling has

been viewed as a static, compile time optimization. Compilers use analytical models to select tile sizes and generate tiled code with fixed tile sizes. Tile sizes that are selected and fixed at compile time can be far from optimal due to changes in execution environments. Such fixed tile size codes are rigid and cannot adapt themselves to changes in the execution environment.

The Self Adapting Numerical Software (SANS) effort [41] is a strong evidence of the need for numerical software—primarily loop programs—to be more adaptive. An important parameter that is adapted/tuned in SANS is the tile size [39]. Further, tile size is also an important parameter tuned by iterative compilers [73] and the so called auto-tuners such as ATLAS [126], OSKI [124], and PHiPAC [16]. Run-time tile size adaptation has been shown to improve performance in the context of parallelism [83] as well as data locality in shared memory [90]. Another important use of tile size adaptation is in the context of utility computing, where programs are expected to be mobile—migrate and adapt to a new set of resources [46]. Such adaptations with respect to the number of processors and memory characteristics can be directly mapped to tile size adaptations.

The above discussion shows a spectrum of stages at which tile sizes are tuned/fixed/adapted: classic compile-time by compilers; install time by auto-tuners; load-time (beginning of the execution) in parallel programs to adapt for number of available processors; during run-time for data locality in shared memory; and during reconfiguration time in mobile programs for adapting to a new set of resources. As discussed in previous sections, often multiple levels of tiling are used. In such a scenario, we need to generate a multi-level, parameterized tiled loop nest.

1.2.1 Limitations of current approaches

There is an easy solution to the parameterized tiled loop generation problem: simply produce a parameterized tiled loop for the *bounding box* of the iteration space, and introduce guards to test whether the point being executed belongs to the original iteration space. When the iteration space is itself (hyper) rectangular, as in matrix multiplication, this method is obviously efficient. However, many important computations, such as LU decomposition, triangular matrix product, symmetric rank updates, do not fall within this category. Moreover, even if the original iteration space is (hyper) rectangular, the compiler may choose to perform skewing transformations to exploit temporal locality or parallelism (e.g. stencil computations) thus rendering it parallelepiped shaped. Parallelepiped-shaped iteration spaces also occur when skewing is performed to make

(hyper) rectangular tiling legal. For such programs, the bounding box strategy results in poor code quality, because a number of so called "empty tiles" are visited and tested for emptiness. Another drawback for the bounding box strategy is that calculating the bounding box of arbitrary iteration spaces may be time-consuming. The worst-case time complexity of computing a bounding box is exponential [13].

The main difficulty with generating parameterized tiled loop code has been the fact that the Fourier-Motzkin elimination technique that is used for scanning polyhedra [9] does not naturally handle symbolic tile sizes, and leads to a nonlinear formulation. Amarasinghe proposed a symbolic extension of the standard Fourier-Motzkin elimination technique [8, 7] and implemented it in the SUIF system [127]. It is well known that Fourier-Motzkin elimination has doubly exponential worst case complexity. The symbolic extension inherits this worst case complexity, adds to the number of variables in the problem, and reduces the possibilities for redundancy elimination.

Though multi-level tiling is widely used, the multi-level tiled loop generation problem has not been widely studied. In fact, we are aware of only one solution that can generate arbitrary levels of multi-level tiled code for general polyhedral iteration spaces [65]. Their technique is limited to the case when tile sizes are fixed at compile (tiled loop generation) time.

1.2.2 Parameterized tiled loop generation using Outset

We present a simple and efficient approach for generating parameterized tiled code that handles any polyhedral iteration space and parameterized (hyper) rectangular tilings. We show that the problem can be decomposed into two sub problems of generating: (i) loops that iterate over tile origins and (ii) loops that iterate over the points within tiles. These sub problems can be formulated as a set of linear constraints where the tile sizes are parameters, similar to problem size parameters. This allows us to reuse existing code generators for polyhedra, such as CLooG [14], and implement our code generator through simple pre- and post-processing of the CLooG input and outputs. The key insight is expressing the bounds for the tile loops as a super set, called *outset,* of the original iteration space and then post processing the generated loops by adding a stride and modifying the computation of the lower bounds.

We present an algorithm that generates tiled loops from any parameterized polyhedral iter-

ation space, while keeping the tile sizes symbolic variables. The fact that our algorithm can be directly applied to the case when the tile sizes are fixed, makes our method a one-size-fits-all solution, ideal for inclusion in production compilers. We present an empirical evaluation on benchmarks such as LUD and triangular matrix product show that our algorithm is both efficient and delivers good code quality. Our experiments present the first quantitative analysis of the cost of parametrization in tiled loops generation. We also present an algorithm that separates the loops into those that iterate over partial tiles and those that iterate over full tiles. Such a separation has the added benefit that it enables transformations like loop unrolling or software pipelining, (which are often applied only to rectangular loops) to be applied to the (rectangular) loops that iterate over the full tiles. Our implementation is available as open source software [55].

The concept of outset can also be used for generating multi-level tiled loops. We propose a technique for generating multi-level tiled loops where the tile sizes can be fixed (constants) or symbolic parameters or mixed. Our technique provides multiple-levels of tiling at the same cost of generating tiled loops for a single level of tiling. We propose a novel formalization of the classic tiling transformation [62, 136] to multiple levels. We propose a method for separating partial and full tiles at any arbitrary level, without fixing the tile sizes. We have implemented all the proposed code generation techniques and the tool is available open source [55]. We present extensive evaluation of both the generation efficiency and quality of the generated code on benchmark routines form BLAS, LUD, and stencil computations.

1.3 Overview of the dissertation

The dissertation is broadly separated into two independent parts: (i) tiled loop generation and (ii) optimal tile size selection. In the first part on tiled loop generation, we first introduce the basic concepts of inset and outset. We then present the tiled loop generation algorithms for single-level followed by its extension to multiple-levels. Then, we present the techniques used for separating full/partial tiles.

In the second part on optimal tile size selection, we first present a survey of the current approaches. After introducing the background on posynomials, geometric programs and convex optimization, we present the optimal tile size selection framework. To show that appropriateness

of the framework for modeling a wide variety of TSS problems, we present the reduction of five different TSS models proposed by a different authors in the contexts of TSS for parallelism, data locality, and register locality and ILP. Then we present the three models: (i) multi-level tiling for parallel execution of 3D stencil computations [103]; (ii) tiling for registers and ILP [102]; and (iii) multi-level tiling to improve data locality of uniform dependence computations [101].

Part I

Tiled Loop Generation

CHAPTER 2

Parameterized Tiling and Symbolic Fourier-Motzkin Elimination

> The formulation of a problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill.
>
> – Albert Einstein

In this chapter we present an extension of the classic tiling transformation formulation [62, 135] to the case where the tile sizes are not fixed but left as parameters. We present this formulation and a Symbolic Fourier-Motzkin Elimination (SFME) algorithm for generating parameterized tiled code. We also present proofs of the correctness of the SFME algorithm and its applicability to the system of constraints resulting from the parameterized tiling transformation. This extension of tiling formulation to the parametric case has theoretical significance. However, for efficient practical code generation one should prefer the outset based methods presented in the subsequent chapters.

The work presented in the chapter was done in collaboration with Michelle Mills Strout.

2.1 Background, program and tiling model

The notation $\vec{x}$ indicates that x is a vector. $\vec{0}$ and $\vec{1}$ represent all-zero and all-one vectors, respectively. The relational operators, $<, =, >, \leq$, and $\geq$, between two vectors are component-wise. For $a \in \mathbb{R}$ we write $\lfloor a \rfloor$ to denote its *floor* and $\lceil a \rceil$ its *ceiling*, which respectively are, the largest integer not greater than a, and the smallest integer not smaller than a. When used in the context of vectors, floor and ceiling functions are applied component-wise, for example: $\lceil \vec{x} \rceil = (\lceil x_1 \rceil, \ldots, \lceil x_n \rceil)$. We denote component-wise multiplication of two vectors $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{y} = (y_1, \ldots, y_n)$ with $\vec{x} \circ \vec{y} = (x_1 y_1, \ldots, x_n y_n)$.

The symbolic Fourier-Motzkin elimination algorithm takes advantage of the fact that the bounds for the tiled loop are bilinear with respect to the parameterized tile sizes. If V is a vector space over a ground field K (i.e., for this chapter the field is the set of real numbers), then a function $f : V \rightarrow K$ is called a linear function if for any two vectors $\vec{x}$ and $\vec{y}$ in V and a scalar a in K the following two properties $f(\vec{x} + \vec{y}) = f(\vec{x}) + f(\vec{y})$ and $f(a\vec{x}) = af(\vec{x})$ are satisfied. A function g is called *affine* if it can be written of the form $g(\vec{x}) = f(\vec{x}) + c$, for some linear function f and some constant $c \in K$. For example, $f(x_1, x_2) = 3x_1 + 4x_2 + 3$ is an affine function.

If V_x and V_y are two vector spaces over some ground field K, a function $h : V_x \times V_y \rightarrow K$ is called *bilinear* if for a fixed $\vec{v} \in V_x$, $h(\vec{v}, \vec{y})$ is linear for all $\vec{y} \in V_y$ and for a fixed $\vec{u} \in V_y$, $h(\vec{x}, \vec{u})$ is linear for all $\vec{x} \in V_x$. Informally, for a fixed value of $\vec{x}$, $h()$ is linear in $\vec{y}$, and vice-versa. For example, $h(x_1, x_2, y_1, y_2) = 2x_1 y_1 - 3x_2 y_2$ is a bilinear function, whereas $h'(x_1, x_2, y_1, y_2) = 2x_1^2 y_1 - 3x_2 y_2$ is not. One can also define bi-affine functions in a fashion similar to that of affine functions.

An inequality of the form $f(\vec{x}) \leq 0$, for any affine function $f(\vec{x})$ will be loosely called as a linear inequality, though strictly it should be called an affine inequality. In a similar vein, an inequality of the form $h(\vec{x}, \vec{y}) \leq 0$, where $h(\vec{x}, \vec{y})$ is a bilinear (or biaffine) function is called a bilinear inequality.

Our notation here closely follows that of Xue's [136]. A rectangular tiling is fully characterized by the tile size vector $\vec{s}$, where s_i is the tile size for the ith dimension of the iteration space.

```
for i1 = 1...N1
  for i2 = 1...i1
      S(i1, i2)
```

Figure 2.1. A 2D loop nest with triangular iteration space.

Given an iteration space

$$P = \{\vec{i} \mid Q\vec{i} \leq \vec{q} + B\vec{p}\},$$

a rectangular tiling τ maps iterations of the n-dimensional iteration space P into a $2n$-dimensional iteration space. In general, τ is defined as follows:

$$\tau(\vec{i}) = \begin{pmatrix} \vec{t} \\ \vec{e} \end{pmatrix} = \begin{pmatrix} \left\lfloor \frac{\vec{i}}{\vec{s}} \right\rfloor \\ \vec{i} \end{pmatrix},$$

where $\vec{t} = \left\lfloor \frac{\vec{i}}{\vec{s}} \right\rfloor$ identifies the index of the tile that contains the point $\vec{i}$.

The *tiled iteration space,* denoted by T, is the image of P by the tiling transformation τ, and can be characterized by

$$\mathbf{T} = \{ (\vec{t}, \vec{i}) \mid Q\vec{i} \leq \vec{q} + B\vec{p}, \vec{s} \circ \vec{t} \leq \vec{i} \leq \vec{s} \circ \vec{t} + \vec{s} - \vec{1} \}, \tag{2.1}$$

where the bounds represented by $Q\vec{i} \leq \vec{q} + B\vec{p}$ make sure that all $\vec{i}$ belong to the original iteration space P, and $\vec{s} \circ \vec{t} \leq \vec{i} \leq \vec{s} \circ \vec{t} + \vec{s} - \vec{1}$ defines the iterations that are contained in the tile $\vec{t}$. In addition to the above constraints, the program parameters $\vec{p}$ and tile size parameters $\vec{s}$ will also have some linear constraints. For example, they must all be greater than 1. We gather all these linear constraints into a set C. When the tile sizes are fixed, i.e., $\vec{s}$ is a vector of given constants, then T defines a convex polytope.

The tiled iteration space of the triangular loop nest given in Figure 2.1, for fixed tile sizes $s_1 = 2$ and $s_2 = 3$, is given by

$$\begin{aligned}\mathbf{T}_{tri} &= \{t_1, t_2, i_1, i_2 \,|\, 1 \le i_1 \le N_1, 1 \le i_2 \le i_1, \\ &\quad 2t_1 \le i_1 - 1 \le 2t_1 + 2 - 1, 3t_2 \le i_2 - 1 \le 3t_2 + 3 - 1\}.\end{aligned}$$

It is easy to note that this is a convex polyhedron of four dimensions. In addition to the above constraints we also have the following constraints on the program parameter N_1 and the tile size parameters s_1 and s_2.

$$C_{tri} = \{N_1, s_1, s_2 | N_1 \ge 1, 1 \le s_1, s_2 \le N_1\}. \tag{2.2}$$

The constraints C_{tri} can be viewed as the *context* in which the tiled iteration space $\mathbf{T}_{tri}$ is defined.

2.2 Parameterized Tiled Iteration Space

When the tile sizes are not fixed, but used as symbolic parameters, the constraints that define $\mathbf{T}$ in Equation (2.1) are no longer affine, but *bilinear.* The inequalities that define the constraints are formed with functions that are bilinear over the index space spanned by $(\vec{t}, \vec{i})$ and the parameter space spanned by $(\vec{p}, \vec{s})$. We will work with this *parameterized tiled iteration space* (PTIS), $\mathbf{T}(\vec{t}, \vec{i}, \vec{s}, \vec{p})$, in which the tile sizes are symbolic parameters (not fixed constants).

We can represent the set of bilinear inequalities that define the PTIS, $\mathbf{T}(\vec{t}, \vec{i}, \vec{s}, \vec{p})$ (c.f. Equation 2.1) in a matrix form as follows:

$$\begin{bmatrix} 0 & Q \\ S & -I \\ -S & I \end{bmatrix} \begin{pmatrix} \vec{t} \\ \vec{i} \end{pmatrix} \le \begin{pmatrix} \vec{q} \\ 0 \\ -1 \end{pmatrix} + \begin{bmatrix} B & 0 \\ 0 & 0 \\ 0 & I \end{bmatrix} \begin{pmatrix} \vec{p} \\ \vec{s} \end{pmatrix}, \tag{2.3}$$

where $S = \mathsf{diag}(\vec{s})$ is a diagonal matrix with the tile sizes from $\vec{s}$ as its entries, and I is an identity matrix of appropriate size. For notational convenience we denote the matrix form in Equation 2.3 by the following simpler form

$$\Gamma \vec{z} \le \vec{\gamma}, \tag{2.4}$$

where Γ is the matrix on the left hand side of Equation 2.3, $\vec{z} = (\vec{t}\ \vec{i})^T$, and $\vec{\gamma}$ is the matrix

expression on the right hand side of Equation 2.3. Hence a PTIS is completely characterized by $\Gamma\vec{z} \leq \vec{\gamma}$ and the set of linear constraints on the program and tile size parameters C.

2.2.1 Properties of a PTIS

Let us consider a PTIS defined by $\Gamma\vec{z} \leq \vec{\gamma}$. A closer look at the definition of PTIS in Equation 2.1, and the expanded matrix form in Equation 2.3, reveals that the program size parameters $\vec{p}$ will always only appear as an additive part in $\vec{\gamma}$, and not in the bilinear part in Γ. The entries of Γ are either rational numbers (coming from the linear inequalities of the original iteration space, i.e., form Q) or linear functions of tile size variables $\vec{s}$ (coming from the last two block rows of Equation 2.3). In fact, there is even more structure to Γ, which is stated in the following *bilinear set* property.

Definition 2.2.1 (BLIS-PROPERTY). *Let l be any column of Γ. All the components of l are either exclusively rational numbers or exclusively zeros and linear functions of a single tile size variable, s_k for some $k = 1 \ldots n$.*

Note that PTIS (c.f. Equations. 2.1 and 2.3) satisfies BLIS-PROPERTY. The BLIS-PROPERTY is a fundamental property which is also preserved after every step of the symbolic Fourier-Motzkin elimination algorithm we propose. In a geometric sense, similar to the projections of polyhedra onto lower dimensions, one can view PTIS as a set, and observe that the operation *p*rojection onto lower dimensions preserves the BLIS-PROPERTY.

2.2.2 PTIS of the Example

Let us now look at the constraints that define the parameterized tiled space of the triangular loop nest given in Figure 2.1. We have

$$\begin{aligned} \mathbf{T}_{tri}(\vec{t},\vec{i},\vec{s},\vec{p}) \;=\; & \{t_1, t_2, i_1, i_2 \mid \\ & 1 \leq i_1 \leq N_1,\ 1 \leq i_2 \leq i_1, \\ & s_1 t_1 \leq i_1 - 1 \leq s_1 t_1 + s_1 - 1, \\ & s_2 t_2 \leq i_2 - 1 \leq s_2 t_2 + s_2 - 1\} \end{aligned}$$

where $(\vec{t},\vec{i}) = (t_1, t_2, i_1, i_2)$ and $\vec{s} = (s_1, s_2)$, $\vec{p} = (N_1)$. The constraints C on the program and tile size parameters are given by Equation (2.2). $\mathbf{T}_{tri}$ can be represented in matrix form of Equation (2.4) as follows:

$$\begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 \\ s_1 & 0 & -1 & 0 \\ -s_1 & 0 & 1 & 0 \\ 0 & s_2 & 0 & -1 \\ 0 & -s_2 & 0 & 1 \end{bmatrix} \begin{pmatrix} t_1 \\ t_2 \\ i_1 \\ i_2 \end{pmatrix} \leq \begin{bmatrix} -1 \\ N_1 \\ -1 \\ 0 \\ -1 \\ s_1 \\ -1 \\ s_2 \end{bmatrix}.$$

One can observe that the bilinear set property BLIS-PROPERTY is satisfied in $\mathbf{T}_{tri}$.

Two properties of FME that have been very useful in the context of code generation are:

Definition 2.2.2 (FME-PROPERTY 1). *Given a system $\mathcal{S} = \{\vec{z} \mid \Gamma\vec{z} \leq \vec{\gamma}\}$, let $\mathcal{S}'$ be the set of constraints after elimination of a variable z_i. For every valid value of z_i' of z_i there exists a $z' \in \mathcal{S}'$ such that we can extend z' with z_i' to get a solution to the original system of constraints $\mathcal{S}$.*

Definition 2.2.3 (FME-PROPERTY 2). *The* FME *algorithm terminates with an inconsistent set of constraints if and only if the original set of constraints is inconsistent.*

In the next section we show how an extension of this classic method can be used to generate tiled loops with variable tile sizes.

It is well known that FME is in spirit an elimination algorithm, whose principles are applicable to a broader class of quantifier elimination problems. Eaves and Rothblum [43, 44] studied the transfer of the elimination principles to other problems, such as elimination of variables in a system of linear constraints, where the coefficients of the linear constraints are not constants but parameters. Weispfenning [125] has proposed an efficient variant of FME which can also eliminate variables in a system of linear constraints where the coefficients in a linear constraint are polynomials of parameters.

Such extensions of FME to parametric problems do not come for free! An important step in the FME algorithm is distinguishing the sign of the coefficient of a variable. When the coefficients are constants, as in the case of linear constraints, this is straight forward. However, when the coefficients are parameters, or polynomials of parameters, whose result could be of either sign, both the positive and negative cases need to be considered and this results in an exponential sized tree of

cases which distinguish the sign of the coefficients. This explosion of cases makes it impossible to apply FME to any reasonable sized problem, when the sign of the coefficients are indeterminable.

A key insight that makes such parametric FME work for our problem, at no additional complexity than FME for simple linear constraints, is the following. *The sign of coefficients in the bilinear constraints that define a* PTIS *(c.f. Eq. 2.4) can always be determined. Further, this property is preserved across elimination of variables.* This is formally stated and proved in Section 2.5.

The general FME style elimination algorithms considered by Weispfenning [125] and Eaves and Rothblum [43, 44] also enjoy the two important properties, namely FME-PROPERTY 1 and FME-PROPERTY 2. This allows us to use the SFME algorithm to check whether the given input set of constraints is feasible or not, and also for removing redundant constraints, as shown in Section 2.7.

2.2.3 The SFME Algorithm

The Symbolic Fourier Motzkin Elimination (SFME) algorithm is given in Algorithm 1. It takes two inputs: (i) an $m \times (n+1)$ column augmented matrix Γ constructed from $\bar{\Gamma}$ and $\vec{\gamma}$ related by a system $\bar{\Gamma}\vec{z} \leq \vec{\gamma}$ and (ii) a set of linear constraints C on the program and tile size parameters. It eliminates z_n from the system Γ. It returns L_n, U_n and Γ' as results, which respectively are, the lower bounds on z_n, upper bounds on z_n, and the set of constraints on the remaining $(z_1, \ldots, z_{n-1})$ variables. Note that these are also in column augmented form.

2.3 Symbolic FME Algorithm

We successively call the SFME until all variables are eliminated. Let $\mathscr{B}$ be the list that collects all the lower and upper bounds of all the eliminated variables. After the last variable, i.e., z_1 is eliminated, the resulting set of constraints in Γ' (returned by the last call to SFME) are constraints involving the program and tile size parameters, i.e., $\vec{p}$ and $\vec{s}$. If these constraints are inconsistent, then the original system of constraints given to SFME is inconsistent. After all the variables are eliminated, we have their bounds in $\mathscr{B}$. We perform a global redundancy check among the bounds in $\mathscr{B}$ as discussed in 2.7. After performing this check, we generate loops for each variable using bounds in $\mathscr{B}$, as discussed in 2.6. A detailed description of the steps in the SFME algorithm

Algorithm 1 Symbolic Fourier Motzkin Elimination (SFME) algorithm. Eliminates one variable from a given system of constraints.

Input: A $m \times (n+1)$ column augmented matrix Γ, related to a system $\bar{\Gamma}\vec{z} \leq \vec{\gamma}$. Γ is the matrix $\bar{\Gamma}$ augmented with the column $\vec{\gamma}$. A set of linear constraints C on the program and size parameters.
Output: L_n, U_n, and Γ'. L_n and U_n are matrices with the lower and upper bound rows of z_n, the eliminated variable, respectively. The new set of rows that constitute the bounds of the remaining variables $(z_1, \ldots, z_{n-1})$ is returned in (the column augmented matrix) Γ'.

1. Compute lower and upper bound matrices.
   $L_n \leftarrow \{\Gamma_{*,n} | \Gamma_{i,n} < 0\}$. *(lower bound rows)*
   $U_n \leftarrow \{\Gamma_{*,n} | \Gamma_{i,n} > 0\}$. *(upper bound rows)*
   $R_n \leftarrow \{\Gamma_{*,n} | \Gamma_{i,n} = 0\}$. *(rest of the rows)*

2. `EliminateRedundants`(L_n, C)
   (eliminate redundant lower bounds of z_n)
   `EliminateRedundants`(U_n, C)
   (eliminate redundant upper bounds of z_n)

3. **For each** pair of rows $(l_a, u_b) : l_a \in L_n$ and $u_b \in U_n$ **do**
   (to compare l_a and u_b, scale them first and then add them)

   (a) $$\beta_l \leftarrow \begin{cases} |l_{a,n}| & \text{if } l_{a,n} \text{ is rational} \\ |\alpha_{a,n}| & \text{if } l_{a,n} = \alpha_{a,n} \times s_k, \\ & \text{for some } s_k. \end{cases}$$
   (extract abs. value of coefficient of $l_{a,n}$)

   (b) $$\beta_u \leftarrow \begin{cases} |u_{b,n}| & \text{if } u_{b,n} \text{ is rational} \\ |\alpha_{b,n}| & \text{if } u_{b,n} = \alpha_{b,n} \times s_k, \\ & \text{for some } s_k. \end{cases}$$
   (extract abs. value of coefficient of $u_{b,n}$)

   (c) $g \leftarrow \gcd(\beta_l, \beta_u)$

   (d) $x \leftarrow \frac{\beta_u}{g} \times l_a + \frac{\beta_l}{g} \times u_b$
   (scale the pair of rows and add them to get the new row in which the coefficients of z_n is canceled out)

   (e) **if** (`notRedundant`(x, C))
   Add the new row x to Γ'.
   (ignore x if it is a bound implied by the constraints on the parameters C)

4. Add rows R_n to Γ'.

follows.

In step (1), the rows from Γ that correspond to the lower and upper bounds of z_n are calculated and stored respectively in L_n and U_n. The rows that do not contribute towards any bound for z_n are stored in R_n. Notice that in this step we should be able to determine the sign of $\Gamma_{i,n}$, for all $i = 1 \ldots m$, so that we can categorize it as a lower or upper bound. We will always be able to do this as discussed and proved in Section 2.5.

In step (2), the redundant lower bounds are eliminated. Intuitively, if a lower bound x is always greater than another lower bound y, for all values of $\vec{z}$, and the parameters $\vec{p}$ and $\vec{s}$, then we can eliminate x, since it will never be the binding one. Note that we are only doing a *local* check for redundancy, i.e., within the lower bounds in L_n. In a similar fashion, the redundant upper bounds in U_n are also eliminated. Redundancy elimination is further discussed in Section 2.7.

In step (3), pairs of rows, (l_a, u_b) such that $l_a \in L_n$ and $u_b \in U_n$ are considered. The first goal is to eliminate the z_n components from l_a and u_b. To achieve this, we seek to scale $l_{a,n}$ and $u_{b,n}$ appropriately by some factor, so that when the two rows are later added the n-th component will cancel out. Since, the coefficient of $l_{a,n}$ and $u_{b,n}$ could be either a rational number or a linear function, computing the appropriate scaling factor is little involved, and steps are outlined below. However, note that there always exists a scaling factor that can be used to cancel out the z_n components in $l_{a,n}$ and $u_{b,n}$.

If z_n is an index from $\vec{i}$ (an element loop index) then $l_{a,n}$ and $u_{b,n}$ are just rational numbers, but if z_n is an index from $\vec{t}$ (an tile loop index) then $l_{a,n} = \alpha_{a,n} \times s_k$ and $u_{b,n} = \alpha_{b,n} \times s_k$ are linear functions of some tile size variable s_k. To compute the scale factors we extract the coefficients of $l_{a,n}$ and $u_{b,n}$ and assign them to β_l and β_u, respectively. β_l is equal to $|l_{a,n}|$ if $l_{a,n}$ is a rational number, otherwise, it is equal of $|\alpha_{a,n}|$, the coefficient of the linear function $l_{a,n}$. The actual scale factor for l_a and u_b are respectively, $\frac{\beta_u}{g}$ and $\frac{\beta_l}{g}$, where $g = \gcd(\beta_l, \beta_u)$. Steps ($3.a - 3.c$) compute these scale factors. In step ($3.d$), the rows l_a and u_b are scaled and added together to obtain a new row x, in which the z_n component is guaranteed to be 0.

In step ($3.e$), the new row x checked for redundancy. Here we check whether the row x corresponds to a constraint on the parameters, for example, $s_1 \geq 1$. The motivation behind this check is that often comparisons of lower and upper bounds (l_a and u_b) result in cancellation of all the components leading to a a constraint on the parameters. This type of redundancy elimination

is further discussed in Section 2.7.

In step (4), the rows, R_n, that did not contribute towards a lower or upper bound of z_n, are added to Γ'.

2.4 Complexity of the SFME Algorithm

Since we can determine the signs of the coefficients (step (1)) we do not have to maintain a tree of sign distinguishing cases. Hence, the worst case time complexity of SFME is the same as the standard FME algorithm for linear constraints, viz., doubly exponential on the number of constraints. However, for the kind of problems encountered in loop transformations, FME has been used very successfully by many research and production compilers. With regards to the space complexity, the standard FME for linear constraints uses matrices with rational elements, but ours uses matrices with symbolic elements. Hence, SFME would require more space.

We check for redundant constraints at every step of elimination and eliminate as many as possible. We have observed that this interlacing of redundancy check with every elimination step substantially improves both the running time and memory space, since less the number of constraints, lesser the time and space required.

2.5 Sign determination always possible

Determination of the signs of the coefficients (entries of Γ) is required (in step (1) of SFME) to categorize the constraints into lower and upper bounds of a given variable. We now show that we can always determine the sign of the coefficients and hence categorize the constraints. Observe that whenever BLIS-PROPERTY is satisfied, we can always determine the sign of the coefficients, i.e., entries of Γ. In fact, we will show that BLIS-PROPERTY is an invariant that is maintained by the SFME algorithm, whenever its input system Γ satisfies the property. Let us formally state and prove the invariant.

Theorem 2.5.1. *If the input matrix Γ has the* BLIS-PROPERTY, *i.e., every column of it either exclusively contains rational numbers or linear functions of a single tile variable, then the output matrix Γ' returned by* SFME *will also satisfy the same property.*

Proof. We prove by showing that the new rows (constraints) created by SFME posses the BLIS-PROPERTY. Now, consider a row x, created by SFME on step $(3.d)$ with

$$x \leftarrow \frac{\beta_u}{g} \times l_a + \frac{\beta_l}{g} \times u_b.$$

The operations used are multiplication by a constant $\frac{\beta_l}{g}$ (or $\frac{\beta_u}{g}$) and component-wise addition of the rows l_a and u_b. Both the operations will preserve the type of the components of the rows, since both rational numbers and linear functions are closed under scalar multiplication and addition. We also need to show that if the i-th component of l_a and u_b were functions of a tile variable s_y, then we the i-th component of x is also a function of the same tile variable s_y. This is indeed the case, since addition of two linear functions of a single variable w, produces another expression linear in the same variable, w. □

The first call to SFME is with the constraints that define the PTIS (c.f. Eqns. 2.1, 2.3). BLIS-PROPERTY shows that PTIS satisfies the invariant mentioned above. Successive calls to SFME are with the outputs of previous calls to itself, which are guaranteed to have the property.

2.6 Loop generation from computed bounds

For every variable $z_k, \forall k = 1 \ldots 2n$, we have lower and upper bounds, L_k and U_k, computed with the SFME algorithm. We generate loop lower bounds of a index variable z_k by taking the maxima of all its lower bounds, i.e.,

$$LB_k = \max(l_1, l_2, \ldots, l_{|L_k|})$$

where $l_x \in L_k, \forall x = 1 \ldots |L_k|$ and $|L_k|$ denotes the total number of lower bounds of z_k. In a similar fashion, we generate the upper bounds of an index variable z_k by taking the minima of all its upper bounds, i.e.,

$$UB_k = \min(u_1, u_2, \ldots u_{|U_k|})$$

where $u_x \in U_k, \forall x = 1 \ldots |U_k|$ and $|U_k|$ denotes the total number of upper bounds of z_k. Some of the upper (lower) bounds might contain divisions by tile sizes, for such upper (lower) bounds we use the floor (ceiling) functions to round them to integers.

2.7 Redundancy elimination

The criteria for redundancy is simple: *A constraint λ is redundant in a system of constraints Λ if λ is implied by $\Lambda - \{\lambda\}$*. The computationally expensive but sure way of performing this check is to use the criteria: *λ is redundant in Λ if and only if the conjunction of the negation of λ with $\Lambda - \{\lambda\}$ is infeasible.* Such a feasibility test can be done by using SFME itself – recall the FME-PROPERTY 2 from Section 2.3.

A naive way would be to first apply SFME to compute the lower and upper bounds of all the variables, and then check for redundancy of the constraints in this set. In such a method, we carry the redundant constraints produced in every step all the way till the end. Due to the nature of FME method, the redundant constraints at any step gets compared with other constraints, resulting in a larger and larger set of redundant constraints. Though this method would detect all the redundant constraints, due to the huge number of constraints, it is very expensive both in terms of time and memory requirements.

In SFME, to avoid this explosion of constraints, we interleave the elimination steps with local redundancy checks (c.f. steps (2,3.*e*)). These local redundancy checks act as filters and do not necessarily detect all redundant constraints. However, we found them to very effective in removing almost 80% of the redundant constraints. The redundancy check performed in step (2), considers the given set of lower bounds, L_n, in the context of the constraints on program and tile size parameters, C, to see whether some bounds are redundant. The criteria used here is: *if a lower bound x is smaller than another lower bound y, for all values of the index variables, and parameters from C, then x is redundant, since it will never be the binding constraint.* A similar criteria is used to check redundancy of upper bounds in U_n.

For example, during the elimination of i_2 from the PTIS of $\mathbf{T}_{tri}$, we get two lower bounds for $i_2 : \{1, s_2 \times t_2 + 1\}$. By applying the criteria described above, with the knowledge that $t_2 \geq 0$, and $s_2 \geq 1$ from C, we can conclude that $s_2 \times t_2 + 1 \geq 1$, for all values of s_2 and t_2. Hence, we can eliminate 1 form the lower bounds. For a more involved example, consider the following two

upper bounds (encountered during the bounds computation of the 3D stencil example):

$$\begin{aligned} u_1 &\equiv 2s_i \times t_i + 2s_i + N_j + N_i - 3 \\ u_2 &\equiv 2N_j + 2s_i + 2s_i t_i + s_j - 5 \end{aligned}$$

We want the smaller upper bound, and hence would like to check whether $u_1 \leq u_2$, for all values of s_i, s_j, N_j, N_i, and t_i. Observe that $u_1 \leq u_2 \iff 0 \leq N_j + s_j - 2$, but from the constraints of parameters N_j and s_j (given in C) we know that $N_j, s_j \geq 1$. With this knowledge we can easily infer that $0 \leq N_j + s_j - 2$, and hence prove that $u_1 \leq u_2$ for all the values of s_i, s_j, N_j, N_i, and t_i. An important feature of this kind of redundancy elimination is that the question whether an upper bound u_2 is redundant with respect to another upper bound u_1 is reduced to a question on the constraints on the parameters, i.e., N_j and s_j here. Since, the constraints on the parameters are just linear, we can check these constraints very efficiently.

The redundancy check performed at step (3.*e*), considers the situation in which a new constraint obtained by comparing a lower and a upper bound, is redundant with respect to the constraints on the parameters, C. In practice, we have found this check to be very effective. For example, consider the following pair of lower and upper bounds (encountered during the bounds computation of the 3D stencil example discussed in):

$$\begin{aligned} l &\equiv s_j \times t_j + 4 - N_j - i_k \\ u &\equiv s_j \times t_j + s_j + 2 - i_k. \end{aligned}$$

When we compare $l \leq u$, we get $2 \leq N_j + s_j$. We first note that the constraints do not involve any index variable and are on the parameters, N_j and s_j, only. Hence, we do not add it to the result Γ' in SFME (step (3.*e*)). Further, this constraint is implied by the constraints $N_j, s_j \geq 1$ in C. Hence, we can throw away this constraint. Again, checking whether such a constraint on the parameters is implied by the constraints in C can be done very efficiently since these are just linear constraints.

We use the aforementioned local redundancy checks as filters during every elimination step. We use the global redundancy check (discussed above – negate a constraint and check for the

feasibility of it with the rest of the constraints) on the final set of constraints we obtain with lower and upper bounds of all the variables.

2.8 Related Work

There has been relatively little work for the case where tile sizes are symbolic parameters, except for the very simple case of *orthogonal* tiling: either rectangular loops tiled with rectangular tiles, or loops that can be easily transformed to this. For the more general case, the standard solution, as described in Xue's text [136] has been to simply *extend* the iteration space to a rectangular one (i.e., to consider its bounding box), apply the orthogonal technique with appropriate guards to avoid computations outside the original iteration space.

Amarasinghe and Lam [7, 8] implemented, in the SUIF tool set, a version of FME that can deal with a limited class of symbolic coefficients (parameters and/or block sizes), but the full details have not been made available.

Größlinger et al. [53] proposed an extension to the polyhedral model, in which they allow arbitrary rational polynomials as coefficients in the linear constraints that define the iteration space. Their genericity comes at the price of requiring computationally expensive machinery like quantifier elimination in polynomials over the real algebra, to simplify constraints that arise during loop generations. Due to this their method does not scale with the number of dimensions and the number of non-linear parameters.

2.9 Discussion

The concepts presented in this chapter form the mathematical foundation for parameterized tiling. The definition of parameterized tiling as a transformation, its bilinear property, and the SFME algorithm, together provide a foundation for the extension of the loop transformations to the case where the tile sizes are not fixed. For example, what is the result of applying a linear transformation such as skewing to the PTIS? Does the resulting PTIS still have the bilinear property which would allow the use of SFME? We conjecture that PTIS is closed under linear transformations, i.e., the result of a linear transformation of the PTIS with BLIS property is another PTIS which also has the BLIS property.

The SFME algorithm inherits the super-exponential complexity of the FME.. Further, it also requires symbolic arithmetic during its elimination steps. Due to this, we believe that the outset based methods (presented in the next two chapters) for parameterized tiled loop generation are more efficient than SFME.

CHAPTER 3

Parameterized Tiled Loop Generation

> "By no longer requiring the effect of an optimization to persist indefinitely, we can allow executables adapt to changes in their usage and environment. [...] this view helps us to regain the original promise of software—that it is flexible and easy to change. "
>
> —*Overcoming the challenges to feedback-directed optimization*, Michael Smith [118]

TILED loop generation involves the generation of the tiled loop bounds. In this chapter we describe a technique for generating tiled loops which can be used for when the tile sizes are fixed, compile time constants or not fixed but left as symbolic parameters. We first discuss the structure of tiled loops and then present our method for tiled loop generation. The efficiency of our technique is demonstrated via experimental evaluation on kernels from linear algebra computations from BLAS3 and stencil computations. Our technique provides parameterized tiled loops for free in the sense that it takes a comparable amount of time to generate the loops and the quality of the generated code is comparable, if not better. We then discuss a technique for separating partial (boundary) tiles from full (interior) tiles—an enabling step for optimizations such as register tiling. Finally, we discuss related work.

The work presented in this chapter was done in collaboration with DaeGon Kim and Michelle

```
for (k = 1; k <= N_k; k++)
  for (i = k+1; i <= k+N_i; i++)
        S1(k,i);
```

Figure 3.1.
2D iteration space found commonly in stencil computations. The body of the loop is represented with the macro S1 for brevity.

Mills Strout. It was presented in [104].

3.1 Anatomy of Tiled Loop Nests

Tiling is an iteration reordering transformation that transforms a d-depth loop nest into one of depth up to $2d$. In this section we study the structure of tiled loops and develop an intuition for the concepts involved in generating them. In later sections, these concepts are formalized and used in deriving a simple and efficient algorithm for the generation of tiled loops.

