DISSERTATION

PARAMETERIZED AND MULTI-LEVEL TILED LOOP GENERATION

Submitted by

DaeGon Kim

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2010

COLORADO STATE UNIVERSITY

April 30, 2010

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY DAEGON KIM ENTITLED PARAMETERIZED AND MULTI-LEVEL TILED LOOP GENERATION BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Committee Member : Wim Böhm

Committee Member : Edwin K. P. Chong

Committee Member : Michelle Strout

Advisor : Sanjay Rajopadhye

Department Chair : Darrell Whitley

ABSTRACT OF DISSERTATION

PARAMETERIZED AND MULTI-LEVEL TILED LOOP GENERATION

Tiling is a loop transformation that decomposes computations into a set of smaller computation blocks. The transformation has been proven to be useful for many high-level program optimizations, such as data locality optimization and exploiting coarse-grained parallelism, and crucial for architecture with limited resources, such as embedded systems, GPUs, and the Cell architecture. Data locality and parallelism will continue to serve as major vehicles for achieving high performance on modern architecture in multi-core era. In parameterized tiling the size of blocks is not fixed at compile time but remains a symbolic constant so that it can be selected/changed even at runtime. Parameterized tiled loops facilitate iterative and runtime optimizations, such as iterative compilation, auto-tuning and dynamic program adaption. In this dissertation we present a collection of techniques for generating parameterized and multi-level tiled loops from affine control loops and their parallelization.

The tiled loop generation problem even for perfectly nested loops has been believed to have an exponential time complexity due to the heavy machinery like Fourier–Motzkin elimination. Disproving this decade-long belief, we provide a simple technique for generating tiled loop nests even from imperfectly nested loops. Our technique for perfectly nested loops consists of only syntactic processing that is applied only once and independently to each loop bound. Our approach to imperfectly nested

loops is composed of a direct extension of the tiled code generation technique for perfectly nested loops and three simple optimizations on the resulting parameterized tiled loops. The generation as well as the optimizations are achieved only with purely syntactic processing, hence loop generation time remains negligible. We also present three schemes for multi-level tiling where tiling is applied more than once. All the schemes are scalable with respect to the number of tiling levels and can be combined to achieve better performance.

To facilitate parallelization of parameterized tiled loops, we generate outermost tile-loops that are perfectly nested. We also provide a technique for statically restructuring parameterized tiled loops to the wavefront scheduling on shared memory system. Because the formulation of parameterized tiling does not fit into the well established polyhedral framework, such static restructuring has been a great challenge. However, we achieve this limited restructuring through a syntactic processing without any sophisticated machinery.

DaeGon Kim
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
Summer 2010

ACKNOWLEDGMENTS

My journey to academia and research was not planned, and I was not prepared. However, when I look back, the journey has been joyful and my heart becomes thankful. Throughout and in the center of this journey there has always been a lighthouse providing everything I need to finish this journey. I am thankful to my advisor, Dr. Sanjay Rajopadhye, for not only providing invaluable teaching, directions, and encouragement but also being such a great role model. Without him, I would have been shipwrecked during this journey and been lost deep under the formidable sea of academia.

I would like to thank Dr. Michelle Strout for interesting discussions and timely feedback. I would like to thank Dr. Wim Böhm for his questions. They always remind me the importance of honesty and basics. I have been privileged to take classes from Dr. Robert Liebler. Thanks to him for teaching and inspiring me to see things in many different ways. I would like to thank Dr. Edwin Chong for his wondeful teaching on nonlinear optimization and agreeing to serve on my graduate committee at short notice.

I would like to thank to the group of Albert Cohen at INRIA in France and the group of Kevin O'Brien at the IBM TJ Watson Research Center at Yorktown Heights for memorable and vaulable experience both personally and professionally.

I have been lucky to work with Gautam Gupta and Lakshminarayanan Renganarayanan. They have been not only great colleagues but also good friends. Whenever I had "looking-good but not-working" ideas, Gautam gave me a counter example in a

few minutes. At those moments, I was not sure whether I truly liked him. Lakshmi gave me a good start on the research in this dissertation, and he also literally created an opportunity to work at IBM research lab for the lazy internship seeker. I would like to thank all the members of the MELANGE group at CSU and all the students in HPC student room.

I want to thank National Science Foundation for financial support. I would like to thank Sharon Van Gorder and Caroll Calliham for gladly helping me go through all the administrative process.

I would like to thank my parents for their love and support. They have been giving me something