Consider the iteration space of a 2D parallelogram such as the one shown in Figure 3.1, which is commonly found in stencil computations [78]. Figure 3.2 shows a geometric view of the iteration space superimposed with a 2×2 rectangular tiling. Observe that there are three types of tiles: *full*—which are completely contained in the iteration space, *partial*—which have a partial, non-empty intersection with the iteration space, and *empty*—which do not intersect the iteration space. The lexicographically earliest point in a tile is called its *origin.* The goal is to generate a set of loops that *scans* (i.e., visits) each integer point in the original iteration space, based on the tiling transformation, where the tiles are visited lexicographically, and then the points within each tile are themselves visited lexicographically. We can view the four loops that scan the tiled iteration space as two sets of loops each, where the first set of two loops enumerates the tile origins and the next set of two loops visits every point within a tile. We call the loops that enumerate the tile origins the *tile-loops* and those that enumerate the points within a tile the *point-loops.*

3.1.1 Bounding Box Method

One solution for generating the tile-loops is to have them enumerate every tile origin in the bounding box of the iteration space and push the responsibility of checking whether a tile con-

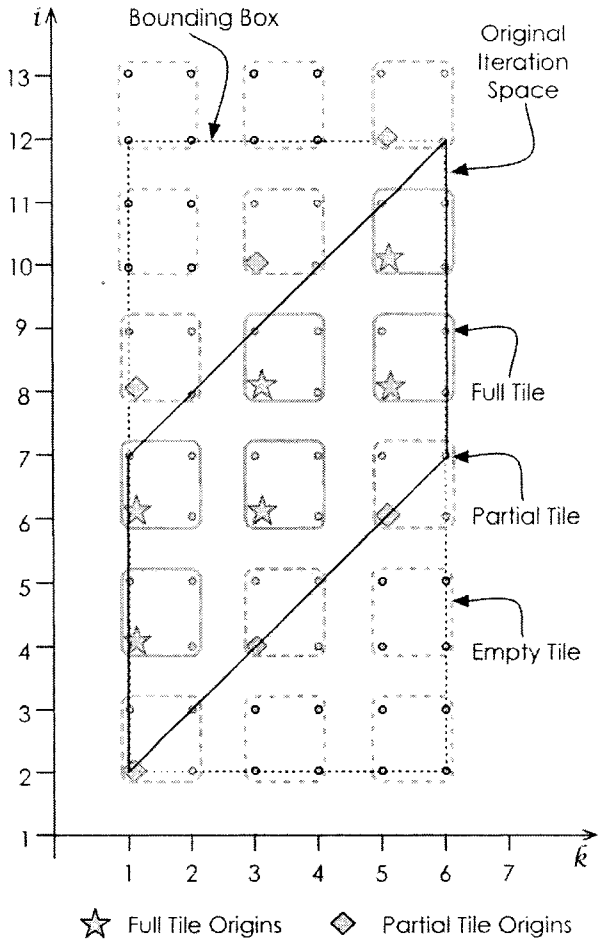


Figure 3.2.
A 2×2 rectangular tiling of the 2D stencil iteration space with $N_i = N_k = 6$ is shown. The bounding box of the iteration space together with full, partial, and empty tiles and their origins are also shown.

```
for (kT = 1; kT <= Nk; kT += Sk)
 for (iT = 2; iT <= Ni+Nk; iT += Si)
  for (k= max(kT,1);k<=min(kT+Sk-1,Nk);k++)
   for (i= max(iT,k+1);i<=min(iT+Si-1,k+Ni);i++)
        S1(k,i);
```

Figure 3.3. Tiled loops generated using the bounding box scheme.

tains any valid iteration to the point-loops. The tiled loop nest generated with this bounding box scheme is shown in Figure 3.3. The first two loops (`kT` and `iT`) enumerate all the tile origins in a bounding box of size $N_k \times (N_i + N_k)$ and the two inner loops (`k` and `i`) scan the points within a tile. A closer look at the point-loop bounds reveals its simple structure. One set of bounds are from what we refer to as the *tile box bounds,* which restrict the loop variable to points within a tile. The other set of bounds restricts the loop variable to points within the iteration space. Combining these two sets of bounds we get the point loops that scan points within the iteration space and tiles. Geometrically, the point loop bounds correspond to the intersection of the tile box (or rectangle) and the iteration space, here the parallelogram in Figure 3.2.

The bounding box scheme provides a couple of important insights into the tiled loop generation problem. First, the problem can be decomposed into two independent problems: generation of tile-loops and the generation of point-loops. Such a decomposition leads to efficient loop generation, since the time and space complexity of loop generation techniques is doubly exponential in the number of bounds. The second insight is the scheme of combining the tile box bounds and iteration space bounds to generate point-loops. Another important feature of the bounding box scheme is that tile sizes need not be fixed at loop generation time, but can be left as symbolic parameters. This feature enables generation of *parameterized tiled loops,* which has many applications as discussed in Chapter 1. However, the bounding box scheme can suffer from inefficiency in the generated loops in that the tile-loops can enumerate many empty tiles.

3.1.2 When Tile Sizes Are Fixed

When the tile sizes can be fixed at the loop generation time an *exact* tiled-loop nest can be generated. Tile-loops that only enumerate origins of tiles that have a non-empty rational intersection with the iteration space are exact. Ancourt and Irigoin [9] proposed the first and now classic

```
for(kT=0;kT<=⌊(Nk/2)⌋;kT++)
 for(iT=max(1,kT); iT<=min(⌊(2*kT+Ni+1)/2⌋,⌊Nk+Ni/2⌋);iT++)
  for(k=max(max(1,2*kT),2*iT-Ni); k<=min(min(2*kT+1,2*iT),Nk);k++)
   for(i=max(2*iT,k+1); i<=min(2*iT+1,k+Ni);i++)
          S1(k,i) ;
```

Figure 3.4. Tiled loops generated for fixed tile sizes using the classic scheme.

solution for generating the exact tiled loops when the tile sizes are fixed. In this case the tiled iteration space can be described as a set of linear constraints and the loops that scan this set can be generated using Fourier-Motzkin elimination [9, 136]. The exact tiled loop nest for the 2D stencil example is shown in Figure 3.4. Note that the efficiency due to the exactness of the tile-loops has come at the cost of fixing the tile sizes at generation time. Such loops are called *fixed tiled loops.*

The classic scheme, in addition to requiring fixed tile sizes, also suffers from loop generation inefficiency. It takes as input all the constraints that describe the bounds of the $2d$ loops of the tiled iteration space, where d is the depth of the original loop nest. Since the method is doubly exponential on the number of constraints, this increased number of constraints might lead to situations where the loop generation time may become prohibitively expensive [51].

Goumas et al. [51] improve on the classic scheme by dividing the loop generation problem into two subproblems, similar to the approach taken with bounding box, but their generated code visits fewer empty tiles than bounding box. However, their solution is still only applicable to fixed tile sizes.

3.1.3 Best Of Both

We propose a tiled code generation method that achieves the best of both worlds: the simple decomposed loop structure used by the bounding box method and the Goumas et al. technique, the code quality provided by the fixed tile size methods, and the benefits of parameterized tile sizes provided by the bounding box method. We develop the necessary theory and use it to derive a method which provides efficient generation of efficient parameterized tiled loops.

The key insight is the construction of a set called the *outset*, which contains all possible tile origins for non-empty tiles. The *outset* is similar to the Tile Origin Space (TOS) constructed by Goumas et al. [51], but there are two important differences. First, the outset we construct

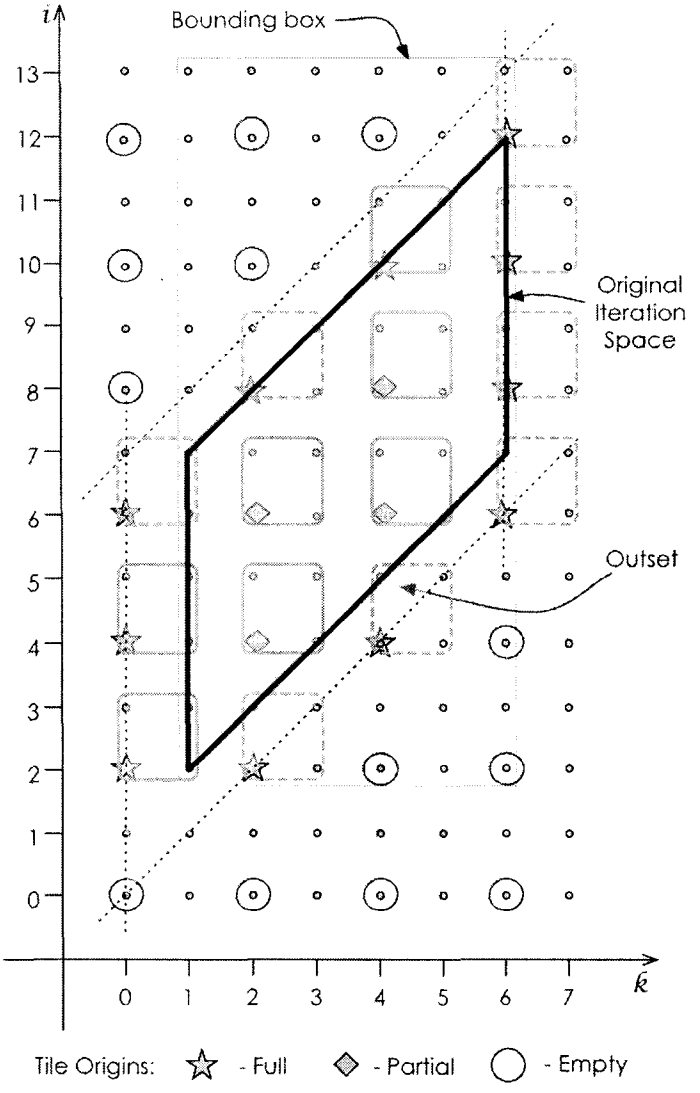


Figure 3.5.
A 2 × 2 rectangular tiling of the 2D stencil iteration space with $N_i = N_j = 6$. The outset and bounding box are also shown. Compare the number of empty tile origins contained in each of them.

```
kTLB = -Sk+2; kTLB = ⌈kTLB/Sk⌉*Sk;
for(kT = kTLB; kT <= Nk; kT += Sk)
 iTLB = kT-Si+2; iTLB = ⌈iTLB/Si⌉*Si;
 for(iT = iTLB; iT <= kT+Ni+Sk-1; iT += Si)
  for(k= max(kT,1);k<=min(kT+Sk-1,Nk);k++)
   for(i=max(iT,k+1);i<=min(iT+Si-1,k+Ni);i++)
          S1(k,i);
```

Figure 3.6.
Parameterized tiled loops generated using outset. The variables `kTLB` and `iTLB` are used to shift the first iteration of the loop so that it is a tile origin, and explained later (Section 3.2.2.2).

includes the tiles sizes as parameters. Second, we feed the outset to any code generator capable of scanning polyhedra, and then simply post-process the resulting code to add a step size and shift the lower bounds of the tile loops. Goumas et al. generate tile loops that iterate over the image of the TOS after applying tiling, and this is expensive.

The outset has all the benefits of a bounding box, but enumerates very few empty tiles. In general, it is parameterized by the tile size, but for illustration purposes Figure 3.5 shows the outset instantiated for the 2D stencil example and 2×2 tiles. In this example, the outset includes only one empty tile origin at (0,0), far fewer than the number of empty tiles that the bounding box includes.

Geometrically, the outset construction can be viewed as shifting of the hyper-planes that define the lower bounds of the loops. For our 2D example, we shift the left vertical line and the two 45 degree lines, where the left vertical line and the top 45 degree line constitute the lower bound of k, and the bottom 45 degree line forms the lower bound for i. These lines are shifted out by a distance that ensures that they will contain the origin of any tile which has a non-empty intersection with the iteration space, i.e., any tile that would contain a valid iteration point. Loops that scan the outset are post-processed and then used as the tile-loops. The tiled loops generated by scanning the outset are shown in Figure 3.6.

The outset has several important properties. It can be constructed without fixing the tile sizes, hence can be used for generating parameterized tiled loops. Second, it can be constructed very efficiently—in time and space linear in the number of loop bounds. In comparison, automatic construction of the bounding box is more expensive—we are not aware of any linear time algorithm that constructs a bounding box given the constraints that define an iteration space. Third, the outset can be used to decompose tiled loop generation into separate tile-loop and point-loop

generation. Fourth, it can be used efficiently in cases when the tile sizes are fixed, parameterized or mixed, i.e., some are fixed and some are left as parameters. These properties lead to a single simple efficient algorithm for both parameterized as well as fixed tiled loop generation. The following sections discusses these properties in more detail.

3.2 Generating the Tile-Loops with Outset

In this section, we describe our method for generating the tile-loops. We first formally define the set that contains all the non-empty tile origins and then motivate an approximation of this set which can be computed efficiently. We then reduce the problem of generating tile-loops to one of generating loops that scans the intersection of the outset polyhedron and a parameterized lattice. We describe a single method that can be used to generate tile-loops for both fixed as well as parameterized tile sizes.

Our input model is perfectly nested loops. Our techniques are applicable to cases where rectangular tiling is valid or can be made valid by any loop transformation, which we assume has been done in a preprocessing step. Many important applications contain loops of this kind.

3.2.1 The Outset and its Approximation

For correctness, tiled code should visit all the tiles that contain points in the original iteration space. To generate the tile loops separately from the point loops, we visit all tile origins within a polyhedron we call the outset. The *outset* includes all possible tile origins where the tile for that tile origin includes at least one point from the original iteration space.

The original loop is represented as a set of inequalities

$$P_{iter} = \{\vec{z} \mid Q\vec{z} \geq (\vec{q} + B\vec{p})\},$$

where z is the iteration vector of size d, Q is a $m \times d$ matrix, each row of which defines a constraint on the iteration space, $\vec{q}$ is a constant vector of size m, $\vec{p}$ is a vector of size n containing symbolic parameters for the iteration space, and B is a $m \times n$ matrix. The tile sizes are represented by a vector $\vec{s}$, where for $i = 1 \ldots d$, s_i indicates the size of the tile in dimension i.

The outset polyhedron is defined as the set of points in the original iteration space that, if they were tile origins, would define a tile that includes at least one point in the original iteration space. Formally, let $tile(\vec{x})$ specify the set of points that belong to the tile whose origin is $\vec{x}$,

$$tile(\vec{x}) = \{\vec{z} \mid \vec{x} \leq \vec{z} \leq \vec{x} + \vec{s'}\},$$

where $\vec{s'} = \vec{s} - \vec{1}$ with $\vec{1}$ being a size d vector containing all ones. The true outset is

$$P_{out} = \{\vec{x} \mid tile(\vec{x}) \cap P_{iter} \neq \emptyset\}.$$

P_{out} as defined above is a union of all tiles whose intersection with P_{iter} is non-empty. Computing this set explicitly is very expensive. So, we derive a reasonably tight approximation of P_{out} that is a single polyhedron and can be directly computed from the constraints of P_{iter}. We denote this approximation by $\widehat{P_{out}}$. As a comparison, one could also view the bounding box as a very loose approximation of P_{out}. $\widehat{P_{out}}$ can be computed in time and space linear in the number of constraints in P_{iter}. Henceforth we call $\widehat{P_{out}}$ the *outset.* The outset discussed in previous sections also refers to $\widehat{P_{out}}$.

We compute the outset, $\widehat{P_{out}}$, by shifting all the lower bounds of the original iteration space along the normal that faces out of the iteration space. The outset is defined as

$$\widehat{P_{out}} = \{\vec{x} \mid Q\vec{x} \geq (\vec{q} + B\vec{p}) - Q^{+}\vec{s'}\},$$

where Q^{+} is a $m \times d$ matrix defined as follows:

$$Q^{+}_{ij} = \begin{cases} Q_{ij}, & \text{if } Q_{ij} \geq 0 \\ 0, & \text{if } Q_{ij} < 0 \end{cases}$$

Note that the $\widehat{P_{out}}$ is defined using the constraint matrix, Q of the iteration space polyhedron. We can compute Q^{+} with a single pass over the entries of Q and hence in time linear on the number of constraints of P_{iter}. We now formally prove that $\widehat{P_{out}}$ contains all the non-empty tile origins.

Theorem 3.2.1. $P_{out} \subseteq \widehat{P_{out}}$.

Proof:

If a point $\vec{x}$ is in P_{out}, then there exists a point $\vec{z}$ such that $\vec{z}$ is in P_{iter}, $\vec{z}$ is in $tile(\vec{x})$, and $\vec{z} = \vec{x} + \vec{i}$, where $\vec{0} \leq \vec{i} \leq \vec{s'}$. Since $\vec{z}$ is in P_{iter}, the following is true:

$$Q\vec{z} \geq (\vec{q} + B\vec{p}).$$

Substituting $\vec{z}$ by $\vec{x} + \vec{i}$, we derive the following:

$$Q\vec{x} + Q\vec{i} \geq (\vec{q} + B\vec{p}), \text{ for } \vec{0} \leq \vec{i} \leq \vec{s'}.$$

Since all the entries in $\vec{i}$ are non-negative and the fact that $Q^+ \geq Q$, it follows that $Q^+\vec{s'} \geq Qi$, and so the point $\vec{x}$ is also in $\widehat{P_{out}}$:

$$Q\vec{x} + Q^+\vec{s'} \geq (\vec{q} + B\vec{p}).$$

Thus, each point that is in P_{out} is also in $\widehat{P_{out}}$. ∎

Notice that though the tile sizes are not fixed and are included as parameters, the outset is still a polyhedron, albeit parameterized by the tile sizes. This key property enables us to generate parameterized tile-loops, for now we can use all the theory and tools developed for generating loops that scan parameterized polyhedra.

3.2.2 Generating tile-loops

The tile-loops enumerate the tile origins. Two choices are available: (i) enumerate the tile origins as coordinates in the tile space or (ii) enumerate the tile origins in the coordinates of the original iteration space. When the former is chosen, we need additional transformations to map the tile origins from the tile space to tile origins in the iteration space coordinates. Our method avoids this transformation and generates loops that directly enumerate the tile origins in the original iteration space coordinates.

We can view the set of tile origins as the points in a lattice whose period is the tile sizes. We

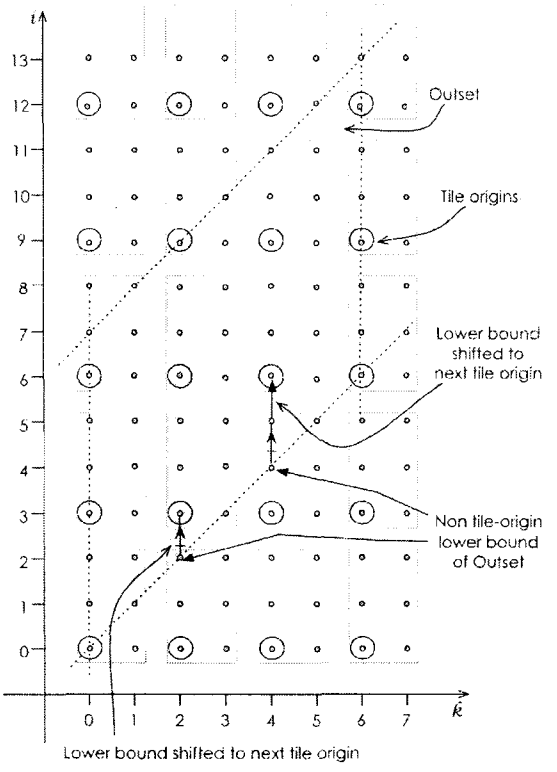


Figure 3.7.
Intersection of a tile origin lattice for 2 × 3 tiles and the outset is shown. The original iteration space is omitted for ease of illustration. Note that the first iteration of the loops that scans the outset could be a non-tile origin. We need to shift this iteration to the next iteration that is tile origin.

define the *tile origin lattice*, $\mathscr{L}(\vec{s})$, as the lattice whose period is given by the symbolic tile size vector $\vec{s}$. Since we do not fix the tile sizes, $\mathscr{L}(\vec{s})$ is actually a *parameterized tile origin lattice.* We also do not require that the tile origin lattice start at any particular coordinate.

The outset contains all the non-empty tile origins and also other points which are not tile origins. Formally, we want to visit the points in the intersection of the outset and the tile origin lattice, i.e., $\widehat{P_{out}} \cap \mathscr{L}(\vec{s})$. The key insight is to generate loops that scans the whole of outset and modify them so that they skip the iterations that are not tile origins.

3.2.2.1 Striding the loops

Figure 3.7 shows an outset and a tile origin lattice for a 2 × 3 tiling. Let us call the loops that scan all the integer points in the outset as *outset-loops.* For the moment assume that the first iteration

of every loop is aligned with a tile origin. Then we can skip the non-tile origins by just adding a *stride* to the loop variable with the corresponding tile size parameter. Such an addition of strides is sufficient because we are interested in the canonical tile lattice (induced by the rectangular tiling). This simple post-processing of the loops that scans the outset gives us the loops that scans the intersection of outset and tile origin lattice. Note that the stride can be a fixed constant or a symbolic parameter. This allows us to use the same method for generating tile loops for both fixed and parameterized tile sizes.

3.2.2.2 Shifting Lower Bounds

We now address the issue of aligning the first iteration of the outset-loops to a tile origin. Figure 3.7 shows two non-tile origins that correspond to first iterations of the i loop. We need to shift the lower bound to an iteration that corresponds to the next tile origin. Let LB_i be the lower bound of a loop variable i. Note that LB_i could be a function of the outer loop indices and parameters. The required shift can be thought of as the difference between the value of LB_i and the next tile origin. This shift can be computed as $\left\lceil \frac{LB_i}{s_i} \right\rceil \times s_i$. Since this shift can be generated for fixed as well as parameterized tile sizes, we have a single method for both fixed and parameterized tiled loop nest generation.

The code previously presented in Figure 3.6 showing the parameterized tiled loops for the 2D stencil example (Figure 3.1) was generated using the scheme described above. Note how the skipping of the non-tile origins naturally translates into parametric strides of the loop variables. Also note how the lower bound shifts can be expressed as loop variable initializations.

3.2.2.3 Implementation

Our code generator takes as input the constraints that define P_{iter}. It constructs the outset ($\widehat{P_{out}}$), which is parameterized by the program and tile parameters. The outset-loops are generated using a standard loop generator for parameterized polyhedra. Thanks to our theory, all that is required to turn them into tile loops is a simple post-processing, actually pretty-printing, to add strides and lower bound shifts. These tile loops are then composed with the point-loops whose generation is described in the next section.

3.3 Generating the Point Loops

The point-loops make up a loop nest that enumerates all the points within a tile. To ensure that they scan points only in the original iteration space, their bounds are composed of tile bounds as well as iteration space bounds. When the point-loops are generated separately, the tile origin is not known.

Consider the triangular iteration space shown in Figure 3.8. Essentially, the intersection of a tile (without fixing the tile origin) and the iteration space is the set of points to be scanned by point-loops. To generate them, we can construct the intersection that is now parameterized by both program parameters and tile origin index. This approach does, however, increase the number of dimensions, which is a major factor at code generation time.

Since the tile bounds for rectangular tiling are simple, we can optimize the generation of the point loops. We first construct a loop nest that scans the original iteration space. Then for each lower bound lb_i, we add the tile lower bound, tlb_i to produce the point-loop lower bound $max(lb_i, tlb_i$. Similarly, for each upper bound ub_i, we add the tile upper bound $tlb_i + s_i - 1$ (s_i is the tile size of i-th dimension) to produce the point-loop upper bound $min(ub_i, tlb_i + s_i - 1)$. The point-loops for the example in Figure 3.8 are given below, with `iT` and `jT` representing the tile origin indices, and `Si` and `Sj` representing the sizes of the tiles along the i and j dimensions.

```
for i=max(1,iT) to min(N,iT+Si-1)
   for j=max(1,jT) to min(i,jT+Sj-1)
      body;
```

In addition, we can also generate simple point loops where iteration space bounds are not included. As shown in Figure 3.8, if a tile is a full tile, i.e., a subset of the iteration space, then the bounds for the original iteration space are not necessary. Such simple point loops are useful for the optimization described in Section 3.5.

3.4 Implementation and Experimental Results

We implemented four different tiled loop generators: two for fixed tile sizes and two for parameterized tile sizes. The loop generators are available as open source software [55]. For fixed-size

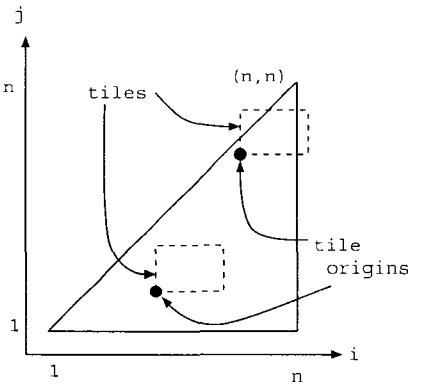


Figure 3.8.
A triangular iteration space and tiles

tiles, we implement the classic and decomposed methods. For the classic method, the constraints that represent the tiled iteration space are constructed from the original loop bounds and then fed to CLOOG [14] to generate the tiled loops. For the decomposed method, we construct an outset with fixed tile sizes and use them to generate tile-loops and generate the point loops separately as discussed in the previous sections. For parameterized tiled code generation, we implement the parameterized decomposed method presented earlier in this chapter and the bounding box method. For the bounding box method, we assume that the bounding box is provided as an input. The bounding box is used in the place of outset to generate tile-loops and the parameterized point loops are generated as in the fixed methods except the tile sizes are now symbolic parameters for the point loops. For the parameterized decomposed method, we first generate the outset from the input loop bounds and use it to generate the tile-loops. We then generate the parameterized point loops and embed them in the tile-loops to get the final tiled loop nest.

The experiments compare the various loop generating techniques in terms of the quality of the generated tile code and the efficiency of the tiled loop generation. Both of these measures depend heavily on the underlying code generator, because the techniques presented in this chapter enable the implementation of parameterized decomposed tiling to use any loop generator capable of generating loops that scan a polyhedron as a black box. For generating the loops that scan a polyhedron we use the CLOOG loop generator, which has been shown to quickly generate high quality loops [14]. However, it is possible to replace the CLOOG generator with a different code generator such as the Omega code generator [70].

	Description	Loop depth/ # tiled loops
SSYRK	Symmetric Rank k Update.	3 / 2
LUD	LU decomposition of a matrix without pivoting.	3 / 2
STRMM	Triangular matrix multiplication.	3 / 2
3D Stencil	Gauss-Seidel Style 2D/3D stencil computation.	3 / 3

Table 3.1. Benchmarks used for code quality evaluation.

3.4.1 Experimental Setup

To evaluate the quality of the generated code, we use linear algebra computation kernels from BLAS3 and a stencil computation, as listed in Table 3.1. The stencil computation has a 3D iteration space, and operates on two dimensions of data. It is necessary to skew the stencil computation before applying tiling. Column 3 in Table 3.1 indicates the loop depth of the original loop, and the number of loops that are tiled.

We ran the experiments on an Intel Core2 Duo processor running at 1.86 GHz with an L2 cache of size 2MB. The system is running SMP Linux. For compiling our tiled loop nests we used `g++` version 4.1.1. with the highest optimization level (`-O3`). The timings use gettimeofday().

3.4.2 Results

For each combination of benchmark and implemented tiled code generation method, we time an approximation of loop overhead, the total run of the tiled benchmark, and the time required to generate the tiled code.

Figure 3.9 shows the loop overhead for SSYRK (symmetric rank k update) as a percentage of the total loop execution time. We time the execution time of the tiled loop bounds with only a counter as the body and divide the measured execution time by the execution time for the loop with the full body including the loop counter. The loop overhead is only approximate, because in the loop with the full loop body some of the loop bound instructions can be scheduled with instructions from the body, therefore this measure is an upper bound on the loop overhead. The approximate loop overhead on average can be as high as 40%. Figure 3.10 shows the total execution

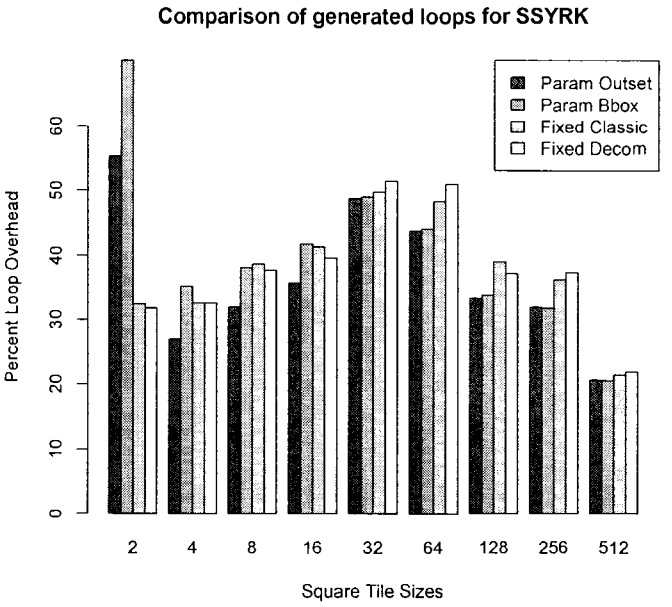


Figure 3.9.
Percentage loop overhead =(counter / body and counter)×100 of the SSYRK for matrices of size 3000 × 3000.

time for the SSYRK as the tile sizes vary. Notice that as the tile sizes become large enough to result in improved performance of the overall loop, the approximate percentage of time spent on loop overhead increases.

Figures 3.10-3.13 show the total execution time for the various benchmarks as the tile size varies. The cache effect that occurs as the tile size better uses cache can most clearly be seen for STRMM, 3D Stencil, and SSYRK. In general, the quality of the generated tiled code is comparable. The outliers occur at smaller tile sizes, where the parameterized tiled code generator based on bounding box significantly increases the running time for all benchmarks. For cache tiling, the smaller tile sizes do not experience the best performance improvement; however, smaller tile sizes are critical for register tiling [64]. Our parameterized decomposed method performs much better than bounding box at smaller tile sizes.

We also performed the same set of experiments on an AMD Opteron dual core processor running at 2.4 GHz with a cache of size 1MB, and obtained similar results as presented here.

The compilation time (the average, in milliseconds over five runs for each benchmark) for the four tiled loop generation methods, viz., fixed classic, fixed decomposed, parameterized bounding

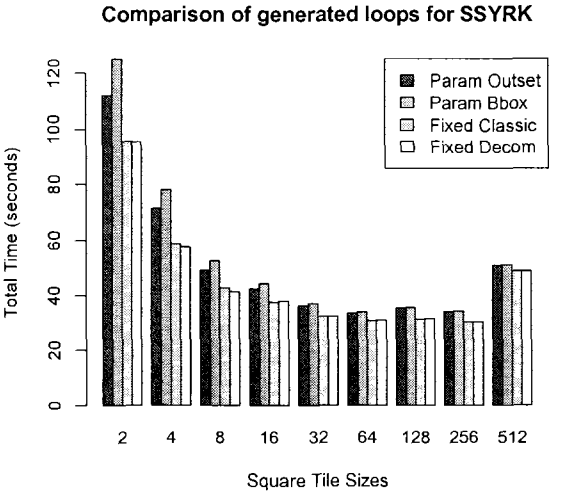


Figure 3.10.
Total execution time for symmetric rank k update for matrices of size 3000 × 3000.

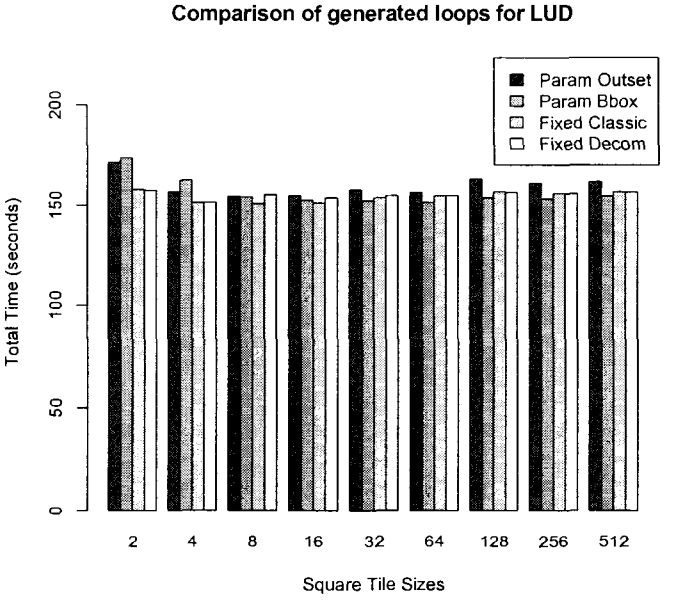


Figure 3.11.
Total execution time for LUD on a matrix of size 3000 × 3000.

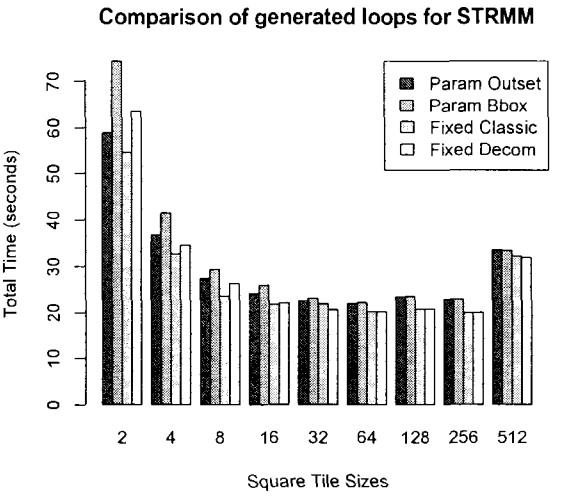


Figure 3.12.
Total execution time for STRMM for matrices of size 3000 × 3000.

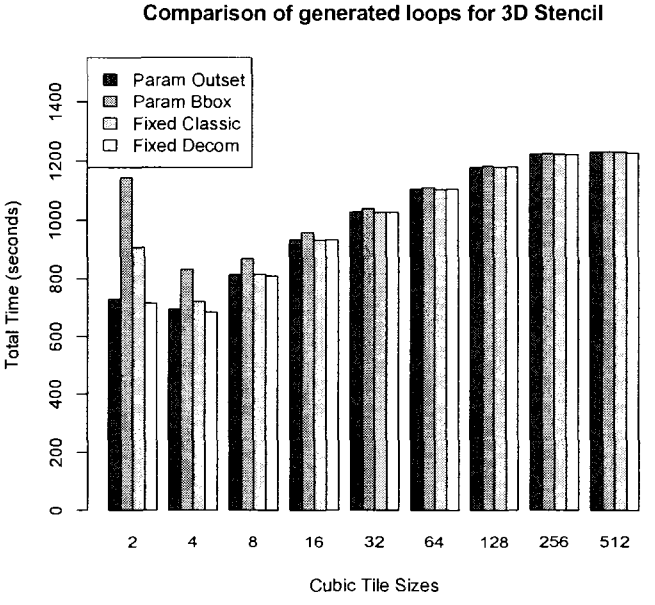


Figure 3.13.
Total execution time for 3D Stencil on a 2D data grid of size 3000 × 3000 over 3000 time steps.

	LUD	SSYRK	STRMM	3D Stencil
fClassic	32.4	28.6	29.0	26.0
fDecom	55.2	51.0	50.4	45.0
pBbox	53.5	53.2	51.2	54.0
pOutset	52.0	53.8	52.1	54.1

Table 3.2.
Tiled loop generation times (in milliseconds) of the four methods on the four benchmarks. The four methods fixed classic, fixed decomposed, parameterized bounding box, and parameterized outset are denoted by fClassic, fDecom, pBbox, and pOutset respectively.

box, and parameterized outset are shown in Table 3.2. The timings include file IO. Further, the timings for the parameterized bounding box method do not include the time to generate the bounding box from the iteration space polyhedron. For the experiments it was given as user input. In a fully automated scenario, this additional time for generating the bounding box will add to the generation time of the bounding box method.

Overall, the cost of code generation for the three methods, viz., fixed decomposed, bounding box, and parameterized outset, falls within the range of 45 to 55 milliseconds. Hence they have very comparable generation efficiency (even when the time to generate the bounding box is not included). Though the fixed classic method seems to be significantly more efficient than the fixed decomposed method, as observed by Goumas et al. [51] and us [71], it has scaling problems as the number of number of tiled loops increase.

In summary, the parameterized decomposed method generates code with performance comparable if not better than both fixed and parameterized tiled code generation methods. For parameterized tiled code generation, the parameterized decomposed method based on the outset is clearly better than the traditional bounding box method, especially for smaller tile sizes. The code generation time for all of the methods is comparable and quite small.

3.5 Finding Full Tiles Using the Inset

One possible reason for loop overhead is the presence, within the loop bounds for each tile, of the bounds for the original iteration space as well as the tile so that no iterations outside of the original iteration space are executed. Ancourt and Irigoin [9] suggest that tiled code may be

optimized by generating different code for full tiles versus partial tiles. Previous work [64] uses index set splitting to break the iteration space into full and partial tiles so that iteration bounds can be removed from the bounds for the full tiles. Other work [51] indicates that they differentiate between full and partial tiles, but details are not provided. Since distinguishing between full and partial tiles is also important for register tiling and possibly hierarchical tiling, we present two possible approaches for doing just that. Both approaches are based on constructing the *inset* polyhedron such that any tile origins within the inset polyhedron P_{in} are tile origins for full tiles. As before, our challenge comes from the fact that we seek to do this for parameterized tile sizes, and our solution again yields us a polyhedron with the tile sizes as additional parameters, thus enabling us to build on well developed theory and tools.

Distinguishing between full and partial tiles is applicable to all of the tiled code generation techniques discussed in Section 3.1. The inset can be computed as quickly as the outset, and it is possible to show that points are in the calculated inset if and only if they are possible tile origins for full tiles. Once the inset has been computed, it is possible to leverage existing code generators to generate the tile loops that traverse the inset executing only full tiles and the outset minus the inset executing partial tiles.

3.5.1 Algorithm for Computing Inset

As in Section 3.1, the original loop in question is represented as a set of inequalities

$$P_{iter} = \{\vec{z} \mid Q\vec{z} \geq (\vec{q} + B\vec{p})\},$$

where z is the iteration vector of size d, Q is a $m \times d$ matrix, $\vec{q}$ is a constant vector of size m, $\vec{p}$ is a vector of size n containing symbolic parameters for the iteration space, and B is a $m \times n$ matrix. The vector $\vec{s}$ specifies the (hyper) rectangle tiling, with s_i indicating the tile size for the ith dimension of the iteration space.

We define the inset polyhedron P_{in} such that any tile origins that lie within the inset polyhedron are tile origins for full tiles. All the points in a tile satisfy an inequality constraint if and only if the extreme points for the tile satisfy the constraint. The extreme points of a (hyper) rectangle tile can be calculated as follows. Let $\vec{s'} = \vec{s} - \vec{1}$ and let $S' = diag(\vec{s} - \vec{1})$. Then S' times any binary

vector of size d is an extreme point of the tile. Formally, the inset is

$$P_{in} = \{\vec{z} \mid \forall \vec{b} \in \{0,1\}^d, Q(\vec{z} + S'\vec{b}) \geq (\vec{q} + B\vec{p})\},$$

It is possible to compute the inset directly from the definition, but that would result in $m * 2^d$ constraints, with many of them being redundant. Instead, we calculate a matrix Q^- from the Q matrix in the constraints for the original iteration space, such that

$$Q^-_{ij} = \begin{cases} Q_{ij}, & \text{if } Q_{ij} < 0 \\ 0, & \text{if } Q_{ij} \geq 0 \end{cases}.$$

The algorithm for computing Q^- is $O(md)$ and results in m constraints for the inset,

$$\widehat{P_{in}} = \{\vec{z} \mid Q\vec{z} \leq (\vec{q} + B\vec{p}) - Q^-(\vec{s'} - \vec{1})\},$$

where $\vec{s}$ is the size d vector of tile sizes and $\vec{1}$ is a size d vector containing all ones.

Theorem 3.5.1. $\widehat{P_{in}} = P_{in}$.

Proof: The proof proceeds by construction. First, we write each bound for P_{in} on a separate line.

$$\begin{pmatrix} Q_{11}S'_{11}b_1 & \cdots & Q_{1d}S'_{dd}b_d \\ \cdots & \cdots & \cdots \\ Q_{m1}S'_{11}b_1 & \cdots & Q_{md}S'_{dd}b_d \end{pmatrix} \geq (\vec{q} + B\vec{p}) - Q\vec{z}$$

Note that the above inequality is true for all binary vectors $\vec{b}$. Each row represents 2^d constraints: one for each possible value of the binary vector $\vec{b}$. Since all of the entries in the S' matrix are non-negative, it is possible to select a particular binary vector for each row that results in the least possible value for each entry and therefore provides a tight bound for all the constraints represented by that row. Specifically that binary vector has entry b_j equal to one if and only if Q_{ij} is negative. Selecting the binary vector for each row, which results in the tightest bound is equivalent to calculating the matrix Q^-.

For all binary vectors $\vec{b}$, the following is true:

$$\begin{pmatrix} Q_{11}S'_{11}b_1 & \cdots & Q_{1d}S'_{dd}b_d \\ \cdots & \cdots & \cdots \\ Q_{m1}S'_{11}b_1 & \cdots & Q_{md}S'_{dd}b_d \end{pmatrix} \geq Q^{-}\vec{s'} \geq (\vec{q}+B\vec{p})-Q\vec{z},$$

where $\vec{s'} = \vec{s} - \vec{1}$. Therefore, $\widehat{P_{in}}$ is P_{in} with all redundant bounds removed. ∎

3.5.2 Code Generation Implementation

One property of an inset P_{in} is that $tile(z) \cap P_{iter} = tile(z)$ for all $z \in P_{in}$. In other words, constraints on the iteration space are redundant for any tile whose origin is in the inset. By removing these unnecessary loop bounds in the point loops, we can possibly reduce the loop overhead further. One may perform this optimization by checking whether a tile origin belongs to the inset before executing point loops or by splitting the inset from the outset.

To use the check approach, code must be generated that determines if a particular iteration lies within the inset. The other approach is to split the inset from the outset. Consider the fact that $P_{in} \subseteq P_{out}$. We associate a statement X_1 with P_{in} and a statement X_2 with P_{out} and feed both polyhedra to a code generator. Now, if a loop nest scans both P_{out} and P_{in} without guards, then loops that scan the inset must include both statements. Another advantage is easiness of incorporating. Consider the containment relation of P_{out} and P_{in}. Clearly, $P_{in} \subset P_{out}$. Now, if a loop nest scans both P_{out} and P_{in} without guards assuming that these are associated to two different statements, then loops that scan the inset must have two statements. Now, we know that iteration constraints are redundant whenever there are two statements in the loop since $P_{in} \subseteq P_{out}$. Therefore, we replace the loop bodies with statements X_1 and X_2 with the tile loops for full tiles, and we replace the loop bodies with statement X_2 only with tile loops for partial tiles.

This splitting scheme based on the union of inset and outset provides a way to enable a full versus partial tile optimization for parameterized tile code. Also, it is easy to incorporate this scheme using existing code generators. Note that many code generators have been designed and developed to remove guards by splitting the iteration space into disjoint regions associated to different sets of statements.

The tradeoff between splitting and inserting a check has not been fully explored. For register tiling, it would seem that checking each tile to determine if it is full clearly introduces too much overhead. However, splitting can result in significant blowup in code size, which can cause instruction cache problems. An advantage of splitting over checking is that it reduces loop overhead without introducing additional overhead although checking is preferable in terms of code size.

3.6 Related Work

Ancourt and Irigoin proposed a technique [9] for scanning a single polyhedron, based on Fourier-Motzkin elimination over inequality constraints. Le Verge et al. [79, 80] proposed an algorithm that exploits the dual representation of polyhedra with vertices and rays in addition to constraints. The general code generation problem for affine control loops requires scanning *unions* of polyhedra. Kelly et al. [70] solved this by extending the Ancourt-Irigoin technique, and together with a number of sophisticated optimizations, developed the widely distributed Omega library [93]. Quillere et al. proposed a dual representation algorithm [94] for scanning the union of polyhedra, and this algorithm is implemented in the CLooG code generator [14] and its derivative Wloog is used in the WRaP-IT project.

Techniques for generating loops that scan polyhedra can also be used to generate code for fixed tile sizes, thanks to Irigoin and Triolet's' proof that the tiled iteration space is a polyhedron if the tile sizes are constants [62]. Either of the above tools may be used (in fact, most of them can generate such tiled code). However, it is well known that since the worst case complexity of Fourier-Motzkin elimination is doubly exponential in the number of dimensions, this may be inefficient. Methods for generating code for non-unimodular transformations use techniques similar to ours, however they use fixed lattices and we use a parameterized lattice.

Our work is similar in scope to that of Goumas et al. [51], who decompose the generation into two subproblems, one to scan the tile origins, and the other to scan points within a tile, thus obtaining significant reduction of the worst case complexity. They proposed a technique to generate code for *fixed-sized, parallelogram* tiles. Their technique computes an approximation to the outset, similar to our $\widehat{P_{out}}$. Specifically, they compute the *image* of $\widehat{P_{out}}$ by the tiling

transformation, H, and generate code to scan this image. Because of this, their code has ceiling and floor operations, and the loop body must compute an affine function of the loop indices to determine the tile origins. Their method can handle arbitrary parallelogram shaped tiles, and they also use a technique similar to our inset to optimize the code. Note however, that all their techniques are applicable only to fixed tile sizes.

In contrast, our algorithm handles parameterized tile sizes. The key insight is that we view the outset as a polyhedron with, other than the program parameters, n additional parameters, namely the tile sizes. This allows us to efficiently leverage most of the well developed tools, and our technique performs as well as, if not better than, all others, at no additional cost.

There are also a number of additional differences. Our algorithm generates tile loops whose indices always remain in the coordinate space of the original loop. This avoids floor and ceiling functions, and enables us to generate tile loops through a very simple post-processing: adjust the lower bounds, and introduce a stride corresponding to the tile size. Our method is restricted to transformations that can be expressed as a composition of a unimodular transformation, followed by a rectangular tiling (blocking).

The work by Amarasinghe and Lam [7, 8] and Größlinger et al. [53] are related and were discussed in the previous chapter.

Jiménez et al. [64] develop code generation techniques for register tiling of non-rectangular iteration spaces. They generate code that traverses the bounding box of the tile iteration space to enable parameterized tile sizes. The focus of their paper is applying index-set splitting to tiled code to traverse parts of the tile space that include only full tiles. Their approach involves less overhead in the loop nest that visits the full tiles; however, they experience significant code expansion. We suggest two possible approaches for differentiating between full and partial tiles: either generate a check to determine if the tile being visited is a full tile, or associate two different loop bodies with the inset and outset and let any polyhedra scanning code generator generate the appropriate code. The trade-off between the overhead due to the check versus the cost due to code expansion that occurs using index-set splitting or loops that scan the union of polyhedra is unclear and an area for further study.

3.7 Discussion

The two polyhedral sets, viz., outset and inset, introduced in this chapter play a fundamental role in tiled code generation. First, as shown in this chapter, the outset is useful for efficient generation of high quality parameterized tiled loop nests. The inset is useful in characterization of a linear condition which are satisfied by the full tile origins. We show in the next chapter how these sets enable efficient tiled loop generation for multi-level tiling and also separation of full and partial tiles at any arbitrary level of tiling.

CHAPTER 4

Multi-level Tiled Loop Generation

In this Chapter we propose a technique for generating multi-level tiled loops where the tile sizes can be fixed (constants) or symbolic parameters or mixed. Our technique provides multiple-levels of tiling at the same cost of generating tiled loops for a single level of tiling. We propose a novel formalization extending the classic tiling transformation [62, 136] to multiple levels. We propose a method for separating partial and full tiles at any arbitrary level, without fixing the tile sizes. We have implemented all the proposed code generation techniques and the tool is available open source [55]. Our technique provides m levels of tiling at the price of one. This claim is justified via a theoretical complexity analysis of our technique and extensive evaluation of both the generation efficiency and quality of the generated code on benchmark routines form BLAS, LUD, and stencil computations.

The work presented in the chapter was done in collaboration with DaeGon Kim, Dave Rostron, and Michelle Mills Strout. It was presented in [71].

4.1 Multi-level Tiling

The input is a perfect loop nest, and it is appropriately transformed so that rectangular tiling is valid. In this section, we describe two multi-level tiling approaches. The first one is an extension

of the classic tiling transformation [136] to multiple levels and is restricted to the case of where the tile sizes are fixed. The second one is based on the concept of the outset (introduced in the previous Chapter) and can be used when the tile sizes are symbolic parameters or fixed constants or mixed.

Our input model is perfectly nested loops. Our techniques are applicable to cases where rectangular tiling is valid or can be made valid by an appropriate preprocessing transformation (e.g., skewing). We assume that this has already been done. The input loop of depth d is represented as a set of m inequalities

$$P_{iter} = \{\vec{z} \mid Q\vec{z} \geq (\vec{q} + B\vec{p})\},$$

where z is the iteration vector of size d, Q is a $m \times d$ matrix, $\vec{q}$ is a constant vector of size m, $\vec{p}$ is a vector of size n containing symbolic parameters for the iteration space, and B is a $m \times n$ matrix. The tile sizes are represented by the vector $\vec{s}$; we use $\vec{s'}$ to denote $\vec{s} - \vec{1}$.

4.1.1 Multi-level tiling for fixed tile sizes

We start with the classic definition of single-level rectangular tiling [136]. Given an iteration space P_{iter} and a vector $\vec{s}$ of fixed tile sizes , the tiled iteration space is given by

$$P^1_{tiled} = \{(\vec{t}, \vec{z}) \mid \vec{s} \circ \vec{t} \leq \vec{z} - \vec{o} \leq \vec{s} \circ \vec{t} + \vec{s} - \vec{1}, \vec{z} \in P_{iter}\}$$

where $\vec{o}$ is an offset and the operator $\circ$ denotes component wise multiplication of vectors. The tiles are enumerated by $\vec{t}$ and the points within a tile are represented by $\vec{z}$. The tiled iteration space denoted by P^1_{tiled} is a polyhedron (as the tile sizes are fixed). Generating the tiled loop nest is now reduced to generating loops that scan the polyhedron P^1_{tiled}. There are standard tools such as OMEGA [70] and CLOOG [14] which can be used for to generate such loops. Note that P^1_{tiled} is a polyhedron only when the tile sizes are fixed and hence the approach is not applicable when the tile sizes are symbolic parameters.

We can extend the definition to multiple levels of tiling as follows. Given an iteration space

P_{iter} and a list of tile size vectors $\vec{s_1},\ldots,\vec{s_m}$, a multi-level tiling can be described in a similar way.

$$\begin{aligned} P^m_{tiled} \;=\; & \{(\vec{t_1},\ldots,\vec{t_m},\vec{z}) \mid \forall i = 1,\ldots,m-1 : \\ & \vec{s_i} \circ \vec{t_i} \le \vec{t_{i+1}} - \vec{o_{i+1}} \le \vec{s_i} \circ \vec{t_i} + \vec{s_i} - \vec{1}, \\ & \vec{s_m} \circ \vec{t_m} \le \vec{z} - \vec{o_m} \le \vec{s_m} \circ \vec{t_m} + \vec{s_m} - \vec{1}, \vec{z} \in P_{iter}\} \end{aligned} \tag{4.1}$$

where o_i is an offset at the appropriate level. All tile sizes are integer constants. Also, note that actual tile sizes are a product of all inner tile sizes because tiling at level k is a tiling on the $(k+1)$ tiled space, not the original iteration space. Although this formulation is a direct extension of Xue's definition of single level tiling [136], to the best of our knowledge, this is first formalization and presentation of it—other formulations [65] of multi-level tiling are based on the strip-mine and interchange view of tiling. Now given the fact that this set P^m_{tiled} is a polyhedron, the scanning loops can be easily generated by existing tools, such as OMEGA test and CLOOG. Our generator for this method uses CLOOG.

4.1.2 Multi-level tiling using the outset

Another view of tiled loop generation is based on the outset method as described in the previous section, where the coordinates of the tile origins are obtained by intersecting the outset P_{out} with a parameterized lattice $Lattice(\vec{s})$. This method does not require the tile sizes to be fixed. Multi-level tiling in this method can be viewed geometrically as shown in Figure 4.1. We start with the first level of tiling of the iteration space and the first level tiles are further tiled to achieve the second level of tiling. In Figure 4.1, the first level of tiling uses 4×4 tiles and the second level uses 2×2 tiles. The geometric view not only aids visualization but also gives a mathematical view of the multi-level tiling: the tile origins at a given level k of tiling can be viewed as the intersection of the tiles at the previous $(k-1)$ level and the lattice parameterized by the tile sizes of level k.

To exploit the geometric view for tiled loop generation we need to handle one important issue. Consider the outer level of tiling shown in Figure 4.1. There are three partial outer-tiles and one full outer-tile. When we apply another inner-level of tiling the outer-tiles become the iteration space for them, and we need to able to handle the different shapes of the partial outer-tiles. We handle this by (over) approximating the partial outer-tiles by full tiles. Such an approximation

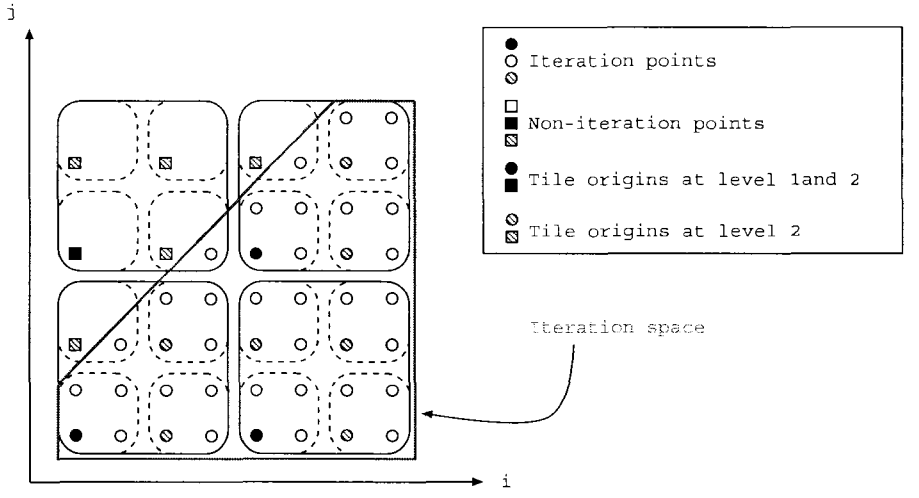


Figure 4.1.
Multi-level tiling as repeatedly tiling each tile on a triangular iteration space

```
// Outermost tile loops that scan the outset
for (t_i^1=1; t_i^1 <= 8 ; t_i^1 += 4)
  for (t_j^1=1; t_j^1 <= min(t_i^1+4,8) ; t_j^1 += 4)

    // Tile loops that scans 4×4 tiles by 2×2 tiles
    for (t_i^2=t_i^1; t_i^2 <= t_i^1 + 3 ; t_i^2 += 2 )
      for (t_j^2=t_j^1; t_j^2 <= t_j^1 + 3 ; t_j^2 += 2 )

        // Point loops that scans the intersection of
        // a 2×2 tile and iteration space
        for (i=max(1,t_i^2); i <= min(t_i^2+1,8); i++)
          for  (j=max(1, t_j^2); j <= min(t_j^2+1,i+1,8); j++)
              S(i,j);
```

Figure 4.2.
A loop nest corresponding to the multi-level tiling in Figure 4.1

allows a uniform treatment of the further levels of tiling. The 2-level tiled loop nest generated using this method for the example is shown in Figure 4.2. Note that the tile-loops at the second level treat partial tiles as full tiles. The general structure of the multi-level tiled loops generated using this method is shown in Figure 4.3. The outermost tile-loops are generated using the outset and all inner-level tile-loops are generated using the bounds of a full-tile, referred to as *Box-tile-loops*. The innermost loop nest consists of the point-loops which have the both the tile bounds and the iteration space bounds. We expect the execution time overhead due to the approximation of inner-level partial tiles by full tiles to be insignificant. Our expectation is confirmed by our experimental results as discussed in Section 4.4.2.

```
tile-loops scanning outset
   Box-tile loops
      ...
       Box-tile loops
         Point-loops
```

Figure 4.3.
Structure of multi-level tiled loops generated with the outset method when partial and full tiles are not separated.

Multi-level tiling based on outset can be formalized as follows. Given an iteration space P_{iter} and a list of tile size vectors $\vec{s_1},\ldots,\vec{s_m}$, the tiled iteration space can be expressed as follows:

$$\begin{aligned}\mathscr{P}^m_{tiled} = \{&(\vec{t_1},\ldots,\vec{t_m},\vec{z}) \mid \forall i=2,\ldots,m : \\ &\vec{t_1} \in P_{out} \cap Lattice(\vec{s_1},\vec{o}), \\ &\vec{z} \in P_{iter} \cap tile(\vec{t_1},\vec{s_1}) \cap \cdots \cap tile(\vec{t_m},\vec{s_m}), \\ &\vec{t_i} \in tile(\vec{t_1},\vec{s_1}) \cap \cdots \cap tile(\vec{t_{i-1}},\vec{s_{i-1}}) \cap \\ &\quad Lattice(\vec{s_i},\vec{t_{i-1}})\}\end{aligned} \tag{4.2}$$

where $Lattice(\vec{s_i},\vec{t_{i-1}})$ is the set of points generated by $\vec{s_i} \circ \vec{x} + \vec{t_{i-1}}$ for any integer vector $\vec{x}$, and $\vec{s_i}$ can be a vector of either symbolic tile size parameters, constants, or a mixture of both. Note that the offset of the lattice depends on the origin of each tile at the previous level. Given a tile, $tile(t_i,s_i)$, the first tile at level $(i+1)$ that is contained in $tile(t_i,s_i)$ must be $tile(t_i,s_{i+1})$ because t_i is still the lexicographical minimum of (t_i,s_i). Otherwise, some points in the iteration space will not be scanned. Correctness of this formulation follows directly from the fact that P_{out} contains origins of the tiles whose union is a super-set of P_{iter}. Further, by including in the formulation, the constraints that define P_{iter} we guarantee that only valid iteration points in the tiles are enumerated. Also note that the formulation does not impose the restriction that outer tile sizes are multiples of inner tile sizes.

In most practical cases, tile sizes $\vec{s_i}$ are component-wise multiples of $\vec{s_{i+1}}$ for all $i=1,\ldots m-1$.

```
tile-loops scanning outset
   Box tile-loops-L1
    ...
     Box tile-loops-Lk
     if ( FULL(Lk-tile) ) {
        ...
        Box-tile loops-Lm
         point-loops with tile bounds only
     } else { // partial-tile-Lk
        Point-loops
     }
```

Figure 4.4.
Structure of multi-level tiled loops generated with the outset method when the partial and full tiles are separated at some tiling level k.

The constraints of the tiled iteration space in (4.2) for this case can be simplified to:

$$\begin{aligned} \mathscr{P}^{m}_{tiled} \;=\; & \{(\vec{t_1},\ldots,\vec{t_m},\vec{z}) \mid \forall i = 2,\ldots,m : \\ & \vec{t_1} \in P_{out} \cap Lattice(\vec{s_1},\vec{o}), \vec{z} \in P_{iter} \cap tile(\vec{t_m},\vec{s_m}), \\ & \vec{t_i} \in tile(\vec{t_{i-1}},\vec{s_{i-1}}) \cap Lattice(\vec{s_i},\vec{t_{i-1}})\} \end{aligned} \tag{4.3}$$

Note that the constraints from all the previous level tilings become redundant with this assumption on tile sizes. From now on for ease of description, we assume that the tile sizes at an outer level are component-wise multiples of all the inner level tile sizes. However, our method does not impose this restriction.

4.2 Separating partial & full tiles

As discussed earlier, separation of partial and full tiles has several applications. In this section, we discuss how the inset (introduced in Chapter 3.5) is used for separation. Separation at any level k implies that the further tilings (for levels $k+1 \ldots m$) are performed only on full tiles of level k. The partial tiles of level k are not further tiled. Consider the number of full and partial outer-tiles in Figure 4.1. There is one full outer-tile and three partial outer-tiles. If we separate full tiles from partial tiles at the outer level of tiling, then there are only four full inner-tiles, since only the full outer-tiles are tiled further. However, we can see that there are 10 full inner-level

tiles in the iteration space. By separating the partial and full tiles at the inner-level (and not at the outer-level) we can actually recognize all the 10 inner-level tiles as full. However, separation at the inner-level leads to more inner-level full tiles but also results in enumeration of more empty inner-tiles. Hence, there is a trade-off between more inner-level tiles versus enumeration of empty tile origins. Further, we can also apply splitting multiple times if needed.

The general structure of such a multi-level tiled loop nest with separation of partial and full tiles at an arbitrary level k is shown on Figure 4.4. Note that the partial tiles at level k are not further tiled and they execute the standard point-loops. On the other hand, the full tiles of level k are further tiled and their body contain a special form of point-loops called *box-point-loops*. These box-point-loops are the loops in which the iteration space bounds are omitted.

To recall, the inset P_{in} represents the set which contains all the full-tile origins. Let us denote by $P_{in}(\vec{s_k})$ the inset computed using the tile sizes of level k and the iteration space P_{iter}. Now we can check *at any level* l whether a tile origin represents a full tile or not by checking whether it belongs to $P_{in}(\vec{s_l})$ or not. This is the key idea underlying our separation algorithm. For any user specified level k of separation we generate the outset $P_{in}(\vec{s_k})$ and use it to test whether a tile is full or partial. This test corresponds to the `FULL(Lk-tile)` test in Figure 4.4.

When the separation happens at level k, the set of points in the full tiles at level k can be described as follows:

$$\begin{aligned}\mathscr{P}^k_{full} \;=\; & \{(\vec{t_1},\ldots,\vec{t_m},\vec{t_{m+1}}) \mid \forall i=2,\ldots,m+1 : \\ & \vec{t_1} \in P_{out} \cap Lattice(\vec{s_1},\vec{o}), \vec{t_k} \in P^k_{in}, \\ & \vec{t_i} \in tile(\vec{t_{i-1}},\vec{s_{i-1}}) \cap Lattice(\vec{s_i},\vec{t_{i-1}})\}\end{aligned}$$

where $\vec{s_{m+1}}$ is $\vec{1}$. The set of points in the partial tiles can be described as follows:

$$\begin{aligned}\mathscr{P}^k_{partial} \;=\; & \{(\vec{t_1},\ldots,\vec{t_k},\vec{z}) \mid \forall i=2,\ldots,k : \\ & \vec{t_1} \in P_{out} \cap Lattice(\vec{s_1},\vec{o}), \vec{t_k} \notin P^k_{in}, \\ & \vec{z} \in P_{iter} \cap tile(\vec{t_k},\vec{s_k}), \\ & \vec{t_i} \in tile(\vec{t_{i-1}},\vec{s_{i-1}}) \cap Lattice(\vec{s_i},\vec{t_{i-1}})\}.\end{aligned}$$

Different levels of separation may be preferred, based on the context in which separation is used. For example, for a 2-level tiling in the context of caches and registers an inner-level of tiling might be preferred. An example of this is shown in our experiments on cache and register tiling.

4.3 The loop generation algorithm

Now we present our algorithm for generating multi-level tiled loop nests with parameterized, fixed, or mixed tile sizes. It is given in Algorithm 2 and its input is the original iteration space, number of levels of tiling, whether the loops are to be split for partial vs. full tile separation, and if so, what is the level at which this split needs to be performed. The output of the algorithm is the multi-level tiled loop nest.

We illustrate the steps of the algorithm on the 2D Stencil example. We seek to generate a 2-level tiled loop nest where full and partial tiles are split at the first level. We first compute an outset of the iteration space with the outer-tile sizes. Then, we generate the point loops whose bounds consists of iteration space bounds and the surrounding tile bounds. The split level determines the tile bounds used in the point-loops generation as shown in lines 2-5 of the algorithm. These loops are generated by a call to CLOOG. Next, we compute the inset of iteration space with respect to the split level (here, first) tile sizes and indices as shown in lines 6-7. The bounds of the inset are shown below.

$$\begin{aligned} P_{in} \;=\; & \{(t_k, t_i) \mid 1 \leq t_k;\, t_k + s_k - 1 \leq N_k; \\ & t_k + s_k \leq t_i; t_i + s_i - 1 \leq t_k + N_i\} \end{aligned} \tag{4.4}$$

where s_k and s_i are symbolic tile size parameters along k and i dimensions, respectively. The guard for splitting partial and full tiles is obtained directly from the inset. The complete multi-level tiled loop nest for the 2D Stencil example with separation at the first level is shown in Figure 4.5. At line 9 we see that the guard is a direct translation from the inset in (4.4).

Once the point-loops and inset based on a split level are generated we can generate all the loops. The construction of the inner-level tile-loops, the guards and the box-tile-loops can be done through a simple pretty printing using the appropriate bounds. Combing these with the

Algorithm 2 An algorithm for generating multi-level tiled loops based on outset approach

```
INPUT : P_iter : Iteration space matrix,
        tileSizes[1...m] : tile size (integer or symbolic parameter) vector,
        tileIndexes[1...m] : tile index name vector,
        split : a boolean value whether full and partial tiles are split
        splitLevel : level at which full and partial tiles are split

BEGIN
    Matrix outset, inset;
    VectorOfString pLoops, comLoops;

    // Compute P_out
1:  outset = computeOutset(P_iter, tileSizes[1],tileIndexes[1]);

    // Scan P_iter, add tile bounds with appropriate level
2:  If (split == true)
3:      pLoops = generatePointLoops(P_iter, tileSizes[m],tileIndexes[m]);
4:  else
5:      pLoops = generatePointLoops(P_iter,tileSizes[splitLevel], tileIndexes[splitLevel]);

    // Compute P_in when split is greater than 0
6:  If (split == true)
7:      inset = computeInset(P_iter, tileSizes[splitLevel],tileIndexes[splitLevel]);

    // Combine point-loop, box-loop and guard for split
8:  comLoops = combine(pLoops, tileSizes[1...m],tileIndexes[1...m], splitLevel, inset );

    // Generate loops that scans outset while printing
    // comLoops instead of point-loop
9:  printScanningLoops(outset, comLoops);
END
```

previously generated point-loops (as shown in line 8) we get all the loops except the outer-most tile-loops. This is generated by a call to CLOOG to generate loops that scan the outset and post-processing it to add lower bound shifts and strides. The resulting tile-loops are shown in lines 2-5 of Figure 4.5. Finally we compose these outermost tile-loops to obtain the complete tiled loop nest with separation of partial and full tiles.

4.3.1 Complexity & scalability of the algorithm

Let us first consider the case where no full vs. partial tile separation is performed. Intuitively, the key steps are computing the outset to generate the outermost tile-loops and constructing all

```
// Outermost tile loops that scan the outset
T1kLB = -S1k+2; T1kLB = LB_SHIFT(T1kLB,S1k);
for (T1k = T1kLB; T1k <= Nk; T1k += S1k) {
  T1iLB = T1k-S1i+2; T1iLB = LB_SHIFT(T1iLB,S1i);
  for (T1i = T1iLB; T1i <= T1k+Ni+S1k-1; T1i += S1i) {
    // Is (T1k,T1i) a full tile at level 1?
    if ( T1k-1 >= 0 \&\& -T1k+Nk-S1k+1 >= 0 \&\&
      -T1k+T1i-S1k >= 0 \&\& T1k-T1i+Ni-S1i+1 >= 0 ){
     // Box-loops scanning origins of level 2 tiles.
     for (T2k = T1k ; T2k<=T1k+S1k-1 ; T2k += S2k )
      for (T2i = T1i ; T2i<=T1i+S1i-1 ; T2i += S2i )
       // Box-loops scanning points in level 2 tiles.
       for (k = T2k ; k<=T2k+S2k-1 ; k++ )
        for (i = T2i ; i<=T2i+S2i-1 ; i++ )
              S1 ;
    } else { // (T1k,T1i) is a partial tile at level 1
    // Point loops scanning partial tiles at 1st level.
    for (k= max(T1k,1);k<=min(T1k+S1k-1,Nk);k++)
     for (i= max(T1i,k+1);i<=min(T1i+S1i-1,k+Ni);i++)
          S1 ;
    }
} }
```

Figure 4.5.
A multi-level tiled loop for the 2D Stencil. The body of the loop is by S1.

the box-tile-loops and constructing the point-loops. The construction of the outset can be done in time linear on the number of bounds on the original loop nest. Further, the construction of the box-tile loops is a simple pretty-printing using the tile indices and sizes. The construction of the point-loops and the tile-loops using the outset are done via CLOOG. The complexity of each of these calls to CLOOG is exponential in the number of bounds of the *original loop nest,* not the number of bounds in the tiled loop nest. Hence, the entire multi-level tiled loop nest construction involves two calls to an exponential function and a couple of functions that are linear on the number of bounds on the original loop nest and the number levels of tiling. The key point to note is that the number of calls to the exponential function do not depend on the number of levels. In fact, for any arbitrary number of levels of tiling, exactly two calls are made to the exponential-time function. Now, if we consider separation of partial and full tiles, all that is required is the computation of the inset (which can be done in linear time) and the pretty printing of it as a guard. On the whole, the time complexity of our algorithm is determined by the time taken by the two calls to CLOOG, and is constant with respect to the number of levels of tiling. The experimental results in Section 4.4.1 confirm this, and also further validate our claim that we can generate multi-level tiled loops at the cost of a single-level tiled loops.

In contrast the time for the classic method depends on the number m of multi-level tiling.

For an original loop nest of depth d, the number of dimensions and constraints increase by md and $2md$, respectively, as the level of tiling increase to m (assuming all the dimensions are tiled). This results in an exponential space/time complexity which grows with the number of levels of tiling. The experimental results in Section 4.4.1 show how this exponential growth with respect to number of levels renders the technique inapplicable beyond two levels of tiling. The multi-level tiled loop generation method proposed by Jiminéz et al. [65] has an exponential time complexity at each level of tiling, and this grows linearly with the number of levels of tiling.

4.4 Experimental Validation

We implement three different multi-level tiled loop generators. The first generator is for the case when the tile sizes are fixed, and uses the classic tiling method discussed in Section 4.1.1. The second generator is capable of generating tiled code with the tile sizes that are fixed or parameterized or mixed and is based on the method discussed in Section 4.1.2. The third generator implements the additional feature of splitting (or separating) partial and full tiles at some user specified level. The generators are implemented in C++. The CLOOG [14] loop generator is used internally to generate the point-loops and the loops that scan the outset. Our technique is independent of the internal code generator and for example, we could use OMEGA [70] instead of CLOOG. We chose CLOOG for its robustness across several benchmarks and its code generation speed (up to 4×faster than OMEGA [14]).

To evaluate the generation efficiency and the quality of the generated code we conduct three sets of experiments. The benchmarks used for the experiments are given in Table 4.1. The benchmarks 2D Stencil and 3D Stencil correspond to a Gauss-Siedel style stencil where a 1D array (or 2D array resp.) is updated over a time step loop. For these two benchmarks, we first applied skewing to make rectangular tiling valid and then used the skewed iteration space as input to our generator. The skewing makes the iteration space non-rectangular. The benchmark LUD is LU decomposition computation without pivoting. The benchmarks SSYRK and STRMM are routines from BLAS3 and correspond to symmetric rank k update and the triangular matrix product computations, respectively. The loop nest depth of the benchmarks is shown in the third column of Table 4.1 and for the experiments, all the loops are tiled at all the levels for all the benchmarks.

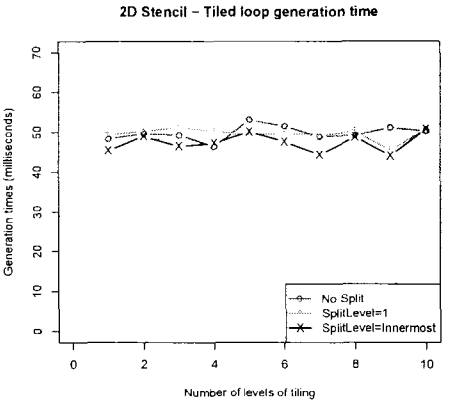


Figure 4.6.
Generation time for multi-level tiling of 2D Stencil.

The three sets of experiments we conduct are aimed at evaluating the (i) the generation efficiency of loop generators, (ii) the cost of parameterization, i.e., what is the execution time cost for not fixing the tile sizes and leaving them as parameters, and (iii) the effect of the level at which partial and full tiles are separated. The following sections discuss each of these experiments.

4.4.1 Generation efficiency

We evaluate two aspects of the generation efficiency. First, we evaluate how our method scales with respect to the number of levels of tiling. Second, we compare the generation times for the parameterized and the fixed method. The second comparison also evaluates the overhead due to the over-approximation of the inner-level partial tiles by full tiles (cf. Section 4.1.2). All the generation efficiency experiments were run on an Intel Core2 Duo processor running at 1.86 GHz with an L2 cache of size 2MB. We used `g++` 4.1.1. with `-O3` optimization level to compile our loop generators. The timings use `gettimeofday()`. Our code generator supports arbitrary (hyper-)rectangular tiles. For ease of experimentation we have used square tile sizes.

The generation times for the five benchmarks, 2D Stencil, LUD, SSYRK, 3D Stencil, and STRMM are shown in Figures 4.6,4.7, 4.8,4.9, and 4.10. The x-axis represents the number of levels of tiling and the y-axis represents the generation time (including file IO) in milliseconds. The generation time labeled No Split refers to the case where there is no-splitting of partial and full tiles and the other two - SplitLevel=1 and SplitLevel=Innermost - represent the generation

	Description	Loop depth
2D Stencil	Gauss-Siedel Style 2D stencil computation	2
LUD	LU decomposition of a matrix without pivoting	3
SSYRK	Triangular matrix multiplication	3
STRMM	Symmetric Rank k Update	3
3D Stencil	Gauss-Siedel Style 3D stencil computation	3

Table 4.1. Benchmarks used for evaluating generation efficiency and code quality.

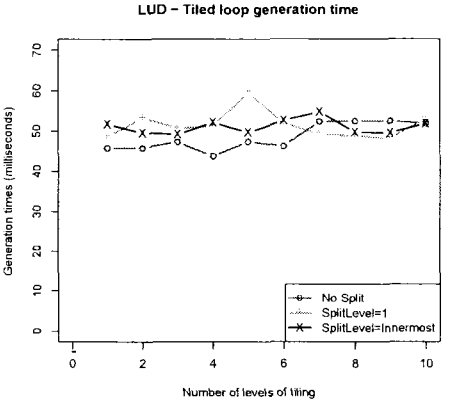


Figure 4.7. Generation time for multi-level tiling of LU decomposition.

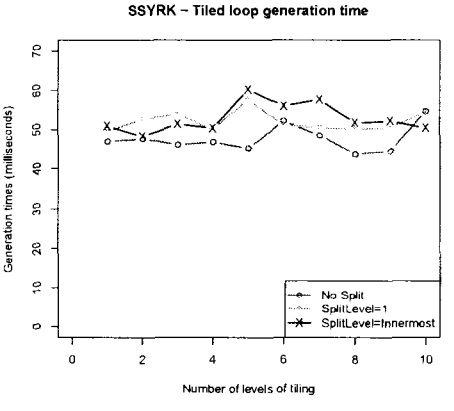


Figure 4.8. Generation time for multi-level tiling of symmetric rank k update (SSYRK).

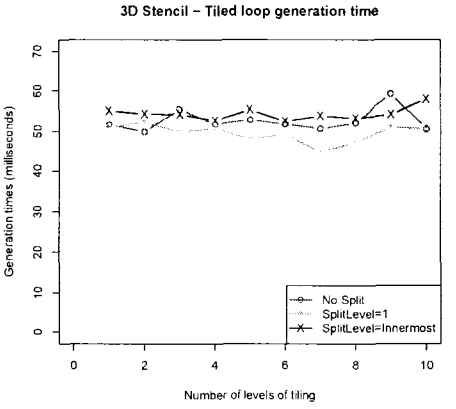


Figure 4.9.
Generation time for multi-level tiling of 3D Stencil.

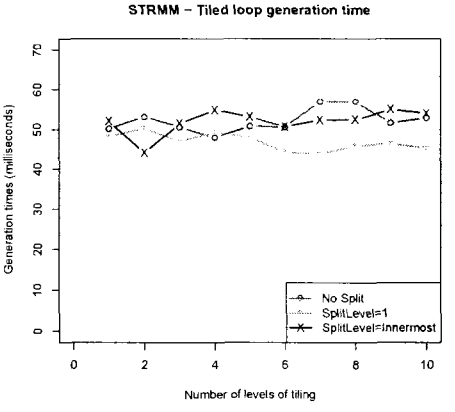


Figure 4.10.
Generation time for multi-level tiling of triangular matrix multiplication (STRMM).

where the splitting is done at level 1 (outermost) and at the innermost level, respectively. Note that the case of a single level of tiling with no splitting corresponds to the experiments from the previous Chapter on parameterized single level tiled loop generation. The main observation from the graphs is that the generation time is fairly flat as the number of tiling levels increase. Almost all the generation times are within the range of 40 to 60 milliseconds. This experimentally confirms our claim that our technique provides a method that can generate multi-level tiled loops at the price of a single-level tiled loop nest. Further, the graphs also show that splitting does not introduce any additional generation cost.

The generation times for the classic method for fixed tile sizes is shown in Figure 4.11 (the

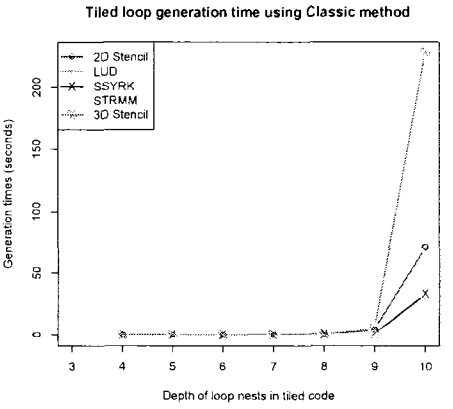


Figure 4.11.
Generation time for multi-level tiling of classic method. The x-axis of the graph is the number of loops in the tiled loop nest. The y-axis is the code generation time in seconds.

scale of x-axis is now in seconds). Note that the x-axis shows the number of loops in the tiled loop nest and not the number of levels tiled. For example, when a 3D loop nest is tiled two levels we will have 9 loops on the tiled loop nest. We show the number of loops in the tiled loop nest, because it is a finer granularity than the number of levels of tiling and shows clearly the exponential (w.r.t. the number of loops) nature of the method. The graph clearly shows that the generation time grows exponentially when the number of loops is 9 or higher. Hence, we could not obtain the generation times beyond two levels of tiling for this method. Although, it is not clear in the graph, the generation time grows exponentially even with smaller number of loops, but the difference of generation time among them is negligible.

4.4.2 Cost of parameterization

We evaluate the cost of parameterization by comparing the execution time of tiled code with fixed tile sizes and parameterized tile sizes. We use two levels of tiling one for the TLB and another for cache. This choice is motivated by our goal to compare two-level fixed and parameterized tiled codes where the differences due to the loop bounds computation can be easily quantified. Other choices for two level tiling such as tiling for parallelism and caches or tiling for caches and registers introduce many factors that influence the execution time and hence measuring the execution time difference due to the loop bounds computation becomes hard. The experiments are done on an Intel Pentium 4 at 3.2 GHz a 512 K L2 Cache and a TLB with 64 entries and pages of size 4K. We

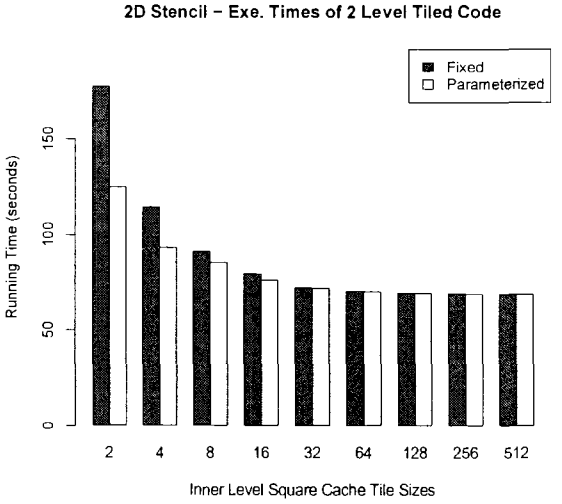


Figure 4.12.
Total execution time for 2D Stencil on a data array of size 65536. The x-axis shows the inner (cache) cubic tile sizes. The outer (TLB) tile size is fixed at 512.

used g++ 4.1.1. compiler with -O3 optimization.

Figures 4.12, 4.13, 4.14 and 4.15 show the execution times of the two-level tiled loops for the 2D Stencil, LU decomposition, SSYRK and 3D Stencil benchmarks, respectively. For the results the shown in the graphs the inner (cache) tile sizes were varied from 2 to 512 and the outer (TLB) tile size is fixed at 512. We also experimented with other outer (TLB) tile sizes and the results are similar to the ones presented here. We can observe that for small tile sizes the parameterized tiled loops are better and the for larger tile sizes they are comparable to the fixed tiled loops. At smaller tile sizes the ceil() and floor() functions used in the classic method induce higher overhead and hence result in slower execution time. Overall, the cost of parameterization seems to be negligible and hence we conclude that parameterized tiled codes should be the preferred choice.

4.4.3 Effect of separation level

We evaluate the effect of separating partial and full tiles at different levels tiling. We use the STRMM benchmark, and we tiled it two levels: one for cache and another for registers. The register tiles were (manually) fully unrolled and the array references were replaced by scalars to facilitate register promotion. The running times for two different cubic register tile sizes ($2 \times 2 \times 2$ and $3 \times 3 \times 3$) are shown in Figure 4.16. Also shown is the running time for one level of tiling

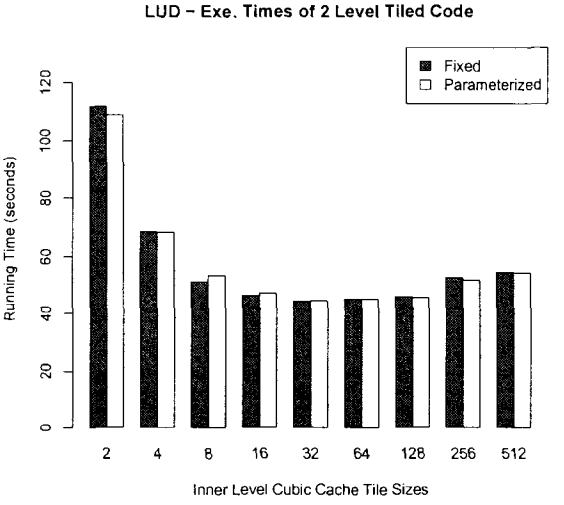


Figure 4.13.
Total execution time for LU decomposition on a matrix of size 2048 x 2048. The x-axis shows the inner (cache) cubic tile sizes. The outer (TLB) tile size is fixed at 512.

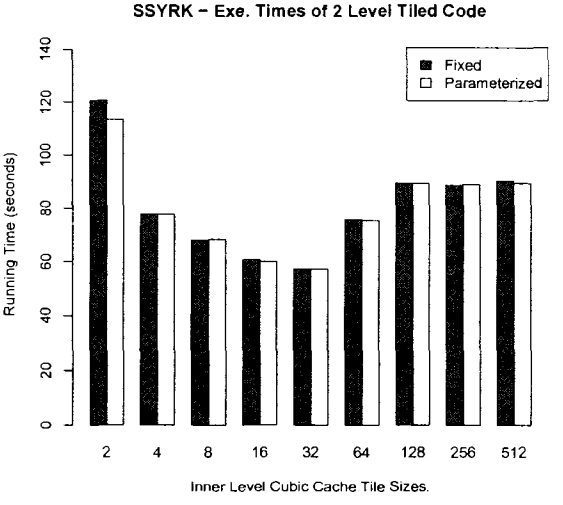


Figure 4.14.
Total execution time for symmetric rank k update (SSYRK) for matrix of size 2048 x 2048. The x-axis shows the inner (cache) cubic tile sizes. The outer (TLB) tile size is fixed at 512.

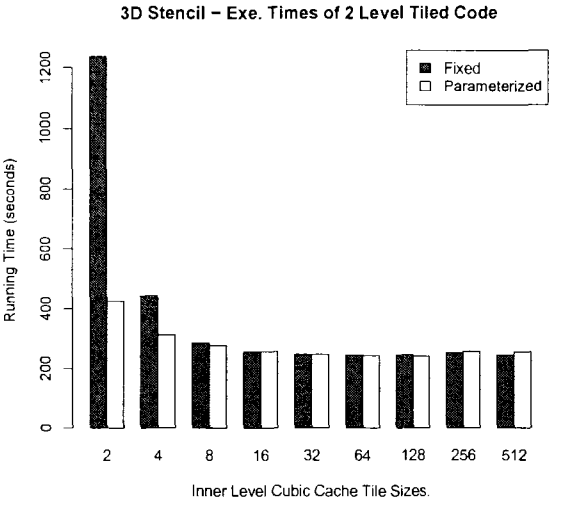


Figure 4.15.
Total execution time for 3D Stencil for a data array of size 2048 x 2048 over 2048 time steps. The x-axis shows the inner (cache) cubic tile sizes. The outer (TLB) tile size is fixed at 512.

for caches. First, the results clearly show (although this is orthogonal to our study) that tiling for both cache and registers gives better performance. Second, they also show how splitting at the second level achieves the best performance (around 13 seconds) when compared to others.

4.5 Related Work

Techniques related to parameterized tiled loop generation, particularly for a single level of tiling, were discussed in the previous chapter. Here we discuss the ones related to multi-level tiling. Rivera and Tseng [108] studied the effect of multiple levels of tiling for improving locality on multi-level caches. Multi-level tiled loop generation was not their focus. For simple rectangular iteration spaces, multi-level tiled loop generation is straightforward and has been used by several tools. However, for arbitrary polyhedral iteration spaces, there has not been much work. Jiminéz et al. [65] propose a technique for arbitrary polyhedral iteration spaces but for the fixed tile sizes case. Their technique is based on the strip-mine and interchange view of tiling. Their technique has a exponential complexity that grows with the number of levels of tiling. First, our technique can handle both fixed as well as parameterized tile sizes. Second, the exponential time complexity of our algorithm is fixed and does not grow with the number of levels. Third, we also propose a

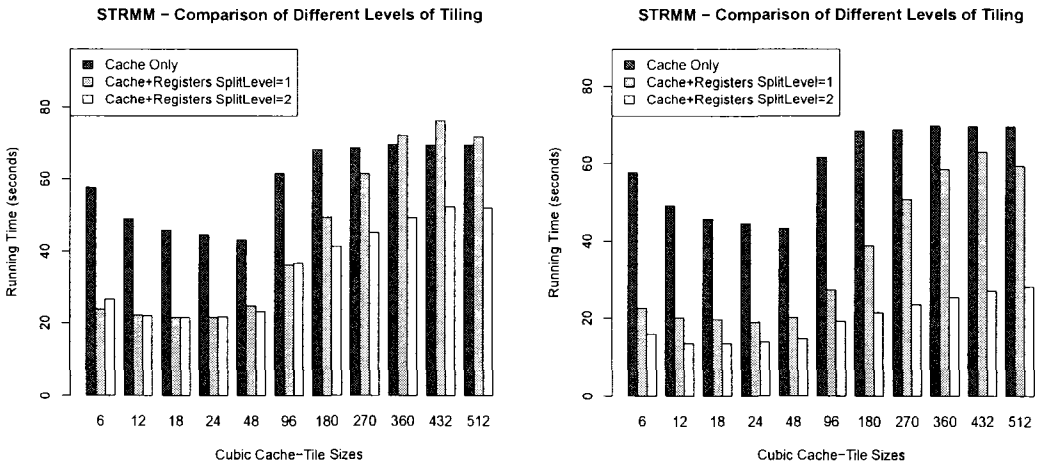


Figure 4.16.
Total execution time for triangular matrix multiplication for matrices of size 2048 × 2048. Two levels of tiling for cache and registers is used. The x-axis shows the cubic cache-tile sizes. The graph on the left is for a register-tile size of $2 \times 2 \times 2$ and the one on the right is for $3 \times 3 \times 3$.

method to separate full and partial tiles at any arbitrary level.

4.6 Discussion

Multi-level tiling is an important technique for mapping iterative computations to computer architectures with many levels of parallelism and memory hierarchy. We have described a method for automatically generating multilevel tiled code for any polyhedral iteration space where the tile sizes can be fixed or parameterized at each level. We have shown that parameterized multi-level tiled code can be generated at the same cost as a single-level tiled code. The code generation scheme can be easily incorporated into existing general compilers and domain-specific code generators. To the best of our knowledge, ours is the first technique proposed for multi-level tiled loop generation with parameterized tile sizes and also the first method to separate partial and full tiles when the tiles sizes are not fixed.

Two important extensions are possible. First, our techniques can be extended to case of imperfect loop nests—possibly, first for a single level and then to multiple levels. Second, the techniques can be extended to the generation of complete multi-level tiled code with both the tiled loops and the appropriate transformed loop body.

Part II

Tile Size Selection

CHAPTER 5

A Unified Framework for Optimal Tile Size Selection

A moment's insight is sometimes worth a life's experience.

— Oliver Wendell Holmes

As discussed in Chapter 1, the cost-model specificity of the tile size selection (TSS) solution methods lead to their non-extensibility and non-scalability. In this chapter we present a TSS framework that overcomes these limitations by providing a cost-model independent solution method. Our framework relieves the user from the tedious process of reasoning about the functions used in the cost model and exploiting their properties to derive a closed form or heuristic search algorithm for finding the best tile sizes.

First we describe the positivity property shared by the terms widely used optimal TSS models. We then introduce the class of functions called *posynomials* and the related class of optimization problems called *Geometric Programs*. To demonstrate the suitability of posynomials for optimal TSS, we present the reduction, to our framework, of five optimal TSS models proposed in the literature by a different authors in the context of using tiling for register reuse to cache locality to coarse-grained parallelism.

Machine and system parameters used in models	Functions modeling quantities of interest
Cache/TLB miss penalty Cache/TLB sizes Number of registers Number of functional units Latency of functional units Network latency Network bandwidth MPI Call start-up cost	Tile volume Number of tiles Number of cache misses Cache/register foot print Idle time in parallel execution Communication volume Loop overhead Temporary storage size Array pad size

Table 5.1.
These parameters and functions are widely used in TSS models. What is the mathematical property common to all these?

5.1 A Fundamental Property

Several authors have exploited particular properties such as linear, quadratic, hyperbolic, etc., of cost functions to derive optimal TSS solutions. Instead of exploiting the specific properties of a cost model to derive a solution, we ask a fundamentally different question. Is there a mathematical property that is inherent to the TSS models? Surprisingly yes! There is a simple property that is shared by almost all the TSS models proposed in the literature. Table 5.1 lists several functions and parameters that are used in TSS models. There is a fundamental mathematical property shared by all them. The property is *positivity*. All the machine and system parameters are positive quantities and the functions model positive quantities. The tile sizes which appear as variables in these functions are also positive. Essentially, the functions used in TSS models estimate positive quantities using positive parameters and positive variables. This positivity property might seem to be a simple one, but it has deep implications. This property distinguishes the class of optimization problems that are solvable in polynomial time and those that are not[1] [22]. As we show in the coming sections we can use this property as a basis to identify a class of polynomials called *posynomials* which can be used to formulate optimal TSS problems that can be solved efficiently.

[1] Use of polynomial functions with this property leads to convex optimization problems which can be solved for real solutions in polynomial time. On the other hand, optimization problems formulated with arbirtrary polynomials are not solvable in polynomial time.

5.2 Posynomials and Geometric Programs

We first introduce the basic building blocks of our formalism—monomials and posynomials—and present their closure properties. After that we introduce a particular class of convex optimization problems called *Geometric Programs* [42]. Monomials and posynomials are used in building the cost models and geometric programs are used in formulating the optimal TSS problem as a constrained optimization problem.

5.2.1 Posynomials

Let x denote the vector $(x_1, x_2, \ldots, x_n)$ of n real, positive variables. A function f is called a *posynomial* function of x if it has the form

$$f(x_1, x_2, \ldots, x_n) = \sum_{k=1}^{t} c_k x_1^{\alpha_{1k}} x_2^{\alpha_{2k}} \cdots x_n^{\alpha_{nk}}$$

where $c_j \geq 0$ and $\alpha_{ij} \in \mathbb{R}$. Note that the coefficients c_k must be non negative, but the exponents α_{ij} can be any real numbers, including negative or fractional. When there is exactly one nonzero term in the sum, i.e., $t = 1$ and $c_1 > 0$, we call f a *monomial* function.[2] For example, $0.7 + 2x_1/x_3^2 + x_2^{0.3}$ is a posynomial (but not a monomial); $2.3(x_1/x_2)^{1.5}$ is a monomial (and, hence a posynomial); while $2x_1/x_3^2 - x_2^{0.3}$ is neither.

Monomials and posynomials enjoy a rich set of closure properties, which are very useful in composition of smaller (say single level) optimal TSS models to build larger (multi-level) ones. Monomials are closed under product, division, non-negative scaling, power and inverse. Posynomials are closed under sum, product, non-negative scaling, division by monomials, and positive integer powers.

[2] Note that this definition of monomial is different from the standard one used in algebra.

5.2.2 Geometric Programs

A *geometric program* (GP) is an optimization problem of the form

$$\begin{aligned} \text{mimimize} \quad & f_0(x) \\ \text{subject to} \quad & f_i(x) \leq 1, \quad i = 1, \ldots, m \\ & g_i(x) = 1, \quad i = 1, \ldots, p \\ & x_i > 0, \quad i = 1, \ldots, n \end{aligned} \tag{5.1}$$

where $f_0, \ldots, f_m$ are posynomial functions and $g_1, \ldots, g_p$ are monomial functions. If $\forall i = 1 \ldots n$: $x_i \in \mathbb{Z}$, we call the GP an Integer Geometric Program (IGP). As presented by Boyd et al. [21] several extensions (*e.g.*, max()of posynomials) of GPs can be easily handled.

5.2.3 Efficient solutions via Convex Optimization

Recent advances [22] in convex optimization provide efficient polynomial time solution methods. GPs can be transformed into convex optimization problems using a variable substitution and solved efficiently using polynomial time interior point methods [74, 22]. The positivity property of the posynomials is extensively exploited in this transformation of GPs to convex optimization problems. The computational complexity of solving GPs are similar to that of solving linear programs [74]. Continuous real solutions can be found in polynomial time. Integer solutions need a branch and bound style algorithm, which in the worst-cast can take exponential time. However, we have found (*cf.* Sec 5.5) that for optimal TSS problems the IGPs are very small (few tile size variables and constraints) and solutions can be found quickly. Further, in the context of optimal TSS, it is very common to solve for real solutions and round them to obtain integer solutions. In such an approach we can obtain the solution in polynomial time irrespective of the complexity of the model.

5.3 Posynomials and TSS models

Posynomials are well suited for describing TSS models. The suitability is evident from the fact that almost all optimal TSS cost functions considered in the literature turn out to be posynomials.

A few of them are discussed in this section.

- **Models for data locality:** In general, as observed by Hsu and Kremer [59], the objective functions used in the context of tile size selection are all functions of the tile variables, cache capacity and cache line size. Due to the positivity of both the tile size variables and the cache parameters, these functions turn out to be posynomials. For example, as shown in Table 5.2 the cost functions used in several widely used optimal TSS models [77, 36, 45, 87, 33, 128, 107, 85] turn out to be posynomials. In addition to this, the TSS models used in the IBM XL compiler as described in [114] and the multi-level data locality tiling model proposed in [101] use posynomials and can be reduced to an IGP.

- **Models for parallelism.** Similar to data locality models, several important and popular models used in TSS for parallelism can be reduced to IGPs. Here, the TSS models are formulated with quantities such as tile volume, number of tiles, idle time in parallel execution, etc. and parameters such as network bandwidth/latency, MPI communication call cost, etc. Due to the positivity of the parameters and quantities they use and the tile size variables, these functions turn out to be posynomials. In particular, the commonly used communication minimal tiling for rectangular tiles [98, 20, 134] can be directly cast as an IGP. Other models that can also be reduced to IGPs include optimal orthogonal tiling [11], 2D semi-oblique tiling [10] and the Multi-level tiling model for 3D stencil computations [103].

- **Register tiling, auto-tuners, and Multi-level cost models.** The register tiling models proposed in [115] and [102] can be reduced to IGPs. The cost model used for generating high performance BLAS as described in [138] and the multi-level cost model [85] used for quantifying the multi-level interactions of tiling, can also be directly reduced to IGPs.

The fact that such a large number of TSS models—proposed across two decades by a several different authors—can all be reduced to single framework shows the generality and wide applicability of the GP framework. The fact that the functions used (without the knowledge of posynomials) in these models turn out to be posynomials indicates their suitability for TSS and makes one wonder whether they could be the language of optimal tiling!

Cost Model Reference	Cost function used for selecting the optimal tile size
ESS [45]	$C/(h*w)$
LRW [77]	$1/h+1/w+(2h+w)/C$
TSS [36]	$(2h+w)/h*w$
EUC [107]	$1/h+1/w$
MOON [87]	$1/h+1/w+(h+w)/C$
TLI [33]	$1/h+1/w+(h+w)/C+h*w/C^2$
WMC [128]	$C/h*w$
MHCF [85]	$(1/h+1/w)(1/n+1/l)+2/(h*w)$

Table 5.2.
Cost functions used in the literature for optimal cache locality tiling are shown, where C is the cache size, h, w represent the height and width of the rectangular tile, n represents the size of a 2D array and l represents the cache line size. A simple inspection shows that they are all posynomials. This table is derived from Hsu and Kremer [59, table 2].

5.4 Models From Literature

In the following sections we discuss in detail five models from a variety of tiling contexts. The goal is to provide an intuition for why all these models use posynomials and how the optimal TSS problems can be cast as an GP. Our discussion and reasoning about the posynomial nature of the functions used in these models are limited by the amount of details publicly available about them. The following models were chosen for detailed discussion because of their generality, use in production compiler, or uniqueness.

5.4.1 Cache locality model

In this section we show how the cost model proposed by Sarkar and Meggido [116], also used in the IBM XL FORTRAN compiler [114], can be reduced to an IGP. This cost model is applicable to a general class of loops and to tiling of double or triple loops. We chose this cost model for detailed discussion because of its applicability to a general class of loops and its use in a production compiler. Our description is aimed at illustrating how their formulation directly maps to an IGP. Further, we also illustrate how a change in the number of loops tiled affects the structure of their cost model and necessitates a new solution method. Whereas such changes can be easily accommodated in our GP based framework.

The overall strategy of Sarkar and Meggido [116] is to estimate the average memory cost per

```
parameter (n=400)
real*4 b(n,n), c(n,n)
real*8 a(n,n)
do i = 1,n
 do k = 1,n
  a(j,i)=a(j,i)+b(i,k)*c(k,j)
 end do
end do
```

Effective Cache Size (ECS) = 512 lines.
Cache line size (L) = 128 bytes.
TLB page size = 4096 bytes.
Effective TLB size = 128 entries.
Cache miss penalty (μ_c) = 15 cycles.
TLB miss penalty (μ_p) = 50 cycles.

$$\begin{aligned} \mathsf{DL}(s_i, s_k) &= s_i + \left\lceil \frac{4s_i}{128} \right\rceil s_k + \left\lceil \frac{4s_k}{128} \right\rceil \\ &\approx s_i + \frac{(s_i+31)s_k}{32} + \frac{s_k+31}{32} \end{aligned} \tag{5.2}$$

$$\begin{aligned} \mathsf{DP}(s_i, s_k) &= \left\lceil \frac{3200 s_i}{4096} \right\rceil + \left\lceil \frac{4s_i + 1600 s_k}{4096} \right\rceil + \left\lceil \frac{4s_k}{4096} \right\rceil \\ &\approx \frac{100 s_i + 28}{128} + \frac{s_i + 400 s_k + 623}{1024} + \frac{s_k + 1023}{1024} \end{aligned} \tag{5.3}$$

$$\begin{aligned} \text{minimize} \quad & \frac{\mu_c \times \mathsf{DL}(s_i, s_k) + \mu_p \times \mathsf{DP}(s_i, s_k)}{s_i s_k} \\ \text{subject to} \quad & \mathsf{DL}(s_i, s_k) \leq \mathsf{ECS} \\ & 1 \leq s_i, s_k \leq 400 \\ & s_i, s_k \in \mathbb{Z} \end{aligned} \tag{5.4}$$

Figure 5.1.
This figure is based on the example given by Sarkar and Meggido [116]. Example loop nest and hardware parameters are shown on the left. The optimization problem (Eq. 5.4) for selecting the tile sizes is shown on the right.

iteration and select the tile sizes that minimize it. The memory cost of a tile ($\vec{s}$) is calculated as $\mu_c \times \mathsf{DL}(\vec{s}) + \mu_p \times \mathsf{DP}(\vec{s})$, where $\mathsf{DL}(\vec{s})$ and $\mathsf{DP}(\vec{s})$ are the number of distinct cache lines and pages touched by a tile, respectively, and μ_c and μ_p are the cache and TLB miss penalties, respectively. The average memory cost per iteration is calculated by dividing the memory cost of tile by the tile volume.

Consider tiling the two (i and k) loops (cf. Figure 5.1) with a tile of size $s_i \times s_k$. A row of a is computed using a column of c and s_i rows with s_k columns of b. The arrays are laid out in column major order and observe that the line size (128 bytes) is much smaller than column size (n). Since every access to a will come from a distinct line, there will be s_i lines touched to access a row of s_i elements. On the other hand, since we access a column of c, it will touch consecutive memory locations and hence will hit $\left\lceil \frac{4s_k}{128} \right\rceil$ lines, where 4 denotes the bytes per array element and 128, the cache line size. A similar analysis will show that accesses to b will hit $\left\lceil \frac{4s_i}{128} \right\rceil s_k$ lines. The sum of these three quantities is $\mathsf{DL}(s_i, s_k)$, (cf. Eq. 5.2). A similar reasoning with the TLB page size yields $\mathsf{DP}(s_i, s_k)$. Further details can be found in the original papers [116, 114]. To the make the functions DL() and DP() tractable the authors use the following continuous approximation:

$\left\lceil \frac{a}{b} \right\rceil \approx \frac{a+b-1}{b}$, when $b > 0$. These approximations are also shown in Eqns. (5.2) and (5.3). The resulting optimization problem for picking the tile sizes is shown in Eq. (5.4).

A closed form solution for tiling double loops is derived by exploiting the following observation: the objective function of the optimization problem has the structure $\frac{A}{s_1} + \frac{B}{s_2} + \frac{C}{s_1 s_2} + D$, where A, B, C, and D are constants and s_1 and s_2 are tile sizes. One can observe this structure in Eq. (5.4). However, when tiling three loops this structure of the objective function is lost—the tile volume $s_1 s_2 s_3$ appears in the denominator of a term (cf. Sarkar and Meggido [116]). Due to this, a closed form solution is no longer available. Hence, for finding optimal tile sizes when three loops are tiled, they resort to a search based algorithm. This is a classic example of sensitivity of the solutions to the structure of the functions used in the cost model: an extension form double loops to triple loops requires a different solution method.

The optimization problem formulated by Sarkar and Meggido [116] for tiling double or triple loops can be reduced to an IGP. The key observation behind this reduction is the posynomial property of the functions used in the objective function and the constraints. First observe that the variables (s_i, s_k) take only positive values, and all the parameter constants (ECS, μ_c, μ_p,etc.) are also positive. Further, both $\mathsf{DL}(s_i, s_k)$ and $\mathsf{DP}(s_i, s_k)$ (with the continuous approximation) are posynomials. Using the property that posynomials are closed under addition and division by monomial, it is easy to verify that the objective function of Eq. (5.4) is also a posynomial. The constraints in Eq. (5.4) can all be easily brought to the required GP form (cf. Eq. (5.1)). The integer constraints on s_i and s_k makes the GP an IGP. Hence the optimal TSS problem given by Eq. (5.4) is an IGP. Due to the posynomial nature of DL() and DP(), the reasoning directly applies to the whole class of loops considered by them. Generalization to the case when triple loops are tiled is straight forward based on the closure properties of posynomials and monomials. For example, one can directly observe that the example optimization problem for tiling 3 loops given by Sarkar and Meggido [116, Figure 4] can be cast as an IGP.

5.4.2 Parallelism model

Andonov et al. [10] use a detailed cost model, with total execution time as objective function, for optimal TSS of 2D parallelogram iteration spaces, often found in stencil computations. The detailed cost model and the general constrained optimization based approach motivates us to

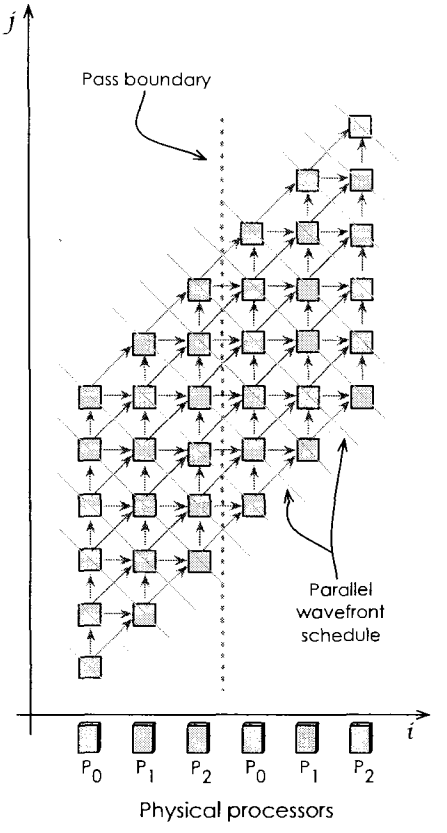


Figure 5.2.
A tile graph is shown resulting from a 2×2 tiling of the parallelogram iteration space is shown.

choose this model for a detailed discussion of its reduction to an IGP. We will describe their model and will show how their problem formulation reduces to an IGP. Further, when we extend the iteration space model to 3D parallelepipeds, their solution is not applicable. Whereas, the IGP approach directly accommodates such an extension.

Figure 5.2 shows a tile graph—nodes are tiles and edges are dependencies between tiles—resulting from a 2×2 tiling of a parallelogram shaped iteration space. Such iteration spaces result from a skewing transformation of loops to make rectangular tiling valid. Tile graph is a suitable abstraction for deriving a model of the parallel execution time. Also shown in the tile graph is the allocation of tiles to processors. Observe that the allocation is load balanced—all processors are allocated an (almost) equal number of tiles. The diagonal lines show the parallel schedule under which the processors execute the tiles.

The total execution time, T, can be modeled as the sum of the latency and the computation time of the last processor: $T = L + (\mathsf{TPP} \times \mathsf{TET})$, where, L denotes the latency of the last processor to start, TPP denotes the number of tiles allocated per processor, and TET is the time to execute a tile (sequentially) by a single processor. Here, the term $\mathsf{TPP} \times \mathsf{TET}$ denotes the time any processor takes to execute all the tiles allocated to it. Given that we have a load-balanced processor mapping, this term is same for all processors. In the following derivations, P is the number of physical processors, N_i and N_j denote the size of the iteration space along i and j, respectively and s_i and s_j are the tile sizes along i and j respectively.

The time to execute a tile, TET, is the sum of the computation and communication time. The computation time is proportional to the area of the rectangular tile and is given by $s_i \times s_j \times \alpha$. The constant α denotes the average time to execute one iteration. The communication time is modeled as an affine function of the message size. Every processor receives the left edge of the tile from its left neighbor and sends its right edge to the right neighbor. This results two communications with messages of size s_j, the length of the vertical edge of a tile. The cost of sending a message of size x is modeled by $\tau x + \beta$, where τ and β are constants that denote the transmission cost per byte and the start-up cost of a communication call, respectively. The cost of the two communications (a send and a receive) performed for each tile is $(\tau s_j + \beta)$. The reason for accounting for the cost of a single call is because typically a non-blocking send call is used and its cost is hidden. The total time to execute a tile is now $\mathsf{TET} = s_i s_j \alpha + (\tau s_j + \beta)$. The number of tiles allocated to a processor

is equal to the number of columns allocated to a processor times the number of tiles per column: $\mathsf{TPP} = \frac{N_i}{s_i P} \times \frac{N_j + s_i}{s_j}$.

The dependencies in the tile graph induce the delay between the start of the processors. The slope $\sigma = \frac{s_j}{s_i}$, known as the *rise*, plays a fundamental role in determining the latency [58, 10]. The last processor can start as soon as the processor before it completes the execution of its first two tiles. Formally, the last processor can start its first tile only after $(P-1) \times (\sigma + 1)$ time steps. Since at each time step a processor computes a tile, $(P-1) \times (\sigma+1) \times \mathsf{TET}$ gives the time after which the last processor can start, i.e., $L = (P-1) \times (\sigma+1) \times \mathsf{TET}$. To ensure that there is no idle time between passes, we need to constrain the tile sizes such that by the time the first processor finishes its column of tiles, the last processor must have finished its first tile. The time the first processor takes to complete a column of tiles is equal to $\frac{N_j+s_i}{s_j} \times \mathsf{TET}$ and the time by which the last processor would finish its first tile is $((P-1+1) \times (\sigma+1)) \times \mathsf{TET}$. The no idle time between passes constraint is $\frac{N_j+s_i}{s_j} \times \mathsf{TET} \geq (P \times (\sigma+1) \times \mathsf{TET}$. Using the terms derived above we can now formulate an optimization problem to pick the optimal tile size.

$$\begin{aligned} \text{minmize} \quad T = & \left[((P-1)(\sigma+1)) + \left(\frac{N_i}{s_i P} \times \frac{N_j + s_i}{s_j}\right)\right] \\ & \times (\alpha s_i s_j + (\tau s_j + \beta)) \\ \text{subject to} \quad & \frac{N_j + s_i}{s_j} \geq P(\sigma+1),\ s_i, s_j \geq 1,\ s_i, s_j \in \mathbb{Z}. \end{aligned} \tag{5.5}$$

The solution to the above optimization problem yields the optimal tile sizes, i.e., the tile sizes that minimize the total execution time of the parallel program, subject to the constraint that there is no idle time between passes. Andonov et al. [10] propose a closed form solution obtained through a detailed case by case analysis of the above optimization problem. This analysis exploits the structure of the objective function and constraints to find closed form solution. We can transform the optimization problem given in Eq. *5.5* to an IGP. The objective function T is directly a posynomial. With the approximation of $N_j + s_i \approxeq N_j$ the constraint transforms into

$$\frac{P(\sigma+1)s_j}{N_j} \leq 1$$

which is the required form for a GP constraint. Adding to it the obvious constraints that tile sizes are integers and positive, i.e., $s_i, s_j \in \mathbb{Z}$, $s_i \geq 1$ and $s_j \geq 1$, we get an IGP.

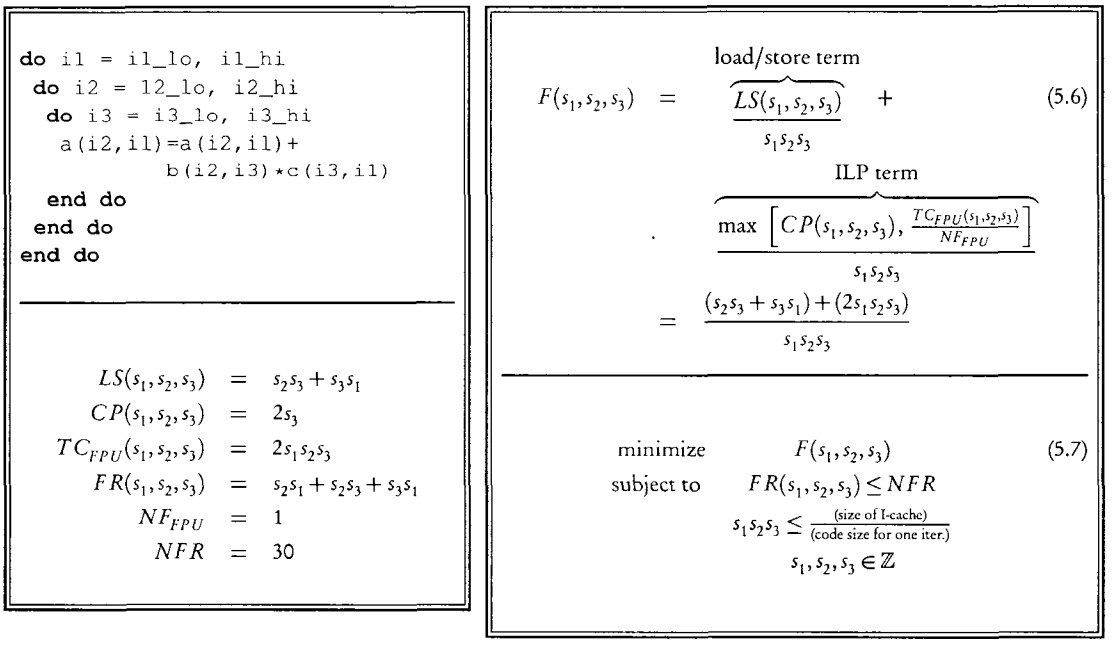


Figure 5.3.
This figure is based on the example of Sarkar [115]. The example code for matrix multiply and some of the terms used in the problem formulation are shown in the left. The optimization problem for selecting the tile sizes is shown on the right.

Stencil computations with 2D or 3D data grids, after skewing to make rectangular tiling valid, have 3D or 4D parallelepiped iteration spaces. Though the above cost models can be extended to model these higher dimensional cases, extending the solution method to find a closed form solution is not straight forward at all—and is still an open problem. On the other hand, a solution via IGP naturally accommodates such extensions based on the posynomial properties of the extended cost model. We have proposed one such extension [103] and it is discussed in detail in Chapter 6.

have proposed one such extension for stencil computations with 3D iteration spaces and it can be directly cast as IGP.

5.4.3 Register tiling model

Loop unrolling is used to increase instruction level parallelism (ILP) and enable register promotion. The unrolled iterations have multiple copies of the loop body, and expose the array references for scalar replacement, a technique used for register promotion [25]. We can view loop unrolling as tiling for registers and ILP. In fact, the legality condition for unroll-and-jam and tiling

is the same [115]. Further, choosing the unroll factors can be viewed as selecting tile sizes—note that they both can take only positive values. The I-cache and register requirements can be modeled as capacity constraints. Such a view leads to a more general formulation of the loop unrolling problem as shown by Sarkar [115]. In fact, it is the generality of formulation and the detailed cost model that motivates us to choose this for detailed discussion. We present this general formulation of Sarkar [115], viewed as a tiling transformation. We present the complete formulation for an example and show how the resulting optimization problem for selecting the tile sizes (unroll factors) can be cast as an IGP. The reasoning about the functions used in this formulation, directly generalizes to the whole class of loops considered by Sarkar [115].

The overall approach is to find the tile sizes that minimize the average cost per iteration subject to the capacity constraints. Two kinds of capacity constraints are considered, viz., the register and I-cache. As shown in Figure 5.3 (Eq. 5.6) the objective function, F, that measures the cost per iteration is the sum of an ILP term and a load/store term averaged over the tile volume $s_1 \times s_2 \times s_3$. All functions take the tile sizes, s_1, s_2 and s_3 as arguments and estimate quantities related to the unrolled loop body. The load/store term, $LS()$, estimates the number of cycles spent on load and store instructions. The ILP term estimates the parallel execution time of the unrolled loop body. Intuitively, the parallel execution time is the maximum of the critical path length in the unrolled body, $CP()$, and the number of cycles spent on functional units. For the example, we have only floating point operations and hence only floating point functional units ($TC_{FPU}(), NF_{FPU}$) and floating point registers ($FR(), NFR$) are considered. The estimated values of all the functions for our example are shown in Figure 5.3. The number of floating point registers required by the unrolled loop body, $FR()$, is estimated by counting the number of loop invariant references to array `a` (equal to $s_2 s_1$) and the number of distinct values of arrays `b` and `c` (equal to $s_2 s_3 + s_3 s_1$). The estimation is based on the Ferrante et al. [47], which is also used in the context of tiling for data locality presented earlier (cf. Section 5.4.1). A detailed explanation on how the functions are estimated is given by Sarkar [115]. The optimal tile sizes are found by an enumeration based search algorithm which uses the objective function $F()$ to evaluate the merits of each tile size vector. The algorithm enumerates all feasible tile sizes (those that meet the capacity constraints) and for each one of them calculates the value of $F()$, and then selects one that minimizes $F()$.

The optimization problem for selecting the tile sizes given in Figure 5.3 (Eq. 5.7) can be di-

$$B_k = HW\left(\frac{1}{H}+\frac{1}{S_k}+\frac{1}{N}\right) \quad (5.8)$$

$$M_k = N^3\left(\left(\frac{1}{H}+\frac{1}{W}\right)\left(\frac{1}{N}+\frac{1}{S_k}\right)+\frac{2}{HW}\right) \quad (5.9)$$

$$\begin{array}{lll} \text{minimize} & E = & i_t M_t(H,W)+i_c M_c(H,W) \quad (5.10) \\ \text{subject to} & & B_t(H,W) \le 0.75C_t \\ & & B_c(H,W) \le 0.75C_c \\ & & H,W > 1 \text{ and } H,W \in \mathbb{Z}. \end{array}$$

Figure 5.4.
A Multi-level (TLB and cache) cost model for single-level tiling from Mitchell et al. [85]. i_k is the miss penalty for memory module k and C_k is the capacity of memory module k. Types of memory modules are TLB and cache and denoted by $k = t$ and $k = c$.

rectly cast as an IGP. This can be verified by observing that all the functions, parameters and constants used in its formulation are posynomials . This is due to the fact that these functions have tile sizes as variables and architectural parameters as constants, both of which are positive. The presence of max() in the objective function $F()$ is not problem. It can be eliminated easily by introducing new variables [21, 101].

Again, the above reasoning generalizes to the whole class of loops considered by Sarkar [115]. For the load store function, $LS()$, and the functions that model the register and I-cache requirements, this generalization can be easily verified with the details in Sarkar [115] and Ferrante et al. [47]. Sarkar [115] does not give enough details about the estimation of the functions that model the critical path length, $CP()$, and cycles spent on resource classes, $TC_j()$, and hence, we do not know whether there are cases for which these functions are not posynomials. However, we expect these functions to be posynomials, since they use positive quantities—tile sizes and machine parameters—to model another positive quantity—the number of cycles. To summarize, we have shown, based on the available details, that their formulation can be cast as an IGP.

5.4.4 Multi-level tiling model

Mitchell et al. [85] use three examples to show the need for multi-level cost functions even when tiling for just one level. In this section, we present one of them, viz., a multi-level cost model that captures the interactions between TLB and caches. The example used is matrix multiplication with (k,i,j) as the outer schedule and (i,k) as the inner schedule. The j loop is not tiled and i and k loops are tiled with tile sizes H and W, respectively. Square matrices of size N are considered.

Figure 5.4 shows the multi-level cost model. For a given memory module k, the function B_k

$N_B \times N_B$= Cache tile size
$N_U \times M_U$= Register tile size
C_1, B_1 = Cache capacity and line size
N_R= Number of registers
L_s= Skew factor for scheduling

$$F_1 = \left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B N_U}{B_1} \right\rceil + \left\lceil \frac{M_U}{B_1} \right\rceil N_U \leq \frac{C_1}{B_1} \quad (5.11)$$

N_B is a multiple of M_U, N_U, and 2. (5.12)

$$F_2 = M_U N_U + N_U + M_U + L_s \leq N_R \quad (5.13)$$

Figure 5.5.
Cost functions used by Yotov et al. [138, Figure 20] to select the cache and register tile sizes.

(Eq. 5.8) estimates the number of blocks required to hold a $H \times W$ sub matrix, and M_k (Eq. 5.9) estimates the number of misses. The optimization problem (Eq. 5.10) is formulated using these functions and other constants, as described in the caption of Figure 5.4. Note that the objective function E accounts for misses at both the TLB and cache levels and the constraints on B_t and B_c account for both TLB and cache capacities. They have found that the $B_t(H, W) \leq 0.75C_t$ constrains the width W more tightly and $B_c(H, W) \leq 0.75C_c$ constraints the height H more tightly. They derive closed form solution for this optimization problem.

The optimization problem (Eq. 5.10) can be directly cast an IGP, as shown in the following reasoning. Based on the structure of the functions B_k and M_k it is evident that they are posynomials. The positivity of the constant parameters, i_t, i_c, C_t, and C_c implies that the objective function is a posynomial and the constraints can be put into posynomial inequalities.

5.4.5 Auto-tuner model

Auto-tuners such as ATLAS [126] automatically generate and tune high-performance libraries. Model driven empirical search is used by these auto-tuners to select parameter values. Tile sizes are one of the important parameters tuned in these libraries. For example, cache and register tile sizes are parameters tuned by ATLAS and PHiPAC. Yotov et al. [138] propose detailed models that can be used in auto-tuners for high performance BLAS [1]. They propose models to tune the matrix multiply routine, which is at the heart of level 3 BLAS. We use these models to show the appropriateness of posynomials in modeling cost functions used in auto-tuners. We describe the models they use for selecting the cache and register tile sizes and show that they are posynomials. They do not define an optimization problem with an objective function but rather use the cost functions as constraints to guide the search for the parameters.

Figure 5.5 shows the cost functions F_1 and F_2, used for selecting the cache and register tile sizes,

respectively. Observe that the cache tile shape is restricted to squares and hence there is only one variable N_B, however the register tile shapes are rectangles and hence we have two variables N_U and M_U. They first solve for M_U and N_U using Eq. 5.13, and substitute the solution in F_1 to make it a function of just N_B. After this substitution F_1 becomes a quadratic function which can be solved directly. With a continuous approximation of the ceilings in F_1, we can see that both F_1 and F_2 are posynomials. Further, the constraint that N_B has to be a multiple of N_U, M_U and 2 can be easily cast a monomial constraint. For example, the constraint that N_B is a multiple of 2 is equal to the following monomial constraint: $\exists k > 0, k \in \mathbb{Z} : N_B = 2k \iff \frac{N_B}{2k} = 1$. In fact, with a suitable objective function, one can even use F_1 and F_2 to build a multi-level cost model, cast it into a GP, and solve for N_B, N_U, and M_U simultaneously.

5.5 PosyOpt Framework

We have implemented the optimal TSS framework as a tool called PosyOpt. The implementation uses MATLAB and YALMIP [82] a tool which provides a symbolic interface to several optimization tools on top of MATLAB. The symbolic interface allows a high level specification of the optimal TSS problems. The overall structure of our tool PosyOpt is shown in Figure 5.6. The optimal TSS problems are specified at a high level using posynomials as a IGP. These problems are then automatically transformed to a convex optimization problem. The transformed problem is then fed to the convex optimization solver of MATLAB and solved for real solutions. Integer solutions are found via a branch-bound algorithm which internally uses the MATLAB solver for solving continuous relaxations. The output of our tool is the set of optimal tile sizes.

Note that the specification and subsequent refinement and extensions are performed at the posynomial level (*cf.* top box in Figure 5.6). These steps are done without any concern about the solution method. The only concern is that the specifications and extensions use posynomials and formulate GPs. Further, as shown in the Figure 5.6 (top box) different models can be combined or composed together to form multi-level tiling models. We have found the closure properties of monomials and posynomials to be very useful during the extensions and compositions.

We envision three different users for our tool: (i) modelers would use it for designing cost models for TSS, (ii) Auto-tuners (such as ATLAS [126]) and model-driven empirical search meth-

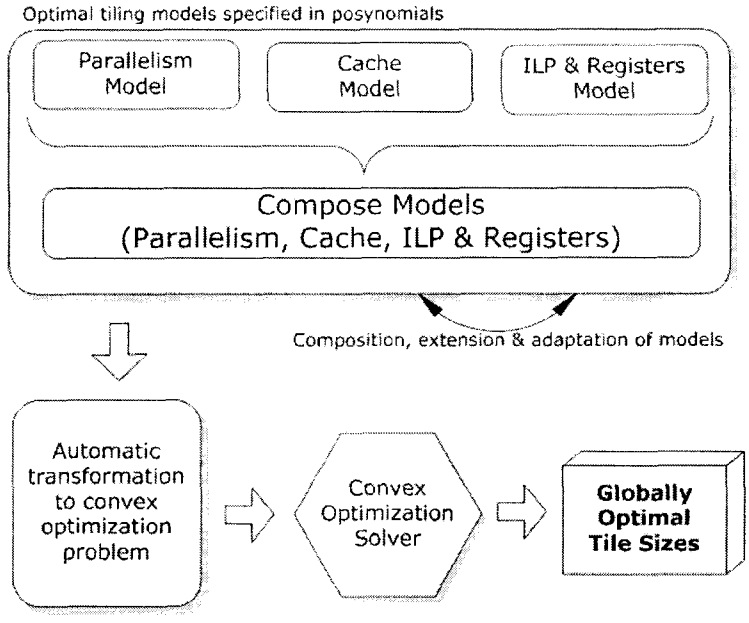


Figure 5.6. Overall structure of the PosyOpt tool.

ods [138] would use it to pick a good initial solution and then explore the neighborhood to refine the solution, and (iii) compilers would use it for statically selecting a good tile size. The current choice of MATLAB for implementation suits very well for uses in modeling and auto-tuners. The modular structure of our tool allows the replacement of MATLAB based symbolic interface by any other tool. When our tool is used in the context of a production compiler, there will not be any need for a symbolic interface, the compiler would be using a particular chosen model and for this given model, as explained in Sec. 5.5.1, we can directly solve for the optimal solutions.

5.5.1 Running time experiments

Using our tool, we formulated and solved a variety of single level and two level TSS problems. The number of variables in any optimization problem is determined by the loop nest depth and the number of levels of tiling. For example, a 2D loop nest tiled twice, would have 4 variables in the TSS problem. The problems we experimented had 2 to 9 tile size variables and up to twenty constraints. The time our tool takes to find the integer solutions range from 10 to 50 microseconds. Note that our tool uses a symbolic interface and the reported timings include the overhead of symbolic preprocessing, transformation to convex optimization problem format and

calls to the solver. Also note that if we choose to find just real solutions (and later round them) the solving times are much faster. Since the feasible space for all GPs are convex regions, we found in almost all the cases, the integer solution can be obtained via a rounding of the real solution. For the uses in the context of a auto-tuner or designing a model, the current speed of our tool seems very reasonable, particularly given the ease with which the problem can stated and solved.

In our experience with using our tool (and MATLAB), we found many additional optimizations are possible. In the context of a production compiler, a particular TSS model is used. Given this, there is no necessity to actually go through a symbolic interface, and we can directly generate the convex optimization problem in the required (matrices and vectors) format. Second, we can directly use the C language function call interface to call the solver. Third, for any given model, we can study the region of the feasible space where the optimal solution lies and choose to just solve for real solutions and round them to obtain integer solutions.

5.6 Conclusions

We have proposed a framework based on a simple yet fundamental property of functions used in optimal TSS models. Our framework not only generalizes the TSS models proposed in the literature, but also provides the foundation for developing more sophisticated and particularly multi-level tiling models. The ability to compose well understood single level models to form multi-level models allows reuse of the knowledge developed by several researchers across two decades. Our tool, PosyOpt is well suited for building auto-tuners and model-driven/iterative optimizers. A simplified version is ideal for inclusion in compilers. We are currently benchmarking the GP solvers with respect to the models collected in our repository and studying whether real solutions are sufficient.

CHAPTER 6

Exploration of Parallelization Strategies for 3D Stencil Computations

THIS chapter presents an use of posynomials and GPs for finding optimal tiling and parallelization strategies for stencil computations. The key idea is to characterize the space of legal tilings and useful parallelizations, and explore this space by exploiting the fast solution methods available for solving GPs. This exploration, of not just the tile sizes but tiling and parallelization strategies plus the tile sizes, is an example of the wider class of optimizations that are enabled by the use of efficient solution methods provided by the PosyOpt framework. Such explorations have the potential for discovering new parallelization strategies. We show that even a partial exploration of the space of parallelization strategies lead to strategy which is up to a factor of two faster than the standard implementation.

The work presented in this chapter was done in collaboration with Manjukumar Harthikote-Matha and Rinku Dewri. It was presented in [103].

6.1 Introduction

Stencil computations form the basis for a wide range of scientific applications from simple Jacobi to complex multigrid solvers. Their inclusion in major benchmarks like SPEC [119], HPF-

BENCH [61], PARKBENCH [92], and NAS Parallel Benchmarks [88], clearly show their importance. The development of special purpose stencil compilers [23] and implementation of pattern matchers in general compilers [113] to identify stencil computations, highlight the potential for performance improvements from loop transformations and optimizations.

Tiling [62, 135, 136] is a loop transformation that can be used for (i) partitioning data and computations among parallel processors and (ii) reordering computations within a single processor to improve data locality. For stencil computations a variety of multi-level tiling schemes are possible. For example, consider just two levels of tiling: an outer level for parallelism and an inner level for data locality. For every outer level tiling strategy, many parallelizations are possible, and for each such parallelization, several inner level (for locality) tiling strategies are possible. The best schemes are those with lowest execution times, which depend on optimal choices of tiling and parallelization strategies and parameters. Not only are there many such schemes, for each of them the space of the tile sizes is also huge. The global question is *which combination of tiling and parallelization strategy with which parameters produces the minimum running time for a given set of program size parameters and a given parallel machine?* It is time consuming and error prone to develop parallel implementations for each combination of tiling and parallelization scheme and experiment with them to find a good one, or to even eliminate the obviously poor ones.

There have been extensive studies [78, 109, 81, 133, 49, 68, 67] on tiling stencil computations for locality. Schemes for tiling stencil computations for parallelism can be classified based on whether or not they tile the outermost time loop. The commonly used data partitioning scheme [52] does not tile the time loop and uses the "owner-computes" rule to determine the computation distribution. Early work by Wolfe [131] shows that skewing can be used to enable tiling of the time loops. Recently, Wonnacott [132] shows that time skewing can be used to tile for parallelism as well as locality. Several important issues are not addressed by these authors. For a given stencil computation,

- what is the space of legal tiling and parallelization schemes?
- what are the trade-offs between these schemes?
- how do the tiling choices at the parallelization level affect the choices at locality[1]?

[1]Mitchell *et al.* [85] point out that ignoring such tiling interactions will lead to suboptimal solutions.

- what are the globally optimal tile sizes?

A study of these issues will enable us to develop high performance, multi-version, platform specific implementations of stencil computations. As an analogy, consider the matrix multiplication code generated by ATLAS [126]. The generated final code has different versions for different shapes of matrices, and makes several platform specific choices for optimizations. Our experiments show that stencil computations are similar, i.e., the optimal strategy depends on the shape of the domain (size of the grid and the number of time steps). We envision a tool that explores the space of legal tiling and parallelization schemes, selects optimal parameters and generates a multi-version high performance implementation of a given stencil computation. As a first step towards such a tool, the work presented in this chapter makes the following contributions.

- We characterize the space of possible legal tilings and load balanced parallelizations for 2D/3D Gauss-Siedel 9-point stencil. We focus on two candidates from this space to illustrate the need to explore this space. Even this partial exploration led us to derive a new strategy which is up to a factor of two faster than the standard implementation.
- We develop analytical models for the parallel execution times of the two strategies. We formulate a constrained optimization problem for the optimal tile sizes and transform it to a convex optimization problem, which can be solved efficiently.
- For both the strategies, we study an additional level of tiling for locality and analyze the interactions between the choices at different levels.
- We experimentally validate our analytical models. We discuss the performance improvements and trade-offs obtained with various strategies. We show how the best strategy depends on the shape of the stencil iteration space. This leads to a division of the input space into regions where different strategies perform better.

In the next section we characterize the space of legal tilings and parallelizations. In Sections 6.3 and 6.4 we discuss in detail the tilings and parallelizations for the two strategies and derive analytical models for their execution times. We present experimental validation and discuss performance improvements and trade-offs in Section 6.5. We discuss related work in Section 6.6 and present our conclusions and future work in Section 6.7.

```
for k = 1...N_k
  for i = 1...N_i
    for j = 1...N_j
      A[i,j] = ω * (A[i-1,j] + A[i-1,j-1] +
                    A[i,j-1] + A[i-1,j+1] +
                    A[i+1,j] + A[i+1,j+1] +
                    A[i,j] + A[i,j+1] + A[i+1,j-1])
```

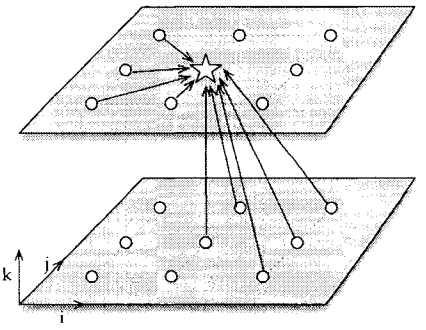


Figure 6.1.
(Left) Gauss-Siedel style successive over-relaxation code. 9 point stencil computation. (Right) Dependences of the 9 point stencil computation.

6.2 Space of Tiling and Parallelizations

We consider 2D/3D stencil computations in which a two dimensional data grid of size $N_i \times N_j$ is updated iteratively over N_k time steps. We call N_i, N_j, and N_k as the loop size parameters and let $\vec{N} = (N_i, N_j, N_k)$. As a representative of this class (3D stencils) we consider the Gauss-Siedel 9 point stencil computation given in Figure 6.1 (left). The computation domain is a 3D cuboid of size $N_i \times N_j \times N_k$. A graphical view of the nine dependences are shown in Figure 6.1 (right). Gauss-Siedel (in place updates) stencils are expected to have faster convergence than the Jacobi stencils, which use all the 9 values from previous time steps. On the other hand, the dependences of the Jacobi stencil are easier to tile and/ or parallelize. We consider the difficult (to tile and parallelize) but faster converging Gauss-Siedel stencils. Our characterization and models are directly adaptable and applicable to other types of 2D/3D stencils.

6.2.1 Tiling and parallelization model

Tiling [62, 135] partitions the iteration space into groups which are executed in an atomic fashion – all iterations in a given tile are executed by a processor before any iteration of its next tile. Note that this notion of atomicity still permits any legal (re)ordering of the computation and communication steps within a tile. A *rectangular tiling* is one where rectangles are used for partitioning. We consider rectangular tiling possibly preceded by a skewing transformation to make it legal. We denote the tile sizes along the dimensions i, j, and k of the 3D iteration space by s_i, s_j, and s_k,

respectively. The *tile graph* consists of nodes representing tiles and edges between them representing the dependences between tiles. It is well known that [10, 136] if the s_i's are large as compared to the elements of the dependence vectors of the original loop, then the dependencies between the tiles are *unit vectors* (or binary combinations thereof, which can be neglected for analysis purposes without loss of generality). An important property is that the tile graph with such unit dependence vectors can be viewed as an n-dimensional system of uniform recurrence equations [69]. Such a view allows us to use the powerful systolic array synthesis methods [95, 96] to formally reason about optimal parallelizations of the tile graph. In the context of exploring the space of possible tiling and parallelizations, such a formal reasoning helps in constraining the search space to a few valid and good candidates.

In any parallelization, the dependences in the tile graph induce some delay before which all the processors can start executing. We call this initial delay the *latency* of a parallelization strategy. Once all the processors begin to execute, any idle time incurred by a processor is a consequence of the chosen parallelization. We restrict ourselves to parallelizations that are free of such idle times. We call such parallelizations *idle-free*. We also restrict ourselves to allocations that are *load-balanced,* i.e., to ones that allocate an equal amount (except at boundaries) of computation to every processor. For the stencil computations this can always be achieved. Practical experience as well as our analytical models predict that optimal performance can only be achieved under such idle free load balanced parallelizations. Further, for allocation functions we restrict to orthogonal projections—ones that are parallel to some canonical axes.

To summarize, we consider rectangular tiling and idle-free load balanced parallelizations only. As shown in the later sections, the set of choices to be considered after these restrictions is still rich.

6.2.2 Need for and implications of skewing

Skewing is a loop transformation that changes the dependence distances in the stencil code. In the context of stencil computations, skewing is often used to transform the dependence distances into non-negative ones, thus making tiling legal. Given the dependences of the 9-pt stencil (*c.f.* Figure 6.1), tiling certain dimensions require certain skewing transformations to make it legal. However, as a side effect, skewing also changes the shape of the iteration space. For instance,

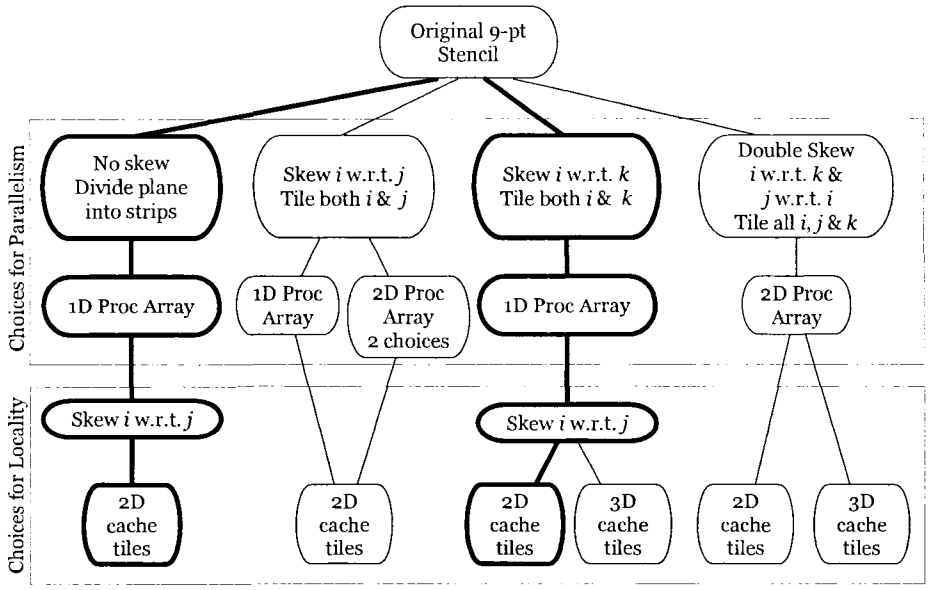


Figure 6.2.
Space of multi-level tilings and parallelizations for the 9-pt. stencil. The choices (path) shown in bold correspond to the two strategies explored in detail.

skewing a rectangular iteration space will make it a parallelogram. As a consequence, a rectangular tiling of the parallelogram iteration space will result in both full (rectangular) and partial (non-rectangular) tiles. Partial tiles increase the tiling overhead and also makes analytical modeling difficult. Hence, there is a trade-off: extra tiling overhead introduced by skewing versus ability to tile additional dimensions.

6.2.3 Space of tilings and allocations for parallelization

The space of possible rectangular tilings for the 9-pt stencil (*c.f.* Figure 6.1) corresponds to the choice of which and how many dimensions do we choose to tile. Note that we are not characterizing the space of tile sizes, which we will do later for each possible tiling. Tiling different dimensions requires a different set of skews of the iteration space. The choices and the corresponding skews are discussed below and a graphical view of them is shown in Figure 6.2 (top box).

1. **Tile the program with *no skewing*.** The program dependences limit such tilings. For example, in order to tile either the i or the j loops, s_k has to be 1. Furthermore, in order to

tile the j loop, s_i must also be 1. Thus the possible tilings are: (i) the trivial $1 \times 1 \times 1$, $N_k \times N_i \times N_j$ and $1 \times N_i \times N_j$ tiles which we discarded for obvious reasons; (ii) $1 \times s_i \times N_j$ tiles; and (iii) $1 \times 1 \times s_j$ tiles, which we discard because the computation-to-communication balance of the tiles is too low[2]. We pursue the $1 \times s_i \times N_j$ tiling. For this tiling strategy, a parallelization on an 1D processor array is the only choice, where the processors are aligned along the i axis (*c.f.* Figure 6.3 (left)). The choices are shown in the left-most branch of Figure 6.2.

2. **Tiling both i and j dimensions.** We need to skew i with respect to j to make tiling along j legal. This case also covers the case of tiling just along j when the tile size along i is 1. For this tiling, we can parallelize the tile graph on an 2D or 1D processor array. For an 1D processor array we align it along the i axis. For the 2D processor array, we can either align the processors along the ik-plane or the ij-plane. For the ij-plane alignment the processors are arranged in a parallelogram shaped grid and for the ik-plane alignment they are arranged in a rectangular grid. The choices are shown in the second branch (from left) in Figure 6.2.

3. **Tiling both i and k dimensions.** We need to skew i with respect to k to make tiling along k valid. Based on a similar reasoning as above, this choice also covers the tiling just along k. For this tiling, a parallelization on an 1D processor array is the only choice. The processors are aligned along the k axis as shown in Figure 6.4 (right). The choices for this strategy are shown in the third branch (from left) in Figure 6.2.

4. **Tiling all the three (i, j and k) dimensions.** We need to first skew i with respect to k and then skew j with respect to i. This choice also covers the case of tiling just j and k. For this tiling, with orthogonal processor allocations, only an 2D processor array is possible. The 2D processor array is aligned along the ik-plane. However, if we expand our space and consider non-orthogonal processor allocations, there are two linear array parallelizations possible[3]. We do not discuss these choices further. The choices related to the 2D processor array is shown in the right most branch in Figure 6.2.

[2]It would be easy to use the ideas in this chapter to confirm analytically and experimentally that this is indeed the case.

[3]These non-orthogonal projections make every communication non-local, which would result in higher communication costs. Based on this intuition we have restricted ourselves to orthogonal projections. It is not clear whether this is always globally optimal but it definitely makes the space more tractable.

One might wonder why the last choice above does not cover all the other cases by appropriately letting the corresponding dimensions (i, j, and/or k) equal to 1? The answer is, skewing creates partial tiles and leads to a different cost function for the total computation time. The overhead of partial tiles should be avoided whenever possible, so that we can derive simpler parallel implementations and more precise execution time models.

6.2.4 Space of tilings for locality

After an outer level of tiling for parallelism we can tile another level for locality. We call a tile from the outer level of tiling (for parallelism) as a *parallel-tile* and a tile from the inner level of tiling (for locality) a *cache-tile.* Correspondingly we also refer to their sizes as *parallel-tile sizes* and *cache-tile sizes.*

We have two choices for cache tiling: tile i and j dimensions only or tile i, j and k dimensions. Both may require additional skewing transformations to make them legal. This additional skewing is not required if it has already been done for the outer (parallelism) level tiling. Given that the data of the stencil is 2D, tiling just the i and j dimensions will allow us to exploit the limited amount of spatial locality. To exploit temporal locality we need to tile the (time) k dimension. The choices are shown in Figure 6.2 (lower box).

6.2.5 Interactions between tilings

Two types of interactions ensue, viz., (i) skewing transformations at parallelism level can enable or disable tiling along certain dimensions for locality, and (ii) the parallel tile sizes restrict the lower and upper bounds of the cache tile sizes. These interactions stem from the fact that a parallel-tile becomes the iteration space for the cache tiling. We describe below two instances where a decision made at the parallelism level affects the choices in the inner level.

Consider the case in which we do not tile the k loop at the parallelism level. This choice leads to parallel-tiles which are slices of the ij-plane and disables cache level tiling of the k loop. These slices become the iteration space for the cache-level tiling. So, we can see that the cache-tiles are forced to be 2 dimensional and hence can only exploit spatial locality. (Recall that we need to also tile the k dimension to exploit temporal locality.) The first two branches (from left) in Figure 6.2 shows this consequence — observe the leaves showing the possibility of only 2D cache tiles.

When tiling j loop is made legal by skewing transformations at the outer level, there is no additional skewing required at the inner cache-level to get 2D cache-tiles. This case is shown in the second and fourth branches (from left) in Figure 6.2. On the other hand, notice that for the strategies shown in the first and third branches (from left) in Figure 6.2, we need to skew i loop with respect to j to make tiling j loop legal and hence get 2D cache-tiles.

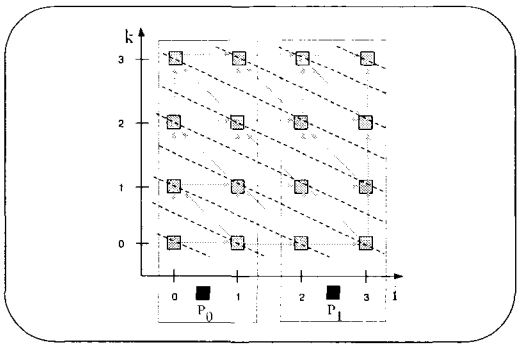
6.3 1D Strips

In this section we consider the first strategy (left most branch in Figure 6.2) and develop an analytical model for the parallel execution time. In the modeling, we use three parameters, viz., α, β and τ, to model the quantities that are dependent on the loop program and parallel architecture on which it is to be executed. α represents the time to execute an iteration of the given loop program. β and τ represent respectively the start up cost of a MPI communication call and the time to communicate a `double` precision data value.

In this tiling strategy, each tile is a $1 \times s_i \times N_j$ rectangular parallelepiped, i.e., only the i loop is effectively tiled. The j loop has a single "tile" of size N_j, and the k loop "tiles" have unit size. Because there is only one tile in the j dimension, the resulting tile graph can be viewed as a 2D grid in the (i,k) plane as shown in Figure 6.3 (left). The dependences between tiles are $[0,1]$ to the north, with a data "volume" of $N_j s_i$, and $[1,0]$ (east) and $[-1,1]$ (north-west), both with volume N_j.

To explore different parallelizations, we first derive the optimal wavefront schedule for the tile graph, which is $t(i,k) = i + 2k$. This schedule is shown as dotted lines across the tile graph in Figure 6.3 (left). It is optimal in the sense that the total execution time for this schedule (assuming unbounded processors) is $\frac{N_i}{s_i} + 2N_k - 1$, which equals the length of the longest path in the graph.

Next, we choose an appropriate allocation of tiles to (virtual) processors. For our rectangular tile graph, only two allocations lead to a load balanced parallelization, namely by columns, or by rows. Allocation by rows, where each processor sequentially executes all the tiles in a row of the tile graph, leads to a parallelization that allows multiple passes. We developed an analytical model for it and determined the optimal tile size. However, this parallelization is almost always outperformed by the column wise allocation, and is not described further in the interests of brevity.




```
for k = 1...N_k
  for each strip S
     receive Rcol[k] of p_{i-1}
     compute Lcol[k] of S
     send Lcol[k] of S to p_{i-1}
     compute MiddleRegion[k] of S
     receive Lcol[k-1] of p_{i+1}
     compute Rcol[k] of S
     send Rcol[k] of S to p_{i+1}
```

Figure 6.3.
(Left) Tile graph of 1D strips tiling. The fastest schedule is shown in dotted lines. **(Right)** Steps performed by each (non-boundary) processor in 1D Strips tiling. `Lcol[]`, `Rcol[]`, and `MiddleRegion[]` corresponds to the left column, right column and middle portion of a strip. The index k and $k-1$ indicates, respectively, whether they are from the same k plane or the previous plane.

Allocation by columns, where tile (i,k) is performed by virtual processor i, yields a parallelization (i.e., a "macro systolic array") that has bidirectional communication: processor i sends to $i+1$ for the $[1,0]$ dependence, and to $i-1$ for the $[-1,1]$ dependence. This has two important consequences.

- Every processor is active only on alternate time steps. This problem can easily be corrected by a well known systolic technique called clustering or serialization. We allocate two adjacent virtual processors to a single physical processor which alternates between the two tiles and is thus always busy. This combined two-tile unit is called a "macro tile" or a *Strip*.

- It precludes adaptation to run on fewer processors in multiple passes, using another common systolic technique called LPGS (for Locally Parallel Globally Sequential) partitioning [86]. This means that $s_i = \frac{N_i}{2p}$, i.e., each macro tile is a $\frac{N_i}{p} \times N_j$ strip.

A processor performs the following steps: receive data required to execute the strip, execute the strip and send computed data to neighbors. As a latency hiding optimization, we can relax the strict order between the receive-compute-send steps, and interleave them. In the execution of a strip, if we allow the processors to move the sends as early as possible and the receives as late as possible, we get the optimized code shown in Figure 6.3 (right). In this version, a processor receives the data to compute its left-most column, computes it and sends the new values immedi-

ately. Then it computes the middle region of the strip, receives the data to compute the right-most column, computes it, and sends the new values.

There are no tiling parameters to choose optimally, and the analytical model developed below predicts the running time for this parallelization. The single pass execution implies that the tile size s_i along i has to be $\frac{N_i}{P}$. Let p_i denote the i^{th} processor, and $p' = p_{P-1}$ the last processor. The total execution time of this tiling and parallelization can be modeled as $T_{\text{strip}} = \mathsf{Latency}(p') + \mathsf{TPP}(p) \times \mathsf{TET}(s_i)$. Where, P is the number of processors, $s_i = \frac{N_i}{P}$ is the tile size, $\mathsf{Latency}(p')$ is the latency for last processor to start, $\mathsf{TPP}(p)$ is the number of tiles allocated to any processor p, and $\mathsf{TET}(s_i)$ is the time to compute a tile. To compute the time to execute a tile, we observe that during the computation of a tile a processor performs $N_j s_i$ computations and communicates its left and right columns, of size N_j, to the previous and next processors. Hence, we have $\mathsf{TET}(s_i) = \alpha \times N_j \times s_i + 4(\tau N_j + \beta)$, where, α, τ, and β are as discussed earlier. Every processor is allocated N_k macro tiles (or strips), hence $\mathsf{TPP}(p) = N_k$.

The last processor can only start after it receives the right most column of its left neighbor, *i.e.*, p' can start after the first $P-1$ processors execute their tiles. Hence, $\mathsf{Latency}(p') = (P-1) \times \mathsf{TET}(s_i)$. By plugging in these functions we get

$$T_{\text{strip}} = (N_k + P - 1) \times \left(\alpha \left(\frac{N_i N_j}{P} \right) + 4(\tau N_j + \beta) \right) \tag{6.1}$$

6.3.1 Cache tiling

Each processor executes a set of parallel-tiles, each of size $\frac{N_i}{P} \times N_j$. This strip can be further tiled to exploit some limited amount of spatial locality. However, to make the tiling of the strip legal, we need to skew the j loop with respect to the i loop. We perform this transformation and then tile both the i and the j loop to obtain 2D cache-tiles. Note that the decision of not tiling the k loop at the outer level results in a situation where we cannot tile the k loop at the inner (cache) level, to exploit temporal locality. We select the best cache-tile sizes empirically, *i.e.*, by running the cache-tiled coded for several tile sizes and selecting the best.

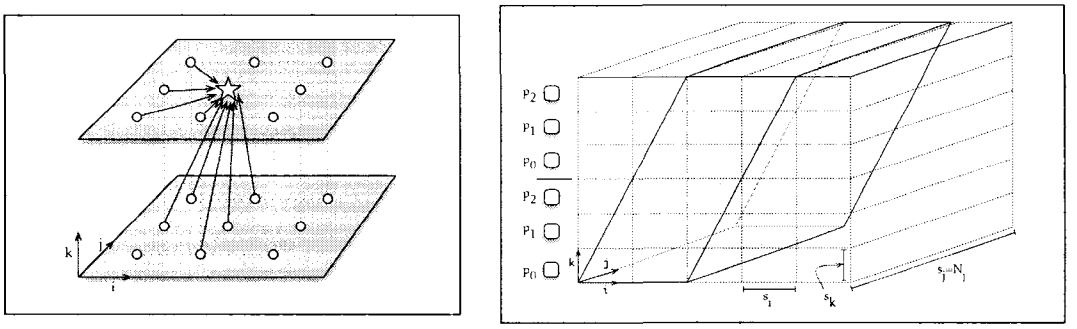


Figure 6.4.
(Left) Skewed dependences that make this tiling legal. **(Right)** Semi-oblique strips tiling.

6.4 Semi-oblique Strips

In this tiling strategy, we seek to tile the k and i dimensions. To make this tiling legal we first skew the i loop with respect to the k loop with the transformation $(k,i,j) \mapsto (k,i+k,j)$. The transformed dependences are shown in Figure 6.4 (left). We then tile the k and i loops with tile sizes s_k and s_i, respectively. We do not tile the j loop and allow $s_j = N_j$. The skewed iteration space together with the tiling is shown in Figure 6.4 (right).

We parallelize this tiled iteration space on a linear array of P processors aligned along the k axis as shown in Figure 6.4 (left). Note that depending on whether $\frac{N_k}{s_k} > P$ or not, there might be more than one pass. Every processor executes one or more rows (along i) of $\frac{N_i}{s_i}$ tiles of size $s_k \times s_i \times N_j$. Such an allocation is load balanced—all the processors execute the same amount of computation (assuming P divides $\frac{N_k}{s_k}$ evenly). During the execution of a tile, a processor receives the bottom $(i\,j)$ face of the tile from the processor below it, computes the tile, and sends the top $(i\,j)$ face to the processor above it. These faces communicated between processors are of size $s_i N_j$.

The execution time of the tiled parallelized loop program is given by $T_{\text{sos}} = \mathsf{Latency}(p_{P-1}) + \mathsf{TPP}(p) \times \mathsf{TET}(s_i, s_k)$, where we have $\mathsf{TET}(s_i, s_k) = (\alpha s_i s_k N_j + 2(\tau N_j s_i + \beta))$. The number of tiles allocated to a processor is $\mathsf{TPP}(p) = \frac{N_k}{s_k P} \times \frac{N_i + s_k}{s_i}$. This follows from the fact that $\frac{N_k}{s_k P}$ gives the number of passes executed by a processor and $\frac{N_i + s_k}{s_i}$ is the number of tiles executed by a processor in one pass.

The slope $\frac{s_k}{s_i}$ (also known as the *rise* [57]) plays a fundamental role in determining the latency. Processor p_{P-1} can start its first tile only after $(P-1) \times \left(\frac{s_k}{s_i} + 1\right)$ tiles are executed. Hence, we have $\mathsf{Latency}(p_{P-1}) = (P-1) \times \left(\frac{s_k}{s_i} + 1\right) \times \mathsf{TET}(s_i, s_k)$. To ensure that there is no idle time

between the passes, we need to make sure that by the time the first processor finishes all the tiles from its first pass, the last processor should have finished at least one tile. This constraint is given by $P\left(\frac{s_k}{s_i}+1\right) \leq \frac{N_i+s_k}{s_i}$. Putting them all together we get the following constrained optimization problem

$$\begin{aligned} \text{minimize} \quad & T_{sos} = \left[\left(\frac{N_k}{s_k P} \times \frac{N_i+s_k}{s_i}\right)+(P-1)\times\left(\frac{s_k}{s_i}+1\right)\right] \times (\alpha s_i s_k N_j + 2(\tau N_j s_i + \beta)) \\ \text{subject} \quad & \text{to} \quad P\left(\frac{s_k}{s_i}+1\right) \leq \frac{N_i+s_k}{s_i} \end{aligned} \tag{6.2}$$

We can transform this optimization problem (Eqn. 6.2) into a Geometric Program (GP), and can be solved efficiently using the tools discussed in Chapter 5.2.3.

6.4.1 Cache tiling

Each parallel-tile is a semi-oblique block of size $s_k \times s_i \times N_j$. Each block can be further tiled for locality. Since we have tiled the k loop at the outer level we can have both 2D as well as 3D tiles at the inner (cache) level. To make tiling the j loop legal, we have to skew it with respect to the i loop. After this transformation, we can either choose to tile only the i and j loop to obtain 2D cache-tiles, which can exploit spatial locality, or tile all the three loops to obtain 3D cache-tiles and exploit both spatial and temporal locality. We explore only the choice of 2D cache-tiles and leave 3D cache-tiles as future work. Note that the optimal parallel-tile size s_i from the outer level affects the iteration space sizes for the 2D cache-tiling, viz., $s_i \times N_j$. We select the best cache-tile sizes empirically, *i.e.*, by running the cache-tiled coded for several tile sizes, which are within the bounds of s_i and N_j, and selecting the best.

6.5 Experimental Results

We have implemented two versions, one with cache tiling and one without, for both the 1D Strips and Semi Oblique Strip (SOS) strategies. The implementation of the 1D Strips is the optimized latency hiding version. For both the strategies, we selected the best cache-tile sizes by running the cache-tiled code (on a single processor) for a range of tile sizes (within the bounds imposed by parallel-tile sizes). For the 1D Strips, we observed that for the small tile sizes range, there is a steep decrease in the running time as we increase the tile sizes. This trend stops and the running

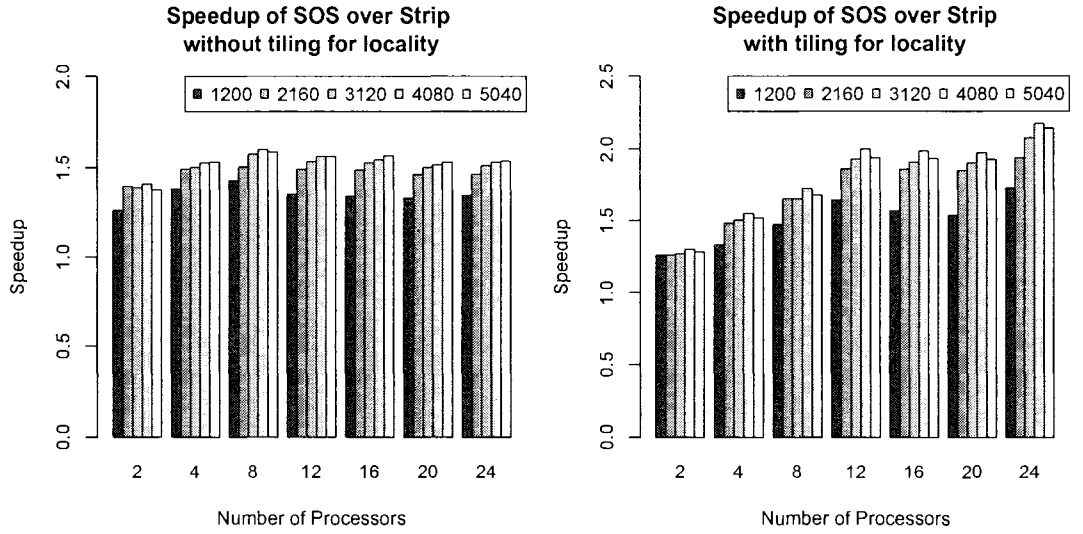


Figure 6.5.
Speedups for SOS over Strip strategy without (left) and with (right) cache tiling. Results for five different grid sizes $N_i = N_j = 1200, 2160, 3120, 4080$, and 5040, each for a set of small time steps $N_k = P$ (the number of processors), are shown.

time becomes relatively constant for larger tile sizes. After experimenting with several strip sizes, we found the cache-tile of size 60 × 140 to be the best and used it for our experiments. For the SOS strategy, the running time had a similar behavior with respect to cache-tile sizes. After experimenting with several grid sizes, we found that the optimal parallel-tile size s_i is always very small, and hence we let the cache-tile size along the i dimension to be the same as s_i. This results in no cache-tiling along i. For the cache-tile size along j we selected a value of 50 which belongs to the flat execution time region.

We used a IBM Cluster 1600 running AIX, at the National Center for Atmospheric Research, Colorado, for our experiments. The IBM Cluster is a Symmetric Multiprocessing (SMP) system. The nodes are made of 1.3-GHz POWER4 processors. The processors in a single node can communicate via shared memory, and the nodes themselves communicate via an SP Switch2 interconnect. We used the IBM `mpcc` compiler for our experiments with standard `-O3` optimization levels. Our parallel implementations are written using the MPI message passing library.

We obtained values of $\alpha = 5.5 \times 10^{-8}, \beta = 4.1 \times 10^{-6}$ and $\tau = 5.3 \times 10^{-9}$ as follows. We ran the loop body of the stencil computation for 1000 iterations and took the average execution time

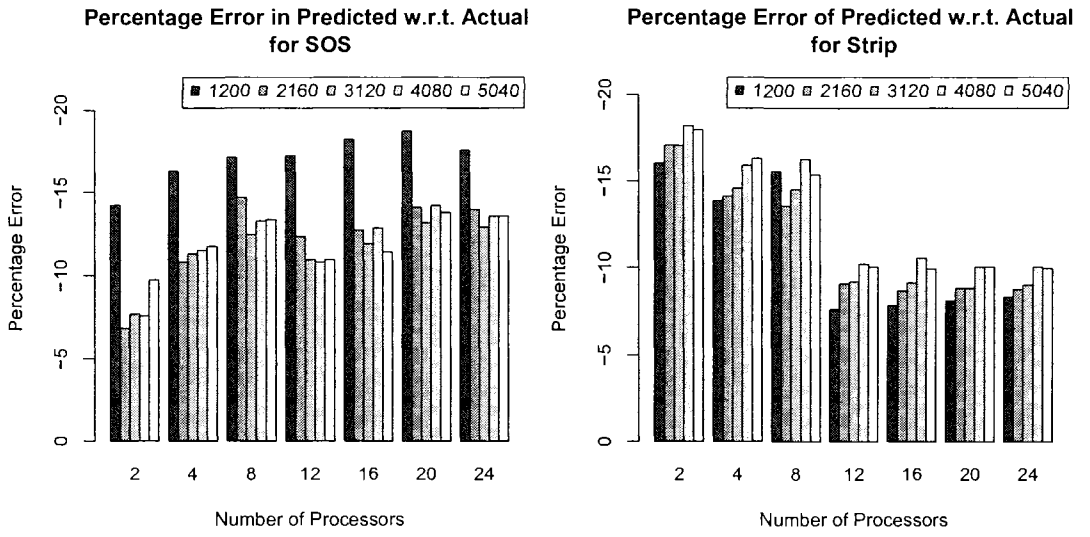


Figure 6.6.
Percentage error in predicted with respected to actual for SOS (Left) and Strip (Right) strategies without cache tiling. Results are reported for five different grid sizes ($N_i = N_j$) each for a set of time steps N_k equal to number of processors P.

as α - the time to compute one iteration. We estimated the cost of communicating one double value (τ), and the MPI communication call start up cost (β), with a ping-pong style MPI program that mimics the communication pattern of our tiled programs.

Stencil computations used in PDE solvers have fast convergence and the number of time steps are usually small, such as 8, 16, or 24. The type of stencil computations used in simulations, such as water models, have large number of time steps. For our experiments we considered these two type of stencils over square grids, *i.e.*, $N_i = N_j$. We found that for small time step stencils the SOS strategy performs better than Strips, and as the number of time steps increases, its performance becomes comparable to that of Strips. This divides the input space into two regions where one of the strategy is clearly preferable over the other.

We present experimental validation and performance improvements for the small number of time steps case. Five different square ($N_i = N_j$) grid sizes viz., 1200, 2160, 3120, 4080, and 5040, with small time steps N_k were used for experiments. For such small time steps in the SOS strategy, the number of processors is set to N_k, since the processors are aligned along the k dimension. Figure 6.5 shows the speedup achieved by the SOS over the Strip strategy (without and with tiling

for locality) for the five different grid sizes with number of processors $P = N_k$. Without cache tiling we obtained speedups of up to 60% (with an average of 40%). Also, we observed higher speedups for larger grids. We observed in single processor experiments that cache tiling helps SOS (30%) more than Strips (5%). These improvements are reflected in their parallel implementations (Figure 6.5(right)). Speedups up to a factor of 2.1 are seen with cache tiling (see $P = 24$ in Figure 6.5(right)). Here, we see a similar trend of higher speedups for larger grid sizes. Clearly, for stencils with small time steps, the new SOS strategy performs much better than the standard Strips strategy. A two fold decrease in running time is significant for such applications.

We validated our analytical models for the two strategies, using more than 100 different combinations of stencil grid sizes and number of processors, and have found them to be reasonably accurate. We present here a subset of them. The percentage error in predicted with respect to the actual for SOS is shown in Figure 6.6(left) and for Strips in Figure 6.6 (right). Our models consistently under predict the execution time. Overall, the predictions are within 20% of the actual execution time, which is good for tiling and design space exploration. Further, for SOS, we conducted experiments to see how close is the predicted to the actual at the optimal tile sizes (s_i and s_k), obtained by solving the constrained optimization problem (*c.f.* Eqn. 6.2). We found that near the optimal running time the predictions are fairly close (within 20%) and at points far from optimum the difference is higher. This is the behavior we desire from such analytical models.

6.6 Related Work

There have been extensive studies [78, 109, 81, 133, 49, 68, 67] on tiling stencil computations for locality. In the space of multi-level tilings that we characterize, these locality improving techniques can be leveraged to improve the implementations at the leaf (uni-processor) level. Data or domain decomposition [52] is a standard scheme used in tiling stencil computations for parallelism. Early work by Wolfe [131] shows that skewing and tiling transformations can be combined to tile for both parallelism and locality. Recently, Wonnacott [132] shows that time skewing can be used to tile for parallelism as well as locality. As discussed in Section 6.1, these authors propose transformations that can enable and make tiling beneficial for parallelism and locality. We characterize the space of possible multi-level tilings and parallelizations with the goal of system-

atically deriving the best implementation for a given stencil.

Andonov *et al.* [10] consider 2D (1D data and 2D iteration space) computations and propose an analytical model similar to ours for estimating the execution time of a tiled program and present analytical closed form solutions for the optimal tile sizes and the number of processors. We consider an 3D iteration space and characterize the possible multi-level tilings and parallelizations. Our analytical BSP style cost models are inspired by theirs. Bordawekar et al. [19] present a technique for optimizing communication for out-of-core distributed stencil computations. They show how a compiler can choose the tiling parameters based on the stencil computation and processor information. Their goal is to minimize the communication, whereas our goal is to find the tiling strategy and tile sizes that minimize the total execution time.

6.7 Discussion

We have characterized the space of legal multi-level tilings and parallelizations for the 2D/3D 9-pt Gauss-Siedel stencil computations. We have shown that a systematic exploration of a part (2 strategies) of this space leads to a new strategy which achieves up to a factor of two improvement over the standard implementation. A two fold decrease in running time is significant for such applications. This illustrates the importance of exploring this space. Further, the exploration helped us to divide the input space into regions in which different designs are better. This shows us the need for runtime data dependent choice of the best implementation. We consider our results as a first step towards a complete exploration of this space.

As a future work, we envision to build a framework that will take a stencil computation as input and will automatically determine the required skewing transformation, and generate analytical models for different tiling and parallelization strategies, and select the best strategy. Majority of the required theory for this is known, and we believe that our GP framework is general enough to integrate all these techniques into a single tool. As an immediate future work, we would like to implement and explore other tiling strategies.

CHAPTER 7

Combined ILP and Register Tiling

EFFICIENT use of multiple pipelined functional units and registers are very important for achieving high performance on modern processors. Instruction Level Parallelism (ILP) and register reuse (through register tiling) are two mechanisms for efficient use of pipelined functional units and registers respectively. Program transformations that expose and exploit ILP and register reuse interact with each other in subtle ways. In this chapter we study the combined problem of optimal ILP and register reuse. We consider the class of uniform dependence, fully permutable, rectangular loop nests. We develop an analytical model of the combined problem and formulate a mathematical optimization problem that chooses the parameters of the ILP-exposing transformation and register tiling so as to minimize the total execution time. We distinguish two cases: when loop permutation can and cannot expose a parallel loop. We show that the combined problem can be reduced to a single IGP for the former case, and to a small set of IGPs for the latter case, both of which can be solved to global optimality. This combined exploration of ILP and register tiling is another example of the broader class of optimizations made possible by the efficient solution methods provided by the GP based approach.

The work presented in this chapter was done in collaboration with Ramakrishna Upadrasta. It was presented in [102].

7.1 Introduction

Efficient use of the multiple pipelined functional units and registers are important to achieve high performance. Instruction level parallelism (ILP) allows a sequence of instructions derived from a sequential program to be parallelized for execution on multiple pipelined functional units in modern processors. Exploiting ILP and register reuse is critical for efficient use of execution resources. Irrespective of whether the target architecture can extract/exploit ILP (like superscalar processors) or not (VLIW processors), compilers *can* transform the program to enhance and expose the parallelism, and schedule the program to exploit the parallelism. No matter what the architecture is, performance is greatly influenced by the quality of the compiler generated code. State-of-the-art compilers perform a variety of program optimizations to expose, enhance and exploit ILP and register reuse.

Loop nests are often the main sources for ILP and register reuse. The traditional approach, shown on the top row of Figure 7.1, uses unroll and jam [6] to expose ILP and scalar replacement to expose register reuse. However, this approach has the disadvantage of increased code size and register pressure. Further, it is hard to quantify the interactions [28] between unroll and jam, scalar replacement and software pipelining, the widely used loop scheduling technique [76, 100, 5].

Loop parallelizing techniques offer many transformations that can expose parallelism. Examples include, loop permutation, loop skewing [6], multi-dimensional scheduling [38], etc. In addition, loop tiling [136] can be used enable register reuse. We propose to use loop permutation and skewing to expose ILP, followed by tiling to enable register reuse. Our approach, shown in the bottom row of Figure 7.1, does not suffer from increased code size. However, enabling register reuse with tiling requires a register allocator for array variables as compared to the use of scalar register allocator in the scalar replacement approach.

Program transformations that expose ILP and those that enable register reuse interact with each other in subtle ways. For example, loop unrolling and loop skewing will expose ILP but might also increase the number of live values and hence the register pressure. On the other hand, register tiling will enable register reuse but might also limit the amount of ILP with the new order of execution of the tiled program. Quantifying and modeling these interactions between

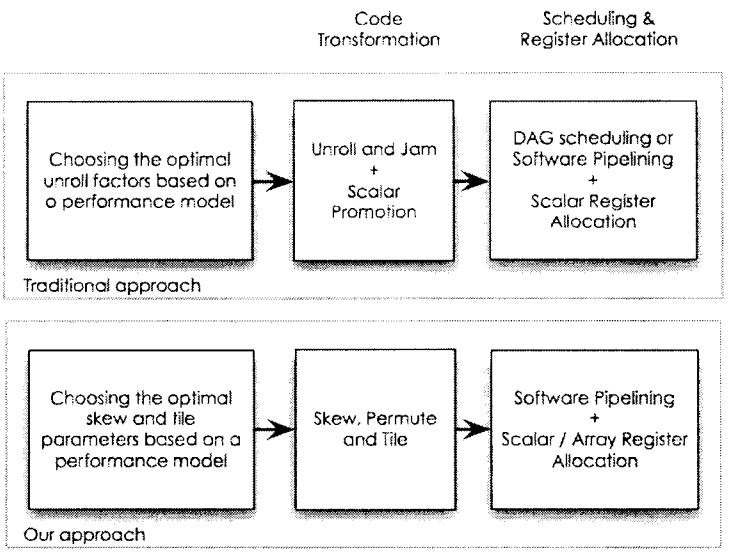


Figure 7.1.
Outline of our approach to ILP and Register Tiling. Top row shows the traditional approach and bottom row shows ours. The choice of code transformation technique influences the parameters to be determined and hence the performance model.

various program transformations is crucial for finding optimal (*w.r.t.* total program execution time) transformations. In this chapter we present a solution to the combined problem of choosing the optimal parameters for the ILP exposing (loop skewing) transformation and register tiling. The key aspects of our solution are outlined below.

- We give an analytical model that quantifies the interaction between the ILP exposing transformation (loop skewing) and register tiling.
- We formulate the optimal ILP and register tiling problem as a mathematical optimization problem. We present a globally optimal solution to this problem by reducing it to a convex optimization problem.
- We distinguish two cases: when loop permutation can and cannot expose a parallel loop. In the former case, we reduce the combined optimization problem to a single integer convex optimization problem. In the later case, when skewing is required to expose ILP, we show that the combined problem can be reduced to a set of integer convex optimization problems.

The solution to our combined problem will produce a loop nest in which the ILP and register reuse are exposed. The scheduling and register allocation phase (cf. Figure 7.1) is an important step

in achieving good performance. This phase is not discussed in this chapter. It can be constructed by adapting well studied techniques like modulo scheduling [100] and register allocation for array variables.

In the next section we present the outline of our solution to the ILP and register tiling problem. In section 7.3, we present the program, tiling, and execution models and describe the basic building blocks of our analytical model. In section 7.4, we formulate the mathematical optimization problem that chooses the optimal skew and tile parameters. In section 7.5, we characterize the condition under which a permutation can expose a parallel loop and present an efficient algorithm to check this condition. In section 7.6, we characterize the space of valid skewing transformations. In section 7.7, we show how the optimal TSS problem can be reduced to a convex program and solved efficiently and in section 7.8, we present the strategy for finding the globally optimal solution to the combined ILP and register tiling problem. In section 7.9, we present a complete example that illustrates our solution method. In section 7.10, we present the related work and in section 7.11, we present a discussion and future work.

7.2 Our approach to ILP and register tiling

Our approach is to use loop skewing as the ILP exposing transformation, and register tiling as the register reuse enabling transformation and software pipelining as the ILP exploiting mechanism. Since we are using register tiling together with loop skewing, we require that after skewing the resulting loop nest admit rectangular tiling.

Software pipeliners look at the innermost loop[1] to find ILP among operations from different iterations of the loop. Hence, if we could transform the loop nest into one in which the inner most loop does not carry any dependences, i.e., all of its iterations can be executed in parallel, then the software pipeliner can find a schedule in which the performance is constrained only by the execution resources as opposed to dependencies. *When sufficient ILP exists and can be exploited, the performance is limited only by the available execution resources – or the execution bandwidth of the machine.* Such a schedule will exploit the maximum possible ILP and have maximum utilization of functional units.

[1]The two exceptions are the works of Rong et al. [110] and Ramanujam [97]. See the related work section for details.

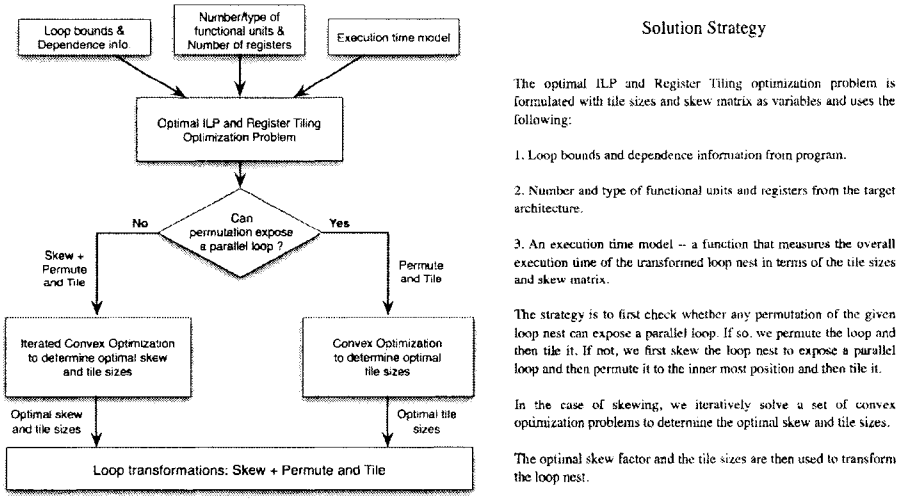


Figure 7.2. Outline of our solution strategy.

Motivated by the above discussion, we seek a transformation that would transform the given fully permutable loop nest into one

- (C1): for which rectangular tiling is valid for any given tile sizes $\vec{t} = (t_1, \ldots, t_n)$. This validity condition reduces to *non-negativity of all the components of all the dependences*, under the reasonable assumption of the tile size being larger than the dependence lengths and the iteration space size being larger than the tile size [135].

- (C2): in which there is at least one loop which does not carry any dependences (i.e., whose iterations are all parallel). We can always permute this loop to the inner most position, as full permutability (of the transformed loop nest) is required by the previous condition (C1).

There are many classes of transformations that can produce a loop nest that would satisfy the above two conditions. Loop skewing is one such class and we have chosen it for following reasons: For uniform dependence loops, we can always find a skewing transformation that will produce a loop that satisfies (C1) and (C2). Second, loop skewing is conceptually simpler and easy to construct, and this allows us to develop an efficient algorithm for finding the optimal skew transformation parameters.

Figure 7.2 shows the outline of the steps involved in our solution methodology. Using the performance model, we formulate an optimization problem whose solution would yield the skew

factor and tile sizes that are optimal with the overall execution time. We check whether permutation can expose any parallel loop. If so, we permute, expose the parallelism, and then tile for registers. In this case, the combined problem reduces to the problem of finding the optimal tile sizes, which can be reduced to a single integer convex optimization problem. When loop permutation cannot expose a parallel loop, loop skewing is required to expose the ILP. In this case, we need to find the optimal skewing and tile sizes. We find these by solving a set of integer convex optimization problems.

7.3 An analytical model

In this section we develop an analytical model that quantifies the interaction between loop skewing and register tiling transformations. A model, similar in spirit, is used in the context of tiling for memory hierarchy [101] and described in Chapter 8.

7.3.1 Program and tiling model

The program class we consider is the class of fully permutable rectangular loop nests with uniform dependence bodies. Note that this class of programs admit rectangular tiling and are also the class for which software pipelining is often applied. We consider an n-dimensional loop nest with constant upper and lower bounds. The loop body contains statements with uniform dependences. Let $\mathscr{L} = [L_1,\ldots,L_n]$ be the given n-dimensional loop nest, where each L_i denotes a loop at depth i. Any n-D vector formed by the loop counters of $\mathscr{L}$ is called an iteration vector. Let $D = [d_1,\ldots,d_m]$ be a matrix whose columns are the (n-D) dependence vectors.

To expose ILP we use skewing and permutation. A skewing (transformation) matrix has the form of an upper triangular matrix with all the diagonal entries equal to 1. The non-diagonal entries are determined by the skewing factors. We denote the skewing matrix that we seek by S. Skewing a loop L_i with respect to a loop L_j, by an appropriate factor f, makes the loop L_i carry all the dependences that were originally carried by loop L_j. A permutation transformation that permutes the i^{th} loop with the j^{th} loop can be represented by an identity matrix (of appropriate size) in which the i^{th} and j^{th} rows are interchanged.

We consider *rectangular* (or *orthogonal*) loop tiling: tiling the loop nest with *hyper-rectangles*

whose boundaries are orthogonal to the canonic axes. We assume that rectangular loop tiling is valid for the given loop nest [136]. Note that the tiled loops are fully permutable. The *tile graph* is the graph where each node represents a tile and each arc represents a dependency between tiles. In our case, each node of the tile graph is a hyper-rectangle of size $t_1 \times t_2 \times \cdots \times t_n$. Note that though our iteration space is rectangular, after skewing, we will have hyper-parallelepiped shaped iteration space, and when we tile this with rectangular tiles, we will have some full rectangular tiles and some partial non-rectangular tiles.

It is well known that [10] if the t_i's are large as compared to the elements of the dependency vectors, then the dependencies between the tiles are *unit vectors* (or binary combinations thereof, which can be neglected for analysis purposes without loss of generality). In general, the feasible value of each t_i is bounded from below by some constant. For the sake of notational simplicity, in this chapter we assume that this is 1.

7.3.2 Architecture and Execution model

We use an atomic tile execution model: tiles are executed sequentially one after the other. However, the parallelism available inside the tile is exploited with software pipelining. We first present the architectural parameters used in the execution model and then introduce the functions that model various aspects of the execution time of the transformed loop nest.

Although we do not provide experimental validation of our execution time model, similar models of execution time have been used by Sarkar [115] (in the IBM XL Fortran compiler) and also by Wolf et al. [129], and they have been thoroughly validated.

Architectural parameters

We seek an abstraction of the architecture (processor and memory features) that is suitable for use in a cost model for tiling loop programs of our program model. Our model uses the following parameters:

- α - *cost of an iteration*: this is the cost of executing an instance of the loop body (in cycles per iteration). In our case, since the innermost loop is completely parallel, a modulo scheduler can always achieve the resource minimum initiation interval (ResMII) [100], and hence α is

equal to ResMII.

- β - the cost (in cycles) for transferring a word from lowest level cache to the registers.
- η - *loop increment and test cost*: this is the cost for incrementing a loop variable and checking its bounds.
- NR - *number of registers available:* depending on the loop body, NR could be either the number of integer or floating point registers.

7.3.3 Fundamental measures

Computation volume. *The computation volume,* $\mathsf{TV}(\vec{t})$*, of a tile is the amount of computation done in a tile.* The computation volume of a tile $\vec{t} = (t_1, \ldots, t_n)$, is the number of integer points in the n-dimensional hyper-rectangle: $\mathsf{TV}(\vec{t}) = \prod_{i=1}^{n} t_i$. The tile volume, $\mathsf{TV}(t)$, represents the volume of full tiles. We approximate the volume of partial tiles with that of the full tiles, and hence use $\mathsf{TV}(\vec{t})$ as the volume for all the tiles.

Load store volume. *The load store volume,* $\mathsf{LS}(\vec{t}, D)$*, of a tile is the total amount of data that is loaded and stored when the tile is executed.* This quantity is also known as the tile foot-print. The dependences and data reuse patterns determine the load store volume. Our program model restricts dependences to be uniform (constant distance). A tile is compute bound if the amount of data accessed (input/output) during the computation of the tile is at least one dimension less than the computation; otherwise the tile is I/O-bound. It is easy to see that with uniform dependences, the load store volume of I/O-bound tiles is proportional to the tile volume $\mathsf{TV}(\vec{t})$. The interesting case, where tiling is really useful, is when the tile is compute bound.

For an n-dimensional compute bound tile, the input and output are $O(x^{n-1})$, where, $x = \max_{i=1}^{n} t_i$, where t_i is the tile size along dimension i. We consider the case in which the input and output are of $O(x^{n-1})$, other cases when the input or output is smaller than $O(x^{n-1})$ can be handled in a similar way using lower dimensional facets. Since our tile graph has dependence vectors that correspond to unit vectors, the $O(x^{n-1})$ input/output of a tile directly corresponds to the $(n-1)$ dimensional facets of the tile, and a constant multiple of every facet contributes to the load store volume of a tile. The constant is determined by the dependence distances. There are n pairs of facets, and in rectangular tiling, each of these is potentially involved in a communication.

The volume of the i^{th} facet, Δ_i, is given by $\prod_{j=1,j\neq i}^{n} t_j$. Now, the load store volume is $\mathsf{LS}(\vec{t},D) = \sum_{i=1}^{n} a_i \Delta_i$, where a_i is a constant that denotes distance along the i^{th} facet that is involved in the communication and is determined by the longest i^{th} dimension component of any dependence vector in the dependence matrix D. Based on the schedule, some facets need not be stored and loaded again. There is at most one such facet, say f, and sharing of f can be captured by excluding it from the load store, i.e., $\mathsf{LS}(\vec{t},D) = \sum_{i=1,i\neq f}^{n} a_i \Delta_i$. We can take care of multiple dependences to the same variable by considering the bounding box of the dependences to each variable and using the diagonal of this bounding box as the columns of D.

Number of tiles. *The number of tiles,* $\mathsf{NT}(\vec{t},\vec{N}) = \frac{N_1 \times \cdots \times N_n}{t_1 \times \cdots \times t_n}$, *counts the total number of tiles after a rectangular tiling with tiles of sizes* $\vec{t} = (t_1,\ldots,t_n)$, *of the rectangular iteration space of size* $\vec{N} = (N_1,\ldots,N_n)$. After skewing, the iteration space may no longer be rectangular and counting the number of tiles in this case is complicated. We use the quantity (iteration space volume)/(tile volume), which is a lower bound on the actual number of tiles, as an approximation. Since we start with a rectangular iteration space and since skewing is a volume preserving unimodular transformation, the quantity (iteration space volume)/(tile volume) is the same as[2] $\mathsf{NT}(\vec{t},\vec{N})$.

Loop overhead. *The loop overhead of a loop is used to account for the cost of loop termination test and loop variable increment.* It is proportional to the number of times the loop body is executed. An n-dimensional rectangular loop nest after one level of tiling will have $2n$ loops. We call the outer n loops *inter-tile loops* and the inner n loops *intra-tile loops.* The i^{th} inter-tile loop is executed precisely $\frac{N_i}{t_i}$ times for *each* instance of the surrounding loop indices. The total overhead of the n inter-tile loops , $\mathsf{LoInterTile}(\vec{t},\vec{N})$, is $\sum_{i=1}^{n} x_i$, where $x_i = \frac{N_1 \times \ldots \times N_i}{t_1 \times \ldots \times t_i}$. The i^{th} intra-tile loop is executed t_i times. The overhead of the set of n intra-tile loops, $\mathsf{LoIntraTile}(\vec{t},\vec{N})$, is $\sum_{i=n+1}^{2n} y_i$, where $y_i = (t_1 \times \ldots \times t_i) \times \mathsf{NT}(\vec{t},\vec{N})$, where $\mathsf{NT}(\vec{t},\vec{N})$ is the total number of tiles and also equal to the number of times the n inter-tile loops surrounding the intra-tile loops will be executed. The total (intra plus inter tile) loop overhead, $\mathsf{LO}(\vec{t},\vec{N}) = \mathsf{LoIntraTile}(\vec{t},\vec{N}) + \mathsf{LoInterTile}(\vec{t},\vec{N})$. Since after skewing the iteration space may not be rectangular, the rectangular tiling might leave some partial and full tiles. Treating partial tiles as full tiles and using the approximation for

[2]Given that we are tiling for registers, the tile sizes are going to be very small and with small tile sizes, this approximation is better.

number tiles, developed above, we can approximate by $\mathsf{LO}(\vec{t},\vec{N})$, the loop overhead of a skewed rectangular loop nest tiled with rectangular tiles.

Given that the (intra and inter-tile) loops are fully permutable and the fact that a loop with larger trip count induces lesser overhead at an outer position, one can choose a permutation that would have the minimum loop overhead. However, a chosen ordering should leave a parallel loop in the inner most position. In this chapter, we do not exploit this flexibility.

When we use skewing to expose ILP, the shape of the iteration space, as well as the dependences change. The iteration space becomes a parallelepiped and the transformed dependences are given by SD, where S and D are the skewing and dependence matrices, respectively.

7.4 Optimization problem formulation

We now formulate an optimization problem (7.1) that clearly captures and quantifies the interaction between the skewing and the register tiling transformations. The objective function is the sum of two terms, the loop overhead and number of tiles times the execution time for a tile, which itself is the maximum of the tile execution time and load store time. The unknowns are the tile sizes ($\vec{t}$) and the skewing matrix (S).

$$\begin{aligned} \text{minimize} \quad & \eta\mathsf{LO}(\vec{t},\vec{N}) + \quad \mathsf{NT}(\vec{t},\vec{N}) \times \max\left(\alpha \times TV(\vec{t}),\ \beta \times \mathsf{LS}(\vec{t},\mathsf{bbox}(SD))\right) \\ \text{s.t.} \quad & \mathsf{LS}(\vec{t},\mathsf{bbox}(SD)) \leq \mathsf{NR} \\ & \vec{N} \geq \vec{t} \geq 1 \\ & SD \geq 0 \\ & \vec{t} \in \mathbb{Z}^n, S \in \mathbb{Z}^{n \times n} \end{aligned} \tag{7.1}$$

where, $\vec{t}$ and S are the variables representing tile sizes and skew matrix, respectively, $\mathsf{NT}(\vec{t},\vec{N})$ is the number of tiles, $\mathsf{TV}(\vec{t})$ is the tile volume, D is the dependence matrix, $\mathsf{LS}(t,\mathsf{bbox}(SD))$ is the load store volume, $\mathsf{LO}(\vec{t},\vec{N})$ is the loop overhead, NR is the number of registers available, α,β and η are respectively the cost of an iteration, load store cost, and loop bounds check cost. All vector inequalities in the constraints are component-wise. The first constraint makes sure that the

register foot print $\mathsf{LS}(\vec{t}\ ,\ \mathsf{bbox}(SD))$ fits in the number of available registers, NR, and the second constraint $\vec{t} \geq 1$ makes sure that the tile sizes are positive and the third constraint $SD \geq 0$ ensures that the skewed loop nest is fully permutable and hence admits a rectangular tiling.

Once we choose a skew transformation S, substituting it in the combined problem gives an optimization problem with $\vec{t}$ as the only variable. Let $\widehat{D} = \mathsf{bbox}(SD)$. Then the resulting optimization problem is shown below (7.2). We call (7.2) the *optimal TSS problem (for a fixed skew).*

$$\begin{aligned} \text{minimize} \quad & \eta\,\mathsf{LO}(\vec{t},\vec{N}) + \ \mathsf{NT}(\vec{t},\vec{N}) \times \max\left(\alpha\mathsf{TV}(\vec{t}),\ \beta\,\mathsf{LS}(\vec{t},\ \widehat{D})\right) \\ \text{s.t.} \quad & \mathsf{LS}(\vec{t}\ ,\ \widehat{D}) \leq \mathsf{NR} \\ & \vec{N} \geq \vec{t} \geq 1 \\ & \vec{t} \in \mathbb{Z}^n \end{aligned} \tag{7.2}$$

Note that, though $\widehat{D}$ is shown as a parameter to the $\mathsf{LS}(\vec{t},\widehat{D})$ function, it is here a given constant vector, and not a variable of the optimization problem.

7.5 Checking whether permutation can expose a parallel loop

We will first introduce some notations (used only in this section) which will make the exposition clear and concise. For any vector x, $x(j)$ represents its j-th component. The *level* of a vector $\mathsf{level}(x)$ is j if $\forall i < j : \ x(i) = 0$ and $x(j) \neq 0$, i.e., $x(j)$ is the first non-zero component of x. A *zero-lead column* is a column vector of the form $(0,0,\ldots,0,c)^T$ for some $c \neq 0$. The j-th *unit vector* e_j is a vector with $e_j(j) = 1$ and $e_j(i) = 0, \forall i \neq j$. A *scaled unit vector*, $\mathsf{suv}(c,j)$ is a vector x of the form $\forall i \neq j : \ x(i) = 0$ and $x(j) = c$ for some non-zero constant c. In other words, $\mathsf{scv}(c,j)$ is an unit vector along j scaled by a non-zero factor c. The dimension of a scaled unit vector is often obvious from the context. An example (of dimension 4) is $\mathsf{suv}(2,3) = (0,0,2,0)$. Note that $\mathsf{level}(\ \mathsf{suv}(c,j)\ ) = j$. $\mathsf{diag}(c_1,c_2,\ldots,c_n)$ constructs a diagonal matrix with $c_1,\ldots,c_n$ as the diagonal entries. A loop is called *parallel* if it does not carry any dependences.

$$D_1 = \begin{array}{c} \begin{matrix} d_1 & d_2 & d_3 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 2 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \qquad D_2 = \begin{array}{c} \begin{matrix} d_1 & d_2 & d_3 & d_4 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 3 \\ 1 & 0 & 2 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 7.3. Example dependence matrices.

7.5.1 Existence of a loop with no carried dependences

We seek to characterize a condition under which there exists no permutation of $\mathscr{L}$ with at least one parallel loop. In other words, in every permutation of $\mathscr{L}$, all the loops carry dependences. We seek a characterization based on the dependences. Let us form a dependence (distance vector) matrix $D = [d_1\ d_2\ \ldots\ d_m]$ whose columns are the m dependences, $d_1, d_2, \ldots, d_m$ present in $\mathscr{L}$'s body. The effect of loop permutation on the dependences is completely captured by permuting the rows of D. In any permutation of $\mathscr{L}$, if there is a dependence d with $\mathsf{level}(d) = j$ then loop l_j, of the permuted loop nest, carries d.

Consider the two dependence matrices D_1 and D_2 given in Figure 7.3. In the matrix D_1, the dependence vectors d_2 and d_3 are scaled unit vectors: $d_2 = \mathsf{suv}(1,3)$ and $d_3 = \mathsf{suv}(2,2)$. Now, in this permutation, the dependences d_1, d_2 and d_3 have levels 1, 3 and 2 respectively and are carried by the loops L_1, L_3 and L_2 respectively. However, we can see that by exchanging rows 1 and 3 of D_1 we can get an innermost loop (row 3 of permuted D_1) with no carried dependences. Now consider matrix D_2 : there exists no permutation of rows of D_2 which can create a parallel loop. What is the structure of the matrix D_2 that induces this property? We seek to characterize this structure in the following discussion leading to Theorem 1.

In any given permutation of the loops, all the n loops will carry dependences if and only if there are (at least) n dependence vectors with levels $1, 2, \ldots, n$. If we have dependence vectors of all levels $(1, 2, \ldots, n)$ in every permutation of the loops in $\mathscr{L}$, then we can say that there is no permutation that will expose a parallel loop.

Theorem 1: Every permutation of the rows of D will contain n columns with levels $1, 2, \ldots, n$ *if and only if* D contains a $n \times n$ sub matrix whose columns can be permuted to form a diagonal matrix, say $\mathsf{diag}(c_1, c_2, \ldots, c_n)$, where $c_1, \ldots, c_n$ are the scale factors of the n scaled unit vectors.

Proof: ($\Longrightarrow$) Assume that every permutation of the rows of D will contain n columns with

levels $1,2,\ldots,n$. Let $x_1,\ldots,x_n$ be these n columns with levels $1,2,\ldots,n$ respectively. Given that we have exactly n vectors each having a different level, they all have to be linearly independent. If we show that these n columns are scaled unit vectors, then we can always permute these columns to form a $n \times n$ diagonal sub matrix of D. To show that $x_1,\ldots x_n$ are scaled unit vectors we will use proof by contradiction. Let us assume that they are (all) not scaled unit vectors. Note that the vector x_n with level n has to be a scaled unit vector. Let the $n-1$ columns each have one more non-zero entry below their first non-zero entry. Without loss of generality we can assume that this entry is the next immediate entry. Then the matrix looks the matrix M given below.

$$M = \begin{pmatrix} x_{1,1} & 0 & \cdots & 0 & 0 \\ x_{2,1} & x_{2,2} & \cdots & 0 & 0 \\ \vdots & x_{2,2} & \cdots & \vdots & \vdots \\ \vdots & \vdots & \ddots & x_{n-1,n-1} & 0 \\ \cdots & & & x_{n,n-1} & x_{n,n} \end{pmatrix} \Longrightarrow M' = \begin{pmatrix} x_{1,1} & 0 & \cdots & 0 & 0 \\ x_{2,1} & x_{2,2} & \cdots & 0 & 0 \\ \vdots & x_{2,2} & \cdots & \vdots & \vdots \\ \vdots & \vdots & \ddots & x_{n,n-1} & x_{n,n} \\ \cdots & & & x_{n,n-1} & 0 \end{pmatrix}$$

Now we can interchange the last two rows of M to get M' in which there is no dependence of level n and hence loop l_n does not carry any dependence. But this is a contradiction to our assumption that every permutation of the rows of D contains n columns with all the levels. Hence the proof. □

Proof: ($\Longleftarrow$) Now we assume that D contains a $n \times n$ sub matrix whose columns can be permuted to form a diagonal matrix say $\mathsf{diag}(c_1,c_2,\ldots,c_n)$. Let C be this $n \times n$ sub matrix of D whose columns can be permuted to form $\mathsf{diag}(c_1,\ldots,c_n)$. We need to show that every permutation of D will contain n column with levels $1,2,\ldots,n$. It is obvious that after any set of row permutations of a diagonal matrix there exists a set of column permutations that will bring it back to diagonal matrix form. Hence, after any set of permutations of C we can column permute C to make it a diagonal matrix. This diagonal matrix form makes it obvious that the n columns have levels $1,\ldots,n$ respectively. Hence the proof. □

Let us look at the two dependence matrices D_1 and D_2 (c.f. Figure 7.3), again, but in the light of Theorem 1. We can see that by exchanging rows 1 and 3 of D_1 we can get an inner most loop (row 3 of permuted D_1) with no carried dependences. There exists no permutations of rows of D_2

Algorithm 3 Algorithm to check whether the input loop nest has any parallel loop.

1. **Input**: Dependence matrix D. **Output**: boolean value indicating whether the input loop nest has any parallel loop or not.
2. Pick all the columns of D which are scaled unit vectors. This can be done in $O(nm)$, where, n is the number of rows of D and m, the number of columns. There can be at most m such columns.
3. As we pick the columns in the previous step we can note their levels. Check whether there are n columns each of which is a scaled unit vector for a distinct j, i.e., $\mathsf{suv}(c_j, j)$ for $j = 1 \ldots n$. This can also be done in time $O(nm)$. If there are such n columns return a **true**; return a **false** otherwise.

which can create a loop with no carried dependences. One can observe that the columns d_4, d_3, and d_2 are scaled unit vectors: $\mathsf{suv}(3,1)$, $\mathsf{suv}(2,2)$ and $\mathsf{suv}(1,3)$ respectively. Further, note that the 3×3 sub matrix formed by the columns d_4, d_3, and d_1 is a diagonal matrix: $\mathsf{diag}(3,2,1)$. One can verify Theorem 1 on these examples.

Theorem 1 gives us an efficient way to check whether there exists at least one loop no carried dependences - we only need to check whether the dependence matrix D contains $n \times n$ sub matrix whose columns can be permuted to form a diagonal matrix $\mathsf{diag}(c_1, \ldots, c_n)$. This can be done in time linear in the size of the dependence matrix D. The outline of the algorithm is given in Algorithm 3.

7.6 Space of valid skewing transformations

When loop permutations alone cannot expose a parallel loop, we need to skew the loop nest. We make two observations regarding the skew matrix S that we seek in the combined optimization problem (7.1). These observations narrow down the search space of S.

- **Only positive skews produce loops that admit rectangular tiling.** We have two constraints: $D \geq 0$ (since our input loop nest admits rectangular tiling) and $SD \geq 0$ (since we require the skewed loop nest to admit rectangular tiling). From Theorem 1, we know that, if the input loop nest does not have any parallel loop, then the dependence matrix D has a $n \times n$ sub matrix whose columns are scaled unit vectors and which can be permuted to form a diagonal matrix, say $M = \mathsf{diag}(c_1, \cdots, c_n)$. Without loss of generality we can as-

sume that that these n columns $c_1, c_2, \ldots, c_n$ have levels $1, 2, \ldots, n$ respectively. At least two of these columns should be made to have the same levels, only then we will have a loop with no carried dependences. Let us view the matrix D as a partitioned as $[M\ N]$, where $M = \mathsf{diag}(c_1, \ldots, c_n)$ is the $n \times n$ diagonal sub matrix and N is the sub matrix that contains rest of the columns of D. We claim that negative skew factors will lead to an invalid transformation by creating negative entries in the sub-matrix M. To see why, let us see what happens when we skew loop L_i with respect to a loop L_j with a negative skew factor $-f$ (cf. Section 7.3.1 for notation). Such a skew would add to the i-th row of M, the j-th row multiplied by $(-f)$. The new i-th row would have $-f \times c_j$ in its j-th entry. This negative entry is not permitted since we require that all the entries of the transformed matrix (SD) be non-negative. Hence, only positive skew factors are valid, since a zero skew factor is just an identity transformation.

- **Skewing any one loop with respect to just one other loop is sufficient and optimal.** We seek to transform the loop nest so that in the transformed loop nest there is one loop that carries no dependences, i.e., parallel. Given that the input loop nest is fully permutable, after skewing, we can permute this parallel loop to the inner most position to get our desired loop nest. To make any one loop, say L_i, parallel, it is sufficient to skew some other loop, say L_j, with respect to L_i. Also, given that (positive) skewing increases the length of the (positive) dependences, skewing with respect to more than one loop will always produce longer (when compared to skewing w.r.t. to just one loop) dependences. And, the longer the dependences, the larger the bounding box and hence, the greater the load store volume, $\mathsf{LS}(\vec{t}, \mathsf{bbox}(SD))$. So, skewing with respect to just one other loop is also optimal. By a similar argument, skewing by a factor larger than 1 to parallelize the loop only increases the load store cost and is sub-optimal.

Based on the two observations made above, we seek to find positive skews of one loop with respect to just one other loop. The number of choices for such skews is $d \times (d-1)$ where, d is the depth of the loop nest we consider. This gives a list of $d(d-1)$ potentially optimal skews. For example, for a loop nest with depth 2 or 3 we will have 2 or 6 choices of skews, respectively.

7.7 Solving the optimal TSS problem

The optimal TSS problem can be cast as a *Geometric Program* (GP) [42]. The concepts of GPs and IGPs introduced in Chapter 5.2 are used here. We show how the problem (7.2) of finding optimal tile sizes can be cast as an Integer Geometric Program(IGP).

7.7.1 Optimal TSS problem is an IGP

The optimal TSS problem (7.2) seeks to choose tile sizes that minimize some criteria and satisfy some constraints. The key insight is that *the variables of this optimization problem, tile sizes, are always positive.* So, polynomial kind of functions of tile sizes naturally become posynomials, when the coefficients are non-negative.

Theorem 3. The optimal TSS problem (7.2) (for finding the optimal tile sizes given the optimal skewing matrix) is an IGP.

Proof. From the definition of the fundamental measures $\mathsf{NT}(\vec{t},\vec{N}), \mathsf{LO}(\vec{t},\vec{N}), \mathsf{TV}(\vec{t})$, and $\mathsf{LS}(\vec{t},\mathsf{bbox}(SD))$ (c.f. Section 8.2.2) one can directly observe that they are all posynomials, since all the coefficients are non-negative, the variables (tile sizes) are always positive, and posynomials are closed under addition. The objective function itself is a sum of posynomials except for the max() function. However, by introducing additional variables the max() function can be completely eliminated. A proof of this can be found in [101]. Now coming to the constraints, $\mathsf{LS}(\vec{t},\widehat{D}) \leq \mathsf{NR}$ can be transformed into a constraint of the required form by dividing the LHS by NR and expressing the resulting inequality as $\mathsf{LS}(\vec{t},\widehat{D})/NR \leq 1$. The rest of the constraints are already in the required form. Hence, the optimal TSS problem (7.2) can be cast as an IGP. □

7.8 Solving the combined ILP and register tiling problem

Recall that, according to our strategy (c.f. Figure 7.2), we need skewing only when the input loop nest does not contain any parallel loop that can be exposed by permutation. Hence, first we check (using Algorithm 3 discussed in Section 7.5) whether the input loop nest has any parallel loop that can be exposed by permutation. If it does, then just permuting the loop to the inner

```
1  for ( i1 = 1; i1 ≤ N1 ; i1++)
2      for ( i2 = 1; i2 ≤ N2; i2++)
3          A[i2] = A[i2-1] + A[i2];
```

Figure 7.4.
Original loop nest. No permutation can expose the parallelism.

most position will achieve our goal. This permutation is always valid, since our input loop nest is fully permutable (since rectangular tiling is valid for it). In this case, we just permute the loop and do not skew (i.e., the skew matrix S becomes the identity matrix). Then the combined problem (7.1) reduces to the optimization problem for finding the optimal tile sizes (for the permuted loop nest), i.e., the optimal TSS problem (c.f. problem (7.2)) with $S = I$ (the identity matrix) and hence $\widehat{D} = \mathsf{bbox}(D)$. This problem can now be directly solved as shown in Section 7.7. Note that when a permutation alone is sufficient, it is globally optimal too, because skewing will always only increase the load store cost and hence the execution time.

When permutation cannot expose a parallel loop, we need skewing to expose ILP. In this case, as shown in Section 7.6, we have $d(d-1)$ choices for the skewing matrix (where d is the depth of the loop nest). We construct $d(d-1)$ optimal TSS problems (with fixed skewing matrices), one for each choice of the skewing matrix. The optimal skew and tile sizes are obtained by solving these $d(d-1)$ optimal TSS problems (7.2) and picking the one that has the smallest objective function value (i.e., the minimum execution time).

7.9 A complete example

Consider the loop nest shown in Figure 7.4. There exists no permutation of the loops that can expose the parallelism to a software pipeliner and one can verify Theorem 1 on the dependence matrix $D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ of this loop nest. However, the loop has lots of parallelism that can be exposed to a software pipeliner by skewing. We have $d(d-1) = 2$ choices for skewing the loops, viz., skewing `i1` w.r.t to `i2` or vice-versa. But, due the symmetric nature of the dependences in D, both the skewing will have the same effect on the bounding box. Let us consider the case, when we skew loop `i2` with respect to the other `i1` and then permute them to make the `i1` loop the inner most. Now all the dependences are carried by the outer loop (`i2`) and the

```
1  for(x1 = 0; x1 ≤ (N2+N1-1) / T1; x1++) {
2    for(x2 = max((-N2+T1*x1)/T2,0); x2 ≤ min((T1*x1+T2)/T2, (N1-1)/T2); x2++) {
3      for(i1 = max(T1*x1+1,T2*x2+2); i1 ≤ min(T2*x2+N2+T2,T1*x1+T1,N1+N2); i1++) {
4        // On the tile boundary use the saved value
5        i2 = max(T2*x2+1,i1-N2);
6        A[i2] = B[i1] + A[i2];
7        for(i2 = max(T2*x2+1,i1-N2) + 1; i2 ≤ min(T2*x2+T2,N1,i1-1) - 1; i2++) {
8          A[i2] = A[i2-1] + A[i2];
9        }
10       // On the tile boundary save the value in B
11       i2 = min3(T2*x2+T2,N1,i1-1);
12       B[i1] = A[i2-1] + A[i2];
13       A[i1] = B[i1];
14     }
15   }
16 }
```

Figure 7.5.
Skewed, permuted, and tiled loop nest. All the iterations of the innermost loop (`i2`) can be executed in parallel.

inner loop (`i1`) is completely parallel. A software pipeliner can exploit all this parallelism to construct a schedule which is constrained only the available execution resources (and not by the dependence constraints). We can further tile this skewed-permuted loop nest to enable register reuse. Figure 7.5 shows the skewed, permuted and tiled loop nest, with tiles sizes `T1` and `T2`.

To determine the optimal tile sizes, we instantiate the combined optimization problem (7.1) with the optimal skew (and permute) matrix $S = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, resulting in an optimal TSS problem, which can solved as discussed in Section 7.7. Now, $\widehat{D} = \mathsf{bbox}(SD) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. Instantiating the optimal TSS problem we get the following problem (7.3):

$$\begin{aligned} \text{minimize} \quad & \frac{N_1 \times N_2}{t_1 \times t_2} \times \max\left(\alpha \times t_1 \times t_2,\ \beta \times (t_1 + t_2)\right) + \eta \left(N_1 \times N_2 + \frac{N_1 \times N_2}{t_2} + \frac{N_1 \times N_2}{t_1 \times t_2} + \frac{N_1}{t_1}\right) \\ \text{s.t.} \quad & t_1 + t_2 \leq \mathsf{NR} \\ & \vec{t} \geq 1 \\ & \vec{t} \in \mathbb{Z} \end{aligned} \tag{7.3}$$

Where, α is the cost per iteration and is equal to the II (initiation interval), β is the cost of moving a data item from the lowest level cache to the register and η is the cost of a loop bound check. NR is the number of (floating point) registers in the architecture.

7.10 Related work

Unroll and jam approach. Sarkar [115] addresses the same problem as ours and uses unroll and jam followed by scalar replacement [25] for exposing ILP and register reuse. He formulates the problem as a discrete optimization problem with unroll factors as the variables and proposes an exhaustive search with heuristics to solve it. Our formulation seeks both skew matrix and tile sizes and is solved to global optimality via convex programming. The class of programs considered by Sarkar, loops with affine dependences, is larger than what is considered by ours, loop nests with uniform dependences. However for uniform dependence loop nests, by setting the skew matrix to identity, viewing the tile sizes as unroll factors, and adding the code size constraint, our method can be directly used to solve the problem addressed by Sarkar. In this sense, for this class of loop nests, the problem of solving for optimal unroll factors is a special case of our problem.

Carr and Kennedy [27] proposes an algorithm to determine the unroll factors that balance the floating-point and memory access operations. This objective function is different from ours, as well as Sarkar's, viz., minimizing the execution time.

Hierarchical tiling. The work of Carter et al. [31], and followed up by Mitchell et al. [85], uses tiling to expose the register reuse as well as ILP. They propose hierarchical tiling as a *hand tuning* technique to better exploit pipelined functional units and registers. Our work is similar to Carter et al.'s work in spirit, however, we have proposed a completely *automatic method* to determine the tile sizes and skew factors.

Code generation for register tiling. Jiminez et al. [64] propose a code generation strategy for non-rectangular loop nests tiled for registers. Their strategy uses index set splitting to strip off the partial boundary tiles and the full tiles are completely unrolled. Hence, they assume that unroll and jam followed by scalar promotion is used for exposing ILP and register reuse. Sarkar [115] also proposes a code generation algorithm which takes the unroll factors as input and produces an unrolled loop nest.

Software pipelining of loop nests. Traditionally software pipeliners have only looked at inner most loop nests. Ramanujam [97] proposed a technique where an integer linear programming formulation is used to find a (software) pipelined schedule that exploits the parallelism available in the whole loop nests. However, he did not consider resource constraints. Rong et al. [112] have

recently proposed a technique called *single dimension software pipelining for multi-dimensional loops*. Their technique computes the initiation interval and (cache) locality of every loop in the given loop nest and picks the best. They do not consider any ILP exposing transformations, like permutation or skewing, and hence, are limited in how ILP can be exploited. Whereas our approach, by the virtue of looking at skewing and permutation, will always be able to expose the available ILP. Rong et al. also propose a method for code generation [111] and recently have addressed the register allocation issue [110]. A similar problem in the context of ILP and caches has been addressed by Wolf et al. [128].

7.11 Discussion and future work

We have formulated the combined problem of choosing an ILP-exposing (skewing) transformation and register tiling. We have proposed an efficient way to check whether permutation can expose any parallel loops. We have distinguished two cases: when loop permutation can expose a parallel loop and when it cannot. For the former case, we have reduced the combined problem to an IGP and for the latter case we have reduced to the combined problem to a small set of IGPs. All these can be solved efficiently using the methods / tools discussed in 5.2.3.

The formulation of the combined problem exposes the fact that the skewing transformation affects the dependences and which in turn affects the overall execution time of transformed loop nest. We see this formulation, and analysis of it, as a first step in understanding the structure of this important complex problem. To the best of our knowledge, this is the first formulation and globally optimal solution of this combined problem.

Two immediate steps are (i) adapting modulo scheduling techniques [100, 5] to schedule the transformed loop nest and (ii) developing array register allocation techniques to map all the array value4s access in a tile to registers. Note that, the modulo scheduler is guaranteed to find the inner most loop nest parallel. Hence, we do not need any dependence analysis to determine the achievable initiation interval and it is constrained only the available resources. Also, from the constraints of the optimal TSS problem, we are guaranteed to have enough registers. Future work would involve extending the input program class to include iteration spaces with parallelepiped shapes. Another direction is to permit non-uniform, say affine, dependences in the loop body.

CHAPTER 8

A Multi-level Data Locality Tiling Model

OPTIMAL tile size selection is a classic problem in compilation of loop kernels. Designing a model of the overall execution time of a tiled loop nest is an important sub problem. On comparison to single-level of tiling, both the problems become harder when tiling is applied at multiple-levels. Due to the complexity of modern architectures and the variations in code generation / optimizations performed by different compilers, modeling the overall execution time of a tiled loop nest is difficult. In this chapter we explore the possibility of deriving an approximate high level execution time model of the tiled loop nest. Our hypothesis is that for the purpose of selecting tile sizes, an high level model is sufficient. This hypothesis has been shown to be true in the context of TSS for parallelism [11, 12, 10, 103]. We seek to explore it in the context of tiling for memory hierarchies. We propose one such high-level model for determining the optimal tile sizes for a fully permutable, perfectly nested, rectangular loop with uniform dependences. We show that the optimal TSS problem, formulated using our model, can be cast as an IGP and solved efficiently. We provide preliminary validation of the model on a small set of loop kernels executed on a simulator. This work was presented in [101].

8.1 Optimal multi-level tiling

Achieving high performance on modern processors requires efficient utilization of the memory hierarchy. Program transformations like tiling [135, 136] try to match the characteristics of a memory hierarchy to the size and order of the data accesses. Multiple levels of tiling [30, 85] are required to match the multiple levels of memory. Determining the optimal size of the tile — one that minimizes the execution time subject to memory characteristics — is a fundamental problem. A model of the overall execution time of a tiled loop nest is an important sub problem. The non-linearity of the functions that describe fundamental properties of a tile, like computation/communication volume, memory footprint, access characteristics, etc., make the problem very hard.

Given a loop nest, the multi-level tiling problem involves the determination of the optimal tile sizes at each level. Usually the optimality is defined based on some cost function which models some aspect of the program execution, for example, number of cache misses, total CPU idle time, etc. Applying tiling at multiple levels with an independent goal or cost function at each level may lead to globally sub-optimal performance [85], since tiling choices from different levels interact with each other. A global metric, like overall execution time, that accounts for interactions from different levels should be used. To use such a global metric, we need a high level analytical model of the overall execution time of the tiled loop nest. Using such an high level model the optimal TSS problem can be formulated as a numerical optimization problem.

We present A high level analytical cost model, similar in spirit to Valiant's BSP model [123] (for parallel programs), for estimating the overall execution time of multi-tiled perfectly nested rectangular loops with uniform dependences. We also discuss how our cost model can be extended to include different processor/memory features and compiler optimizations. We present experimental results that validate our model. We present a formulation of the multi-level optimal TSS problem as an IGP. Our formulation permits an arbitrary number of loops to be tiled and also arbitrary levels of tiling: m-levels of tiling of an n-depth loop.

In the next section we present our high level analytical cost model. In Section 3, we formulate the single-level optimal TSS problem and in Section 4 we extend it to multiple levels of tiling. In Section 5, we describe the Geometric Programming framework and show how the optimal multi-

level tiling problem can be cast as a geometric program. In Section 6, we show the generality of our framework and the extensibility of our cost model. In Section 7, we describe the experimental setup used for the validation of our cost model and then present the results. We describe related work in Section 8 and then conclude in Section 9 with some pointers to future work.

8.2 A high level analytical cost model

In this section we develop a cost model for the total execution time of a tiled loop nest. First we discuss some fundamental measures that can be directly derived from the program. These are the *computation* and *communication volume* of a given tile and the *loop overhead* of a tiled program. The processor architecture dependent parameters, which we call *architectural parameters*, are described next. Then we show how the total execution time can be calculated using the fundamental measures and the architectural parameters. The concepts discussed in this section and the next are in the context of single-level tiling. Extensions of these concepts to multi-level tiling are discussed in Section 8.4.

8.2.1 Program and Tiling Model

We consider an n-dimensional loop nest with constant upper and lower bounds. The loop body contains statements with uniform dependences. Figure (8.1, left) shows our program model. Further, we consider a *compute bound* loop nest - loop nests in which the amount of computation done is at least an order greater than the amount of memory operations. We consider (rectangular or) *orthogonal* loop tiling: tiling the loop nest with *hyper-rectangles whose boundaries are orthogonal to the canonic axes — as are the iteration space boundaries.* We assume that orthogonal loop tiling is valid for the given loop nest [136]. Figure (8.1, right) shows the $2n$-dimensional tiled loop nest. Note that the tiled loops are fully permutable. For example, we can permute the $2n$ loops such that the n^{th} and the $(n+1)^{st}$ loops together correspond to a single loop in the original program. In such a case, we fuse them together.

Let us consider the rectangular iteration space given in Figure (8.1, left) and a tiling of this with rectangular tiles as shown in Figure (8.1, right). The *tile graph* is the graph where each node represents a tile and each arc represents a dependency between tiles. In our case each node of the

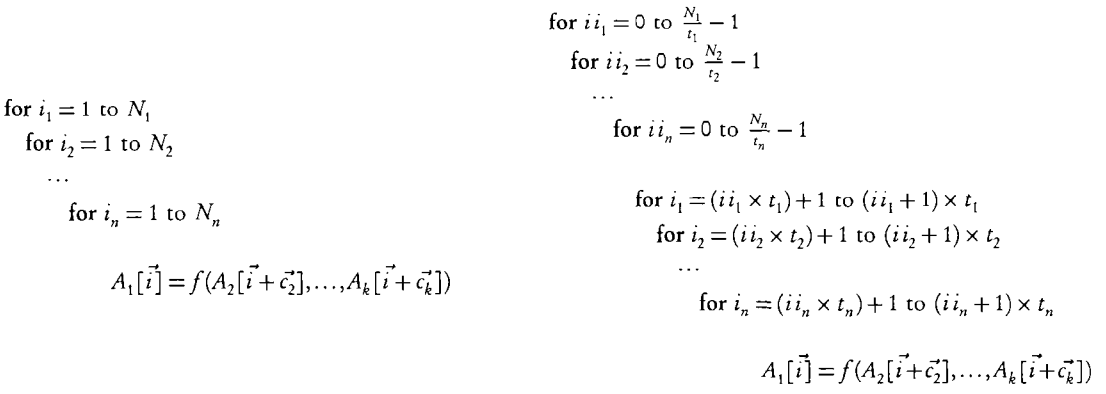


Figure 8.1.
Program model (left): An n-dimensional rectangular loop nest. Tiling model (right): Rectangular tiling of the n-dimensional loop nest

tile graph is an hyper-rectangle of size $t_1 \times t_2 \times \cdots \times t_n$ and the tile graph itself is an hyper-rectangle of size $n_1 \times n_2 \times \ldots \times n_n$, where $n_i = \frac{N_i}{t_i}$. It is well known that [10, 136] if the t_i's are large as compared to the elements of the dependency vectors, then the dependencies between the tiles are *unit vectors* (or binary combinations thereof, which can be neglected for analysis purposes without loss of generality). In general, this implies that the feasible value of each t_i is bounded from below by some constant. For the sake of notational simplicity, in this work we assume that this is 1.

8.2.2 Fundamental measures

The *computation volume*, $\Theta(t)$, of a tile is the amount of computation done in a given tile. The computation volume of a tile t of size $t_1 \times t_2 \times \cdots \times t_n$, is the volume of the n-dimensional hyper-rectangle. $\Theta(t) = \prod_{i=1}^{n} t_i$ where t_i is the tile size along the i-th dimension.

The *communication volume* of a tile, denoted by $\Delta(t)$, is the total amount of data that is input to and output from the tile. For an n-dimensional compute bound tile, the input and output are $O(t^{n-1})$, where, $t = \max_{i=1}^{n} t_i$. We consider the case in which the input and output are of $O(t^{n-1})$, other cases when the input or output is smaller than $O(t^{n-1})$ can be handled in a similar way. Since our tile graph has dependence vectors that correspond to unit vectors, the $O(t^{n-1})$ input/output of a tile directly correspond to the $(n-1)$ dimensional facets of the tile, and a constant multiple of every facet contributes to the communication volume of a tile. The constant

is determined by the dependence distances. There are n pairs of facets, and in the rectangular tiling, each of these n facets is involved in a communication. If Δ_i denotes the volume of the i^{th} facet, then we have $\Delta(t) = \sum_{i=1}^{n} a_i \Delta_i$, where $\Delta_i = \prod_{j=1, j\neq i}^{n} t_j$, and a_i is a constant that denotes distance along the i^{th} facet that is involved in the communication and is determined by the longest i^{th} dimension component of any dependence vector. Let $\mathcal{F}(t)$ denote the memory foot-print of a tile. In our model we have $\mathcal{F}(t) = \Delta(t)$.

The values produced on the iterations at the tile boundaries need to be saved and later used when the corresponding neighboring tile is executed. The storage required for this save and later use can be (and in our model is) accounted in $\Delta(t)$ since it is of the same order. However, the address computation cost related to access of these values stored in array variables could be of the order of any face of the tile, and hence can be expressed as a weighted combination of the faces of a tile. These faces can be of any dimension from $n-1$ to 1. Let this weighted combination of faces be denoted by $\psi(t)$, and we have $\psi(t) = \sum_{i=1}^{F} \rho_i \phi_i$, where F is total number of faces of all dimensions $n-1$ to 1, and ρ_i is a non-negative scalar and ϕ_i is a face. Note that ρ_i can be directly determined from the statements of the tiled loop nest that load-and-store the boundary values.

The *Loop overhead* of a loop is used to account for the cost of loop termination test and loop variable increment. It is of the order of the number of times the loop body is executed. An n-dimensional loop nest after one level of tiling will have $2n$ loops. For ease of notation, we consider the first $1,\ldots,n$ loops (which we call the *inter-tile loops*) and the $n+1,\ldots,2n$ (*intra-tile*) loops separately. The i^{th} inter-tile loop, $i = 1 \ldots n$, is executed precisely $\frac{N_i}{t_i}$ times for *each* instance of the surrounding loop indices. The total overhead of the set of n inter-tile loops, denoted by Λ, is $\Lambda = \sum_{i=1}^{n} x_i$, where $x_i = \frac{N_1 \times \ldots \times N_i}{t_1 \times \ldots \times t_i}$. The i^{th} intra-tile loop, $i = n+1 \ldots 2n$, is executed t_i times. The overhead of the set of n intra-tile loops, denoted by $\lambda(t)$, is $\lambda(t) = \sum_{i=n+1}^{2n} y_i$, where $y_i = t_1 \times \ldots \times t_i$.

8.2.3 Architectural parameters

We seek an abstraction of the architecture (processor and memory features) that is suitable for use in a cost model for tiling loop programs of our program model (Figure 8.1). We have identified the following parameters:

- α - *cost of an iteration*: this is the cost (in cycles) of executing an instance of the loop body.
- β - *bulk transfer rate*: this is the cost (in cycles) for transferring a word of data between memory subsystems. We may have a family of β's one for each level of the memory hierarchy.
- η- *loop increment and test cost*: this is the cost for incrementing a loop variable and checking its bounds.
- $\vec{\rho} = (\rho_1, \ldots, \rho_F)$ - *boundary values load-store cost*: the saving and loading of values at the tile boundaries involves access to array variables that hold these values. Each such access involves address computation and $\vec{\rho}$ represents the F-dimensional vector of this address computation costs, where F is the total number of faces of a tile of dimension $n-1$ to 1. The number of such accesses can be represented by the faces of the tile (see $\psi(t)$ in Section 8.2.2).
- $\mathscr{C}^{(k)}$- *size (capacity) of cache at level k*: The capacity (in bytes) of the cache at the level k.

8.2.4 An analytical cost model

During the actual execution of a tiled loop nest many factors like ILP, reuse, cache hits/misses, etc. affect the running time. However, our model abstracts away from all these low level details and takes an high level view of the execution. In our model, the execution time of a tile is calculated as the sum of the time spent in computation, time spent in communication (data transfer) and the load-store cost of the intermediate values. The execution time of a tiled loop nest is the sum of the execution time for each tile times number of tiles and the loop overhead. Let $\mathscr{T}_{base}(t)$ be the total execution time of the tiled loop nest, then we have

$$\mathscr{T}_{base}(t) = \mathscr{N}(\alpha\Theta(t) + \beta\Delta(t) + \psi(t) + \eta\lambda(t)) + \eta\Lambda(t) \tag{8.1}$$

where, $\mathscr{N}$ is the total number of tiles, α is the cost of executing an iteration, β is the bulk transfer rate, $\psi(t)$, the cost of saving and using boundary iterations, is computed using $\vec{\rho}$, and η is the cost of one loop increment and loop termination test. The model for execution time requires that the

total amount of data accessed by a tile should fit into the cache. This can be stated as $\mathcal{F}(t) \leq \mathcal{C}$, where $\mathcal{F}(t)$ is the memory footprint (c.f., Section 8.2.2) of a tile and $\mathcal{C}$ is the cache capacity.

8.3 Optimal TSS problem formulation

Using the fundamental measures, architectural parameters and cost model discussed in the previous sections, we formulate the single level tiling problem.

8.3.1 Single-level optimal TSS problem formulation

For a single level of tiling, the problem of choosing the tile sizes t_i, $i = 1,\ldots n$ can be formulated as an optimization problem as follows. The objective function $\mathcal{T}(t)$, is the total execution time and we want to minimize it subject to the following constraints: the memory footprint of the tile, $\mathcal{F}(t)$, fits in the cache and the tile sizes (t_i's) are positive. The generic problem is

$$\begin{aligned} \text{minimize} \quad & \mathcal{T}(t) \\ \text{subject to} \quad & \mathcal{F}(t) \leq \mathcal{C} \\ & t_i > 0 \qquad \forall i = 1\ldots n \\ & t_i \in \mathbb{Z} \qquad \forall i = 1\ldots n \end{aligned} \tag{8.2}$$

where, $\mathcal{C}$ is the cache capacity.[1] The choice of the exact function that describes $\mathcal{T}(t)$ depends on the combination of processor features and compiler optimizations we want to model. For instance, one can choose $\mathcal{T}_{base}(t)$ (Eqn. 8.1) or any of the extended cost functions $\mathcal{T}_{oi\_nbc}(t)$, $\mathcal{T}_{hw\_prefetch}(t)$, or $\mathcal{T}_{hi\_opt}(t)$ (discussed in Section 8.6.1) and use in the place of $\mathcal{T}(t)$ in (8.2) to obtain a concrete problem.

8.4 Multi-level optimal TSS problem formulation

Let us consider m levels of tiling of an n-dimensional loop nest. In the tiled program there are $(m+1) \times n$ loops. Let $\mathcal{T}^{(j)}$ denote the execution time of the n-dimensional loop nest tiled j

[1] We can easily include a constraint like $t_i > d_i$ where d_i is the maximum of the projections of the dependence vectors along the i^{th} dimension. However for sake of notational simplicity we stick to $t_i \geq 1$.

levels, i.e., the execution time of the innermost $(j+1) \times n$ loops. We can define $\mathcal{T}^{(j)}$ recursively as follows:

$$\begin{aligned} \text{for } j = 2, \ldots m: \ \mathcal{T}^{(j)} &= \mathcal{N}^{(j)}(\mathcal{T}^{(j-1)} + \beta^{(j)}\Delta(t^{(j)})) + \eta\Lambda^{(j)} \\ \mathcal{T}^{(1)} &= \mathcal{N}^{(1)}(\alpha\Theta(t^{(1)}) + \beta^{(1)}\Delta(t^{(1)}) + \psi(t^{(1)}) + \eta\lambda(t^{(1)})) + \eta\Lambda^{(1)} \end{aligned}$$

where,

- for $j = 1, \ldots, m$: $\mathcal{N}^{(j)} = \frac{t_1^{(j+1)} \times \cdots \times t_n^{(j+1)}}{t_1^{(j)} \times \cdots \times t_n^{(j)}}$, with $t_i^{(m+1)} = N_i$, for $i = 1, \ldots, n$.
- for $j = 1, \ldots, m$: $\Delta(t^{(j)}) = \sum_{i=1}^{n} a_i^{(j)} \Delta_i(t^{(j)})$, with $\Delta_i(t^{(j)}) = \prod_{k=1, k \neq i}^{n} t_k^{(j)}$, for $i = 1, \ldots, n$.
- for $j = 1, \ldots, m$: $\Lambda^{(j)} = \sum_{i=1}^{n} \prod_{k=1}^{i} \frac{t_k^{(j+1)}}{t_k^{(j)}}$, with $t_l^{(m+1)} = N_l$, for $l = 1, \ldots, n$.
- for $j = 1, \ldots, m$: $\beta^{(j)}$ is the bulk transfer rate for moving a byte of data from a memory at level $j+1$ into a memory at level j.
- for $j = 1, \ldots, m$: $\eta^{(j)} = \eta$, since the loop variable increment and termination check cost is the same for every loop at every level.
- The quantities related to the cost of execution of actual statements are relevant only at the inner most level of tiling and hence contribute to the execution time of the inner most level $\mathcal{T}^{(1)}$. These quantities are: $\alpha\Theta(t^{(1)})$ - the computation cost at level 1 and $\psi(t^{(1)})$ - the load-store cost at level 1. Also note that the (inner-most) intra-tile loop overhead $\lambda(t^{(1)})$ contributes only to the execution time of the inner-most level. Hence, the quantities $\alpha, \Theta(t^{(1)}), \psi(t^{(1)})$, and $\lambda(t^{(1)})$ are confined to the inner most level and are defined as in the single level tiling case (c.f. Section 8.2.2).

We formulate the multi-level tiling problem using a generic $\mathcal{T}(t^{(m)})$, which is a function of all the fundamental measures and architectural parameters at the level m. Based on the combination of processor features and compiler optimizations chosen, we can substitute the corresponding $\mathcal{T}(t^{(m)})$ to get a concrete optimization problem. Let us consider m levels of tiling of an n-depth

loop nest, we have the following optimization problem in $m \times n$ variables:

$$\begin{aligned} \text{min.} \quad & \mathscr{T}(t^{(m)}) \\ \text{subject to} \quad & \mathscr{F}(t^{(j)}) \leq \mathscr{C}^{(j)} \quad \text{for } j = 1 \ldots m \\ & 0 < t_i^{(j)} \leq t_i^{(j+1)} \quad \text{for } i = 1 \ldots n, j = 1 \ldots m \\ & t_i^{(j)} \in \mathbb{Z} \quad \text{for } i = 1 \ldots n, j = 1 \ldots m \end{aligned} \tag{8.3}$$

Consider the problem of tiling for $m > 1$, levels of tiling. The optimization problem is not separable, i.e., it cannot be solved one level at a time, since the tile variables $t_i^{(j+1)}$ at a level $(j+1)$ influences $\mathscr{N}^{(j)}, \Lambda^{(j)}$, and upper bounds of $t_i^{(j)}$. Further, $\mathscr{T}^{(j-1)}$ becomes the computation time of a tile at the next level. Hence, a globally optimal solution would require solving the whole optimization problem.

8.4.1 Illustration: Two-level tiling of a doubly nested loop

To illustrate the multi-level tiling formulation, we present the concrete optimization problem for the base cost model ($\mathscr{T}_{base}(t)$) used for a loop nest of depth two ($n = 2$) tiled twice ($m = 2$). We start from the inner most level of tiling ($j = 1$) and move to the outer level ($j = 2$). For the inner most level we have

$$\mathscr{T}^{(1)} = \mathscr{N}^{(1)}(\Theta(t^{(1)})\alpha^{(1)} + \Delta(t^{(1)})\beta^{(1)} + \psi(t^{(1)}) + \eta\lambda(t^{(1)})) + \Lambda^{(1)}\eta$$

where, $\Theta(t^{(1)}) = t_1^{(1)} \times t_2^{(1)}$, $\Delta(t^{(1)}) = a_1^{(1)} t_2^{(1)} + a_2^{(1)} t_1^{(1)}$, $\Lambda^{(1)} = \frac{t_1^{(2)}}{t_1^{(1)}} + \frac{t_1^{(2)} \times t_2^{(2)}}{t_1^{(1)} \times t_2^{(1)}}$, $\psi(t^{(1)}) = t_1^{(1)}\rho_1 + t_2^{(1)}\rho_2$, $\lambda(t^{(1)}) = t^{(1)} + t^{(1)} \times t^{(2)}$, $\mathscr{N}^{(1)} = \frac{t_1^{(2)} \times t_2^{(2)}}{t_1^{(1)} \times t_2^{(1)}}$, $\beta^{(1)}$ is the bulk transfer rate between memory levels 1 and 2, and ρ_1, ρ_2 are the cost of the load-store statements executed $t_1^{(1)}$ and $t_2^{(1)}$ times respectively. The memory footprint at this level is $\mathscr{F}(t^{(1)}) = \Delta(t^{(1)})$. For the next, outer level, $j = 2$, we have,

$$\mathscr{T}^{(2)} = \mathscr{N}^{(2)}(\mathscr{T}^{(1)} + \beta^{(2)}\Delta(t^{(2)})) + \eta\Lambda^{(2)}$$

where, $\Delta(t^{(2)}) = a_1^{(2)} t_2^{(2)} + a_2^{(2)} t_1^{(2)}$, $\Lambda^{(2)} = \frac{N_1}{t_1^{(2)}} + \frac{N_1 \times N_2}{t_1^{(2)} \times t_2^{(2)}}$, $\mathscr{N}^{(2)} = \frac{N_1 \times N_2}{t_1^{(1)} \times t_2^{(1)}}$, and $\beta^{(2)}$ is the bulk transfer rate between memory levels 2 and 3. Note that, $t_1^{(3)} = N_1$ and $t_2^{(3)} = N_2$ since we are tiling a doubly

nested loop of size $N_1 \times N_2$. The memory footprint at this level is $\mathcal{F}(t^{(2)}) = \Delta(t^{(2)})$.

Now, the optimization problem that selects the optimal tile sizes $t_1^{(1)}, t_2^{(1)}, t_1^{(2)}$, and $t_2^{(2)}$ is

$$\begin{aligned}
\text{min.} \quad & \mathcal{T}(t^{(2)}) \\
\text{subject to} \quad & \mathcal{F}(t^{(2)}) \leq \mathcal{C}^{(2)} \\
& \mathcal{F}(t^{(1)}) \leq \mathcal{C}^{(1)} \\
& 0 < t_1^{(2)} \leq N_1 \\
& 0 < t_2^{(2)} \leq N_2 \\
& 0 < t_1^{(1)} \leq t_1^{(2)} \\
& 0 < t_2^{(1)} \leq t_2^{(2)} \\
& t_1^{(1)}, t_2^{(1)}, t_1^{(2)}, t_2^{(2)} \in \mathbb{Z}
\end{aligned}$$

8.5 Optimal TSS Problem is an IGP

The optimal TSS problem can be cast as a *Geometric Program* (GP) [42]. The concepts of GPs and IGPs introduced in Chapter 5.2 are used here. We show how the problem of finding optimal tile sizes can be cast as an Integer Geometric Program(IGP).

The optimal TSS problem seeks to choose tile sizes that minimize some criteria and satisfy some constraints. The key insight is that *the variables of this optimization problem, tile sizes, are always positive.* So, polynomial kind of functions of tile sizes naturally become posynomials, when the coefficients are non-negative. We first show that the single-level optimal TSS problem is an IGP, and use the properties of posynomials and GPs to show that the multi-level tiling problem is also an IGP.

Lemma 8.5.1. *The fundamental measures* $\Theta(t), \Delta(t), \Delta'(t)$ *and* $\Lambda(t)$ *are posynomials*

Proof. From the definition of these measures (c.f. Section 8.2.2) one can directly observe that they are all posynomials, since all the coefficients are non-negative, the variables (tile sizes) are always positive, and posynomials are closed under addition. □

Theorem 8.5.2. *The single level tiling problem (8.2) is an* IGP *for all posynomial objective functions.*

Proof. We need to show that all the constraints in (8.2) can be cast as posynomial inequality constraints or monomial equality constraints as in (5.1). The positivity and integrality constraints on t_i naturally maps into the constraints of GP. The capacity constraint, $\mathscr{F}(t) \leq \mathscr{C}$ can also be easily cast as a posynomial inequality constraint by the following rewrite $\mathscr{F}(t) \leq \mathscr{C} \iff \mathscr{C}^{-1}(\Delta(t)) \leq 1$, which is a posynomial, since $\Delta(t)$ is a posynomial (from Lemma 8.5.1) and $\mathscr{C}$ is a constant. Hence, whenever the objective function is (also) a posynomial, the whole problem is an IGP. □

From Lemma 8.5.1 and Theorem 8.5.2, we can observe the cost function $\mathscr{T}_{base}(t)$ introduced in Section 8.2.4 (Eqn. (8.1)), is a posynomial and using it as $\mathscr{T}(t)$ in the single-level optimal TSS formulation (c.f. Eqn. (8.2)) will yield an IGP.

Theorem 8.5.3. *The multi-level optimal TSS problem (8.3) is an* IGP *for all posynomial objective functions.*

Proof. The proof follows directly from the proof for the single-level case (Theorem 8.5.2) since, we have just added some more constraints that are all similar in form to the ones in (8.2). Hence, whenever the objective function is a posynomial we have an IGP, and we can solve for the tile sizes directly. □

From Lemma 8.5.1, we can observe that $\mathscr{T}_{base}(t)$ (c.f. Section 8.2.4, Equation (8.1)) is a posynomial. Repeated composition of $\mathscr{T}_{base}(t)$ with other posynomials through addition at multiple levels would yield a posynomial since posynomials are closed under addition. Hence, from Theorem 8.5.3, we can observe that using $\mathscr{T}_{base}(t)$ repeatedly at m levels to construct a $\mathscr{T}^{(m)}$ will yield a posynomial $\mathscr{T}^{(m)}$ which can be used in the multi-level optimal TSS formulation (c.f. Eqn. (8.3)) to get an IGP.

8.6 Generality and extensions

At a first look our model might seem simple, however it is general and can be easily extended. In this section, first we show how our analytical cost model can be extended to include various architectural features and compiler optimizations. Then we show the generality of the GP based framework in accommodating other cost models and functions.

PROCESSOR FEATURE	IMPACT ON MISS RATE / MISS PENALTY
Non-blocking cache and out-of-order issue	Hides L1 miss latency and reduces L1 miss rate
Critical word first	Reduces miss penalty
Priority to read misses and merging write-buffers	Reduces miss penalty
Hardware prefetching	Reduces miss rate or miss penalty
Larger cache size or line size	Reduces capacity misses
Higher associativity	Reduces conflict / replacement misses
Victim caches	Reduces conflict misses
COMPILER OPTIMIZATION	
Padding for alignment	Reduces conflict misses
Compiler controlled prefetching	Reduces (or removes) miss penalty
Projective memory allocation	Reduces memory requirement and (thereby the) number of misses
Data remapping	Improves locality and reduces number of misses

Table 8.1.
Widely used processor features and compiler optimizations that influence memory access cost and execution time

8.6.1 Extensibility of the cost model

The cost model can be easily refined to include more details about processor features and compiler optimizations. See Table 8.1 for a list of processor features and compiler optimization influence memory access cost and execution times. Such refinements would either affect the miss rate or miss penalty and can be accommodated by appropriately scaling the bulk transfer rate β or by changing the number of misses $\Delta(t)$. For example, consider the following three scenarios:

- *Out-of-order issue and non-blocking cache:* Consider an out-of-order issue processor with a non-blocking cache. The out-of-order issue together with a non-blocking cache can hide the miss penalty for accesses that are a miss at L1 but a hit at L2, given sufficient ILP in the code. This effect can be modeled by reducing the miss penalty for such misses. We can capture this by determining the the number of L1 misses for which the miss penalty is reduced and then scaling down β by an appropriate factor, say f_{mr}.

$$\mathcal{T}_{oi\_nbc}(t) = \alpha\Theta(t) + (f_{mr}\beta)\Delta(t) + \psi(t) + \eta\lambda(t) + \eta\Lambda(t). \tag{8.4}$$

  To account for set-associativity of the cache, we may have to scale down the cache capacity to an effective cache capacity, as discussed later in this section.

- *Hardware prefetching:* Hardware prefetching can decrease the miss penalty substantially (and not completely remove, since the hit time to a prefetch stream buffer is slightly higher than the cache hit time) for accesses that have spatial locality, by prefetching subsequent blocks and storing them in the stream buffer. This effect can be modeled by scaling down β by a factor f_{mp}.

$$\mathcal{T}_{hw\_prefetch}(t) = \alpha\Theta(t) + (f_{mp}\beta)\Delta(t) + \psi(t) + \eta\lambda(t) + \eta\Lambda(t) \tag{8.5}$$

- *Highly optimized execution:* Consider now an advanced processor with all the features listed in Table 8.1 together with a compiler that can perform all the optimizations listed in Table 8.1. The net effect of would be an almost complete overlap of computation and the data movement. In such a case the execution time is the maximum of the time taken for computation and the memory access time.

$$\mathcal{T}_{high\_opt}(t) = \max(\alpha\Theta(t) + \psi(t) + \eta\lambda(t) + \eta\Lambda(t)\,,\ \beta\Delta(t) + \psi(t) + \eta\lambda(t) + \eta\Lambda(t)) \tag{8.6}$$

  Such a scenario is very common with respect to the hardware features. However, some compiler optimizations like memory reduction, padding and data remapping are not available in all compilers, though the techniques are well understood in the research community.

Other combinations of processor features and compiler optimizations can also be easily included. For example, low-associativity of caches and the use of padding and data remapping [121, 105, 122] can be included by appropriately scaling down the cache capacity $\mathscr{C}$ to an *effective cache size.* This is a well studied [105, 121, 122, 114] and widely used technique. An algorithm of how to compute the effective cache size can be found in [114]. Such a technique is also used by other researchers in the similar context of single-level and multi-level tiling [77, 128, 30, 85].

Observe that the cost functions $\mathcal{T}_{oi\_nbc}(t)$ and $\mathcal{T}_{hw\_prefetch}(t)$ are both posynomials by construction. The function $\mathcal{T}_{hi\_opt}(t)$ is directly not a posynomial. However, it can be transformed into a posynomial qualified with posynomial inequality constraints using the max elimination technique shown in [21]. In a multi-level tiling, if $\mathcal{T}_{hi\_opt}(t)$ is used repeatedly at each levels, then we will have an function with nested max()'s. For this case, we can start from the inner

most max() and repeatedly apply the max elimination technique to obtain a single IGP. Hence, all the three functions can be used in the context of single or multi-level tiling to obtain an IGP and hence can be solved efficiently. This shows how one can combine the extensibility of our cost model with the generality of the GP based framework to include advanced processor features and compiler optimizations.

8.7 Experimental results

For our experiments we used the `sim-outorder` simulator from SimpleScalar tool set [24]. It is a cycle accurate processor simulator with two levels of cache and a TLB. We configured it for an in-order issue processor and we set the caches to be fully associative with sizes 4k(L1) and 64k(L2). As stated in Section 8.6.1, it is well known that the results obtained for fully associative caches can be adapted to set-associative caches by using standard techniques like padding and using an effective cache size, for example see [106, 105, 77, 47, 114]. We experimented with single and multiple levels of tiling of doubly and triply nested loops. We considered five different programs (dep1, dep1-LF, dep2, stat2, and var2) and four different tiling scenarios: one-level tiled doubly nested loop ($m = 1, n = 2$), two-level tiled doubly nested loop ($m = 2, n = 2$), one-level tiled triply nested loop ($m = 1, n = 3$), and two-level tiled triply nested loop ($m = 2, n = 3$). The five programs[2] had the following features:

- dep1: contains a loop body with floating point addition and a dependence of depth one.
- dep1-LF: this is program dep1 with the tile-loop and inter-tile loop of the inner most time dimension fused.
- dep2: contains a loop body with floating point addition and a dependence of depth two. Note that a dependence of depth two requires saving and loading two facets of intermediate values along the dependence direction.
- stat2: contains a loop body with two independent statements that do floating point additions. This loop body has instruction level parallelism and also would have exploited the pipelined floating point addition unit.

[2]the tiled codes are available at: `http://www.cs.colostate.edu/~ln/tiled-loops/`

Program	$m=1, n=2$		$m=2, n=2$		$m=1, n=3$		$m=2, n=3$	
	μ	σ	μ	σ	μ	σ	μ	σ
dep1	-1.71	3.35	15.79	8.71	-11.12	13.60	11.09	7.02
dep1-LF	-1.76	3.33	15.78	8.43	-11.14	13.62	11.04	6.96
dep2	-0.36	3.96	21.72	11.58	-11.20	15.07	16.63	9.53
stat2	5.37	4.14	25.78	8.64	-6.11	11.58	8.88	6.13
var2	17.99	2.36	22.48	4.09	8.95	10.09	10.74	4.83

Table 8.2.
Experimental Results. Mean and standard deviation of the percent error between predicted and simulated execution times. m is the number of levels of tiling and n is the loop nest depth.

- var2: contains a loop body with two dependent statements (the result of the first used as an operand in the second) that do floating point operations. This loop body does not have ILP but may exploit the pipelined floating point unit.

Each of the above five programs together with the four tiling scenarios resulted in twenty different programs. We ran each of these programs on more than ten different tile and program parameter combinations, resulting in more than two hundred different runs.

We measured the percent error in prediction of the execution time by our model, i.e., the percent error between the simulated execution time and the estimated execution time. The mean and standard deviation of the percent error results are presented in Table 8.2. A negative mean indicates an underestimation and a positive one an overestimation of the execution time by the model. One can note from the results that our model predicts the execution time with an error (approximately) between 5 to 30 percent. For the purposes of tiling, a high level model with such an error range seems reasonable.

8.8 Related work

Tiling for memory hierarchy is a well studied problem and so is the problem of modeling the cache behavior of a loop nest. We classify the related work into three categories: models of cache behavior of loop nests, single-level optimal TSS and multi-level optimal TSS.

Models of cache behavior of loop nests. There are several analytical models that measure the number of cache misses for a given class of loop nests. These models can be classified into precise models that use sophisticated (computationally costly) methods and approximate models that

provide a closed form with simple analysis. In the precise category, we have the Cache Miss Equations [50], and the refinement by Chatterjee et al. [34], that use Ehrhart Polynomials [35] and Presburger formulae to describe the number of cache misses. Harper et al. [54] propose an analytical model of set-associative caches and Cascaval and Padua [32] give a compile time technique to estimate cache misses using stack distances. In the approximate category, Ferrante et al. [47] present techniques to estimate the number of distinct cache lines touched by a given loop nest. Sarkar [114] presents an refinement of this model. Although the precise models can be used for selecting the optimal tile sizes, only Abella et al. [2] has proposed a *near optimal* loop tiling using Cache Miss Equations and genetic algorithms. Sarkar and Megiddo [116] have proposed an algorithm that uses an approximate model [47] and finds the optimal tile sizes for loops of depth up to three. No previous work has used any of these models to find optimal tile sizes in the context of multi-level tiling of loop-nests of arbitrary depth. Our execution model, though an approximate one, can be used for multiple-levels of tiling as shown in this chapter.

Single-level optimal TSS. Several algorithms [77, 36, 33, 60] have been proposed for single-level tile size selection (see Hsu and Kremer [60] for good a comparison). The majority of them use a local cost function such as the number of capacity misses or conflict misses, not a global metric like ours, viz., overall execution time. Mitchell et al. [85] illustrate how such local cost functions may not lead to globally optimal performance.

Multi-level optimal TSS. Mitchell et al. [85] was the first one to quantify the multi-level interactions of tiling. They clearly point out the importance of using a global metric like execution time rather than local metrics like number of misses, etc. Further, they also show through examples, the interactions between different levels of tiling and hence the need for a framework in which the tile sizes at all the levels are chosen simultaneously with respect to a global cost function. In this chapter we have proposed one such framework for a restricted class of programs. Other results that show the application and importance of multi-level tiling include [30, 89, 65]. Empirical tools like PHiPHAC [16] and ATLAS [126] use a profile-driven approach to choose the optimal tile sizes. These tools are limited to the set of programs for which they are designed and are time consuming.

8.9 Discussion and future work

We have proposed an high-level TSS model with three properties: (i) a global metric such as execution time; (ii) extensible to arbitrary levels of tiling; (iii) can be used for efficient solution of optimal tile sizes via IGPs. As part of our ongoing work, we plan to validate our cost model with more programs and different cache and processor configurations. As a next step, we will consider two multi-level tiling scenarios: *(a)* an outer level of tiling for parallelism and inner level of tilings for memory hierarchy; *(b)* outer levels of tilings for memory hierarchy and inner level of tiling for instruction level parallelism (ILP). Extending the program model to include non-rectangular loop nests and non-uniform (say, affine) dependences would be the next major step.

CHAPTER 9

Conclusions and Future Work

MULTI-LEVEL tiling is a widely used loop transformation. Lack of tool support has limited its use to optimization experts. The tile size selection framework and tiled loop generation methods proposed in this thesis provide scalable and efficient tools for multi-level tiling. We believe that our tools will enable a wide spread use of multi-level tiling. The scalability and efficiency of our tools make them suitable for inclusion in production compilers, iterative optimizers and auto-tuners. Further, our multi-level tiling tools are a necessity to realize the performance potential offered by systems such as CELL BE [66] and nVidia's CUDA enabled GPUs [37].

Our TSS framework derives its scalability and efficiency properties from the underlying convex optimization methods. By doing so, it brings powerful numerical optimization techniques to the world of compiler optimizations. For example, in numerical optimization, sensitivity analysis is a widely used technique for understanding the sensitivity of the optimal solution with respect to parameters of the optimization problem [21]. Such a technique can be directly applied to optimal TSS problems to study the sensitivity of tile sizes to the parameters (such as cache sizes, network latency, etc.) involved in TSS optimization problem [40, 75]. Questions such as "how much performance can be gained by increasing the cache size? " can be answered with a sensitivity analysis.

Our tiled loop generation algorithms are based on the concepts of two polyhedral sets, viz.,

inset and outset. We have shown that by appropriate use of these two sets we can derive a variety of parameterized, fixed, or mixed tiled loop generation algorithms. By providing an efficient method for computing these sets, we have provided a unified basis for the design and implementation of tiled loop generation algorithms. This efficiency also led to the development of a scalable method which can provide m levels of tiling at the price of just one level.

Specific directions of future work were outlined earlier at the end of each chapters. A few general directions are outlined below.

9.1 Posynomial based modeling

We believe that the use of posynomials for performance modeling is applicable to more than just tile size selection. This belief is based on the fact that almost all the parameters selected or tuned by compilers are positive. The following are other scenarios where we think posynomial based performance modeling will be useful.

- **Prefetching:** Models to estimate the optimal prefetching distance given the overheads and performance benefits.
- **Transactions:** Models to estimate the optimal length of transactions for a given cost of conflict detection and roll back.

Another promising approach is the use of posynomials to learn performance models that can be used for TSS. The promise of this approach is based on the observation that posynomials are widely used in designing TSS models. The idea is to use posynomials as basis functions and fit a posynomial model to the execution time data of a tiled loop nest. Our parameterized tiled loop generation methods can be exploited here to generate parameterized tiled codes that can be executed for a set of tile sizes to collect the execution time data.

9.2 Tile shape and size selection

Recent work by Bondhugula et al. [17, 18] has provided a linear programming based formulation for tile shape selection. They do not address the issue of tile size selection. Our posynomial based

tile size selection framework complements Bondhugula et al.'s work. It would be interesting to combine both the works to formulate an optimization problem that selects both tile shape and sizes. An observation that would be helpful in this combination is that both linear programs and geometric programs are subsets of the broader class of convex programs.

Bibliography

[1] An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw. 28*, 2 (2002), 135–151.

[2] ABELLA, J., GONZALEZ, A., LLOSA, J., AND VERA, X. Near-optimal loop tiling by mean of cache miss equations and genetic algorithms. In *Proceedings of International Conference on Parallel Processing Workshops* (2002).

[3] ABU-SUFAH, W., KUCK, D., AND LAWRIE., D. On the performance enhancememt of paging systems through program analysis and transformations. *IEEE Transactions on Computers 30*, 5 (May 1981), 341–356.

[4] AGARWAL, A., KRANZ, D. A., AND NATARAJAN, V. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 6*, 9 (1995), 943–962.

[5] ALLAN, V. H., JONES, R. B., LEE, R. M., AND ALLAN, S. J. Software pipelining. *ACM Comput. Surv. 27*, 3 (1995), 367–432.

[6] ALLEN, R., AND KENNEDY, K. *Optimizing Compilers for Modern Architectures: A Dependence Based Approach*. Morgan Kaufman, San Francisco, 2002.

[7] AMARASINGHE, S. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Stanford University, 1997.

[8] AMARASINGHE, S. P., AND LAM, M. S. Communication optimization and code generation for distributed memory machines. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation* (New York, NY, USA, 1993), ACM Press, pp. 126–138.

[9] ANCOURT, C., AND IRIGOIN, F. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (April 1991), pp. 39–50.

[10] ANDONOV, R., BALEV, S., RAJOPADHYE, S. V., AND YANEV, N. Optimal semi-oblique tiling. *IEEE Trans. Parallel Distrib. Syst. 14*, 9 (2003), 944–960.

[11] ANDONOV, R., AND RAJOPADHYE, S. Optimal orthogonal tiling of 2-D iterations. *Journal of Parallel and Distributed Computing 45*, 2 (September 1997), 159–165.

[12] ANDONOV, R., RAJOPADHYE, S. V., AND YANEV, N. Optimal orthogonal tiling. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing* (1998), Springer-Verlag, pp. 480–490.

[13] BAGNARA, R., HILL, P. M., AND ZAFFANELLA, E. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006.

[14] BASTOUL, C. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques* (Juan-les-Pins, september 2004), pp. 7–16.

[15] BIKSHANDI, G., GUO, J., HOEFLINGER, D., ALMASI, G., FRAGUELA, B. B., GARZARAN, M. J., PADUA, D., AND VON PRAUN, C. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (2006), pp. 48–57.

[16] BILMES, J., ASANOVIC, K., CHIN, C.-W., AND DEMMEL, J. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing* (1997), ACM Press, pp. 340–347.

[17] BONDHUGULA, U., BASKARAN, M., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)* (Apr. 2008).

[18] BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. A practical and fully automatic polyhedral program optimization system. In *ACM SIGPLAN PLDI* (June 2008).

[19] BORDAWEKAR, R., CHOUDHARY, A., AND RAMANUJAM, J. Automatic optimization of communication in compiling out-of-core stencil codes. In *ICS '96: Proceedings of the 10th international conference on Supercomputing* (1996), ACM Press, pp. 366–373.

[20] BOULET, P., DARTE, A., RISSET, T., AND ROBERT, Y. (pen)-ultimate tiling? *Integr. VLSI J. 17*, 1 (1994), 33–51.

[21] BOYD, S., KIM, S. J., VANDENBERGHE, L., AND HASSIBI, A. A tutorial on Geometric Programming. *To appear in Optimization and Engineering* (2006).

[22] BOYD, S., AND VANDENBERGHE, L. *Convex Optimization*. Cambridge University Press. (Online version available at: `http://www.stanford.edu/~boyd/cvxbook.html`), 2004.

[23] BROMLEY, M., HELLER, S., MCNERNEY, T., AND GUY L. STEELE, J. Fortran at ten Gigaflops: the connection machine convolution compiler. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* (1991), ACM Press, pp. 145–156.

[24] BURGER, D., AND AUSTIN, T. M. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News 25*, 3 (1997), 13–25.

[25] CALLAHAN, D., CARR, S., AND KENNEDY, K. Improving register allocation for subscripted variables. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation* (New York, NY, USA, 1990), ACM Press, pp. 53–65.

[26] CALLAND, P.-Y., AND RISSET, T. Precise tiling for uniform loop nests. In *ASAP '95: Proceedings of the IEEE International Conference on Application Specific Array Processors* (Washington, DC, USA, 1995), IEEE Computer Society, p. 330.

[27] CARR, S., AND KENNEDY, K. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst. 16*, 6 (1994), 1768–1810.

[28] CARR, S., AND SWEANY, P. An experimental evaluation of scalar replacement on scientific benchmarks. *Software Practice and Experience 33*, 15 (2003), 1419–1445.

[29] CARTER, L. Tiling, the universal optimization. Dagstuhl Seminar on Tiling for Optimal Resource Utilization, August 24-28 1998.

[30] CARTER, L., FERRANTE, J., HUMMEL, F., ALPERN, B., AND GATLIN, K. Hierarchical tiling: A methodology for high performance. Tech. Rep. CS96-508, UCSD, Nov. 1996.

[31] CARTER, L., FERRANTE, J., AND HUMMEL, S. F. Hierarchical tiling for improved superscalar performance. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing* (Washington, DC, USA, 1995), IEEE Computer Society, pp. 239–245.

[32] CASCAVAL, C., AND PADUA, D. A. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing* (2003), ACM Press, pp. 150–159.

[33] CHAME, J., AND MOON, S. A tile selection algorithm for data locality and cache interference. In *Proceedings of the 13th international conference on Supercomputing* (1999), ACM Press, pp. 492–499.

[34] CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (2001), ACM Press, pp. 286–297.

[35] CLAUSS, P. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze and transform scientific programs. In *Proceedings of the 10th international conference on Supercomputing* (1996), ACM Press, pp. 278–285.

[36] COLEMAN, S., AND MCKINLEY, K. S. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (1995), ACM Press, pp. 279–290.

[37] Nvidia CUDA toolkit for GPUs. Available at: `http://www.nvidia.com/object/cuda_home.html`.

[38] DARTE, A., ROBERT, Y., AND VIVIEN, F. *Scheduling and Automatic Parallelization.* Birkhauser Boston, 2000.

[39] DEMMEL, J., DONGARRA, J., EIJKHOUT, V., FUENTES, E., PETITET, A., VUDUC, R., WHALEY, R., AND YELICK, K. Self-Adapting Linear Algebra Algorithms and Software. *Proceedings of the IEEE 93*, 2 (2005), 293.

[40] DINKEL, J. J., KOCHENBERGER, M. S., AND WONG, S. N. Sensitivity analysis procedures for geometric programs: Computational aspects. *ACM Trans. Math. Softw. 4*, 1 (1978), 1–14.

[41] DONGARRA, J., BOSILCA, G., CHEN, Z., EIJKHOUT, V., FAGG, G., FUENTES, E., LANGOU, J., LUSZCZEK, P., PJESIVAC-GRBOVIC, J., SEYMOUR, K., ET AL. Self-adapting numerical software (SANS) effort. *IBM Journal of Research and Development 50*, 2/3 (2006), 223.

[42] DUFFIN, R., PETERSON, E., AND ZENER, C. *Geometric Programming - Theory and Applications*. John Wiley, 1967.

[43] EAVES, B. C., AND ROTHBLUM, U. G. A theory on extending algorithms for parametric problems. *Math. Oper. Res. 14*, 3 (1989), 502–533.

[44] EAVES, B. C., AND ROTHBLUM, U. G. Dines-Fourier-Motzkin quantifier elimination and an application of corresponding transfer principles over ordered fields. *Mathematical Programming 53*, 1-3 (1992), 307–321.

[45] ESSEGHIR, K. Improving data locality for caches. Master's thesis, Rice University, September 1993.

[46] FERNANDES, R., PINGALI, K., AND STODGHILL, P. Mobile MPI programs in computational grids. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2006), ACM Press, pp. 22–31.

[47] FERRANTE, J., SARKAR, V., AND THRASH, W. On estimating and enhancing cache effectiveness. In *Fourth International Workshop on Languages and Compilers for Parallel Computing* (August 1991), U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., Lecture Notes on Computer Science 589, Springer Verlag, pp. 328–343.

[48] FRAGUELA, B. B., CARMUEJA, M. G., AND ANDRADE, D. Optimal tile size selection guided by analytical models. In *PARCO* (2005), pp. 565–572.

[49] FRUMKIN, M. A., AND DER WIJNGAART, R. F. V. Tight bounds on cache use for stencil operations on rectangular grids. *J. ACM 49*, 3 (2002), 434–453.

[50] GHOSH, S., MARTONOSI, M., AND MALIK, S. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst. 21*, 4 (1999), 703–746.

[51] GOUMAS, G., ATHANASAKI, M., AND KOZIRIS, N. An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems 14*, 10 (October 2003).

[52] GROPP, W. D. Solving PDEs on loosely-coupled parallel processors. *Parallel Computing 5*, 1-2 (1987), 165–173.

[53] GRÖSSLINGER, A., GRIEBL, M., AND LENGAUER, C. Introducing non-linear parameters to the polyhedron model. In *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004)* (July 2004), M. Gerndt and E. Kereku, Eds., Research Report Series, LRR-TUM, Technische Universität München, pp. 1–12.

[54] HARPER, J. S., KERBYSON, D. J., AND NUDD, G. R. Analytical modeling of set-associative cache behavior. *IEEE Trans. Comput. 48*, 10 (1999), 1009–1024.

[55] HiTLoG: Hierarchical Tiled Loop Generator. Available at: `http://www.cs.colostate.edu/MMAlpha/HiTLoG/`.

[56] HODZIC, E., AND SHANG, W. On supernode transformation with minimized total running time. *IEEE Trans. Parallel Distrib. Syst. 9*, 5 (1998), 417–428.

[57] HÖGSTEDT, K., CARTER, L., AND FERRANTE, J. Determining the idle time of a tiling. In *POPL* (1997), pp. 160–173.

[58] HOGSTEDT, K., CARTER, L., AND FERRANTE, J. On the parallel execution time of tiled loops. *IEEE Trans. Parallel Distrib. Syst. 14*, 3 (2003), 307–321.

[59] HSING HSU, C., AND KREMER, U. A quantitative analysis of tile size selection algorithms. *J. Supercomput. 27*, 3 (2004), 279–294.

[60] HSU, C., AND KREMER, U. Tile selection algorithms and their performance models. Tech. Rep. DCS-TR-401, CS Dept., Rutgers University, Oct. 1999.

[61] HU, Y. C., JIN, G., JOHNSSON, S. L., KEHAGIAS, D., AND SHALABY, N. HPFBench: a high performance fortran benchmark suite. *ACM Trans. Math. Softw. 26*, 1 (2000), 99–149.

[62] IRIGOIN, F., AND TRIOLET, R. Supernode partitioning. In *15th ACM Symposium on Principles of Programming Languages* (Jan 1988), ACM, pp. 319–328.

[63] JIMÉNEZ, M., LLABERÍA, J. M., AND FERNÁNDEZ, A. Register tiling in nonrectangular iteration spaces. *ACM Trans. Program. Lang. Syst. 24*, 4 (2002), 409–453.

[64] JIMÉNEZ, M., LLABERÍA, J. M., AND FERNÁNDEZ, A. Register tiling in nonrectangular iteration spaces. *ACM Trans. Program. Lang. Syst. 24*, 4 (2002), 409–453.

[65] JIMÉNEZ, M., LLABERÍA, J. M., AND FERNÁNDEZ, A. A cost-effective implementation of multilevel tiling. *IEEE Trans. Parallel Distrib. Syst. 14*, 10 (2003), 1006–1020.

[66] KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., AND SHIPPY, D. Introduction to the cell multiprocessor. *IBM J. Res. Dev. 49*, 4/5 (2005), 589–604.

[67] KAMIL, S., DATTA, K., WILLIAMS, S., OLIKER, L., SHALF, J., AND YELICK, K. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness* (New York, NY, USA, 2006), ACM, pp. 51–60.

[68] KAMIL, S., HUSBANDS, P., OLIKER, L., SHALF, J., AND YELICK, K. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance* (2005), ACM Press, pp. 36–43.

[69] KARP, R. M., MILLER, R. E., AND WINOGRAD, S. The organization of computations for uniform recurrence equations. *J. ACM 14*, 3 (1967), 563–590.

[70] KELLY, W., PUGH, W., AND ROSSER, E. Code generation for multiple mappings. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation* (McLean, VA, 1995).

[71] KIM, D., RENGANARAYANA, L., ROSTRON, D., RAJOPADHYE, S., AND STROUT, M. M. Multi-level tiling: m for the price of one. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)* (November 2007).

[72] KISUKI, T., KNIJNENBURG, P. M. W., AND O'BOYLE, M. F. P. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2000), IEEE Computer Society, p. 237.

[73] KNIJNENBURG, P. M. W., KISUKI, T., AND O'BOYLE, M. F. P. Iterative compilation. In *Embedded processor design challenges: systems, architectures, modeling, and simulation-SAMOS*. Springer-Verlag New York, Inc., New York, NY, USA, 2002, pp. 171–187.

[74] KORTANEK, K. O., XU, X., AND YE, Y. An infeasible interior-point algorithm for solving primal and dual geometric programs. *Math. Program. 76*, 1 (1997), 155–181.

[75] KYPARISIS, J. Sensitivity analysis in geometric programming: theory and computations. *Ann. Oper. Res. 27*, 1-4 (1990), 39–64.

[76] LAM, M. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (New York, NY, USA, 1988), ACM Press, pp. 318–328.

[77] LAM, M. D., ROTHBERG, E. E., AND WOLF, M. E. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems* (1991), ACM Press, pp. 63–74.

[78] LAM, M. S., AND WOLF, M. E. A data locality optimizing algorithm (with retrospective). In *Best of PLDI* (1991), pp. 442–459.

[79] LE VERGE, H., VAN DONGEN, V., AND WILDE, D. La synthèse de nids de boucles avec la bibliothèque polyédrique. In *RenPar'6* (Lyon, France, Juin 1994). English version "Loop Nest Synthesis Using the Polyhedral Library"in IRISA TR 830, May 1994.

[80] LE VERGE, H., VAN DONGEN, V., AND WILDE, D. Loop nest synthesis using the polyhedral library. Tech. Rep. PI 830, IRISA, Rennes, France, May 1994. Also published as INRIA Research Report 2288.

[81] LI, Z., AND SONG, Y. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst. 26*, 6 (2004), 975–1028.

[82] LÖFBERG, J. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference* (Taipei, Taiwan, 2004). Available from `http://control.ee.ethz.ch/~joloef/yalmip.php`.

[83] LOWENTHAL, D. K. Accurately selecting block size at runtime in pipelined parallel programs. *Int. J. Parallel Program. 28*, 3 (2000), 245–274.

[84] MCKELLAR, A. C., AND E. G. COFFMAN, J. Organizing matrices and matrix operations for paged memory systems. *Commun. ACM 12*, 3 (1969), 153–165.

[85] MITCHELL, N., HOGSTEDT, N., CARTER, L., AND FERRANTE, J. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming 26*, 6 (1998), 641–670.

[86] MOLDOVAN, D. I., AND FORTES, J. A. B. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Comput. 35*, 1 (1986), 1–12.

[87] MOON, S., AND SAAVEDRA, R. Hyperblocking: A data reorganization method to eliminate cache conflicts in tiled loop nests. Tech. Rep. TR-98-671, University of Southern California, February 1998.

[88] NAS Parallel Benchmarks. Available from `http://www.netlib.org/parkbench/`.

[89] NAVARRO, J. J., JUAN, T., AND LANG, T. MOB forms: a class of multilevel block algorithms for dense linear algebra operations. In *Proceedings of the 8th international conference on Supercomputing* (1994), ACM Press, pp. 354–363.

[90] NIKOLOPOULOS, D. S. Dynamic tiling for effective use of shared caches on multithreaded processors. *International Journal of High Performance Computing and Networking* (2004), 22 – 35.

[91] OHTA, H., SAITO, Y., KAINAGA, M., AND ONO, H. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *ICS '95: Proceedings of the 9th international conference on Supercomputing* (New York, NY, USA, 1995), ACM Press, pp. 270–279.

[92] PARKBENCH: PARallel Kernels and BENCHmarks. Available from `http://www.netlib.org/parkbench/`.

[93] PUGH, W. Omega test: A practical algorithm for exact array dependency analysis. *Comm. of the ACM 35*, 8 (1992), 102.

[94] QUILLERÉ;, F., RAJOPADHYE, S., AND WILDE, D. Generation of efficient nested loops from polyhedra. *International Journal Parallel Programming 28*, 5 (2000), 469–498.

[95] QUINTON, P., AND VAN DONGEN, V. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing 1*, 2 (1989), 95–113.

[96] RAJOPADHYE, S. V., AND FUJIMOTO, R. M. Synthesizing systolic arrays from recurrence equations. *Parallel Computing 14* (June 1990), 163–189.

[97] RAMANUJAM, J. Optimal software pipelining of nested loops. In *IPPS* (1994), pp. 335–342.

[98] RAMANUJAM, J., AND SADAYAPPAN, P. Tiling multidimensional itertion spaces for multicomputers. *J. Parallel Distrib. Comput. 16*, 2 (1992), 108–120.

[99] RASTELLO, F., AND ROBERT, Y. Automatic partitioning of parallel loops with parallelepiped-shaped tiles. *IEEE Trans. Parallel Distrib. Syst. 13*, 5 (2002), 460–470.

[100] RAU, B. R. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture* (New York, NY, USA, 1994), ACM Press, pp. 63–74.

[101] RENGANARAYANA, L., AND RAJOPADHYE, S. A geometric programming framework for optimal multi-level tiling. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 18.

[102] RENGANARAYANA, L., UPADRASTA, R., AND RAJOPADHYE, S. Optimal ILP and register tiling: Analytical model and optimization framework. In *LCPC 2005: 12th International Workshop on Languages and Compilers for Parallel Computing* (2005), Springer Verlag.

[103] RENGANARAYANAN, L., HARTHI-KOTE, M., DEWRI, R., AND RAJOPADHYE, S. Towards optimal multi-level tiling for stencil computations. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS) (to appear)* (2007).

[104] RENGANARAYANAN, L., KIM, D., RAJOPADHYE, S., AND STROUT, M. M. Parameterized tiled loops for free. In *PLDI '07: ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), ACM Press, pp. 405–414.

[105] RIVERA, G., AND TSENG, C.-W. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation* (1998), ACM Press, pp. 38–49.

[106] RIVERA, G., AND TSENG, C.-W. Eliminating conflict misses for high performance architectures. In *Proceedings of the 12th international conference on Supercomputing* (1998), ACM Press, pp. 353–360.

[107] RIVERA, G., AND TSENG, C.-W. A comparison of compiler tiling algorithms. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction* (1999), Springer-Verlag, pp. 168–182.

[108] RIVERA, G., AND TSENG, C.-W. Locality optimizations for multi-level caches. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 1999), ACM Press, p. 2.

[109] RIVERA, G., AND TSENG, C.-W. Tiling optimizations for 3D scientific computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* (2000), IEEE Computer Society, p. 32.

[110] RONG, H., DOUILLET, A., AND GAO, G. R. Register allocation for software pipelined multi-dimensional loops. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 154–167.

[111] RONG, H., DOUILLET, A., GOVINDARAJAN, R., AND GAO, G. R. Code generation for single-dimension software pipelining of multi-dimensional loops. In *CGO '04: Proceedings of the international symposium on Code generation and optimization* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 175–?

[112] RONG, H., TANG, Z., GOVINDARAJAN, R., DOUILLET, A., AND GAO, G. R. Single-dimension software pipelining for multi-dimensional loops. In *CGO '04: Proceedings of*

the international symposium on Code generation and optimization (Washington, DC, USA, 2004), IEEE Computer Society, p. 163.

[113] ROTH, G., MELLOR-CRUMMEY, J., KENNEDY, K., AND BRICKNER, R. G. Compiling stencils in high performance fortran. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)* (1997), ACM Press, pp. 1–20.

[114] SARKAR, V. Automatic selection of high-order transformations in the IBM XL FORTRAN compilers. *IBM J. Res. Dev. 41*, 3 (1997), 233–264.

[115] SARKAR, V. Optimized unrolling of nested loops. *International Journal of Parallel Programming 29*, 5 (2001), 545–581.

[116] SARKAR, V., AND MEGIDDO, N. An analytical model for loop tiling and its solution. In *Proceedings of ISPASS* (2000).

[117] SCHREIBER, R., AND DONGARRA, J. Automatic blocking of nested loops. Tech. Rep. 90.38, RIACS, NASA Ames Research Center, Aug 1990.

[118] SMITH, M. D. Overcoming the challenges to feedback-directed optimization. In *DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization* (New York, NY, USA, 2000), ACM Press, pp. 1–11. Keynote talk.

[119] SPEC CPU2000 benchmark. Available from `http://www.spec.org`.

[120] SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30(3)* (March 2005).

[121] TEMAM, O., FRICKER, C., AND JALBY, W. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (1994), ACM Press, pp. 261–271.

[122] TEMAM, O., GRANSTON, E. D., AND JALBY, W. 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 the 1993 ACM/IEEE conference on Supercomputing* (1993), ACM Press, pp. 410–419.

[123] VALIANT, L. G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (1990), 103–111.

[124] VUDUC, R., DEMMEL, J. W., AND YELICK, K. A. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005* (San Francisco, CA, USA, June 2005), Journal of Physics: Conference Series, Institute of Physics Publishing.

[125] WEISPFENNING, V. Parametric linear and quadratic optimization by elimination. Tech. Rep. MIP-9404, Fakultät für Mathematik und Informatik, Universität Passau, 1994.

[126] WHALEY, R. C., AND DONGARRA, J. J. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)* (1998), IEEE Computer Society, pp. 1–27.

[127] WILSON, R. P., FRENCH, R. S., WILSON, C. S., AMARASINGHE, S. P., ANDERSON, J. M., TJIANG, S. W. K., LIAO, S.-W., TSENG, C.-W., HALL, M. W., LAM, M. S., AND HENNESSY, J. L. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices 29*, 12 (1994), 31–37.

[128] WOLF, M., MAYDAN, D., AND CHEN, D. Combining loop transformations considering caches and scheduling. In *29th International Symposium on Microarchitecture* (December 1996).

[129] WOLF, M. E., MAYDAN, D. E., AND CHEN, D.-K. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture* (Paris, 2–4, 1996), IEEE Computer Society TC-MICRO and ACM SIGMICRO, pp. 274–286.

[130] WOLFE, M. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing* (Philadelphia, PA, USA, 1989), Society for Industrial and Applied Mathematics, pp. 357–361.

[131] WOLFE, M. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing* (1989), ACM Press, pp. 655–664.

[132] WONNACOTT, D. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing* (2000), IEEE Computer Society, p. 171.

[133] WONNACOTT, D. Achieving scalable locality with time skewing. *Int. J. Parallel Program. 30*, 3 (2002), 181–221.

[134] XUE, J. Communication-minimal tiling of uniform dependence loops. *J. Parallel Distrib. Comput. 42*, 1 (1997), 42–59.

[135] XUE, J. On tiling as a loop transformation. *Parallel Processing Letters 7*, 4 (1997), 409–424.

[136] XUE, J. *Loop Tiling For Parallelism*. Kluwer Academic Publishers, 2000.

[137] XUE, J., AND CAI, W. Time-minimal tiling when rise is larger than zero. *Parallel Comput. 28*, 6 (2002), 915–939.

[138] YOTOV, K., LI, X., REN, G., GARZARAN, M. J. S., PADUA, D., PINGALI, K., AND STODGHILL, P. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE 93* (2005), 358–386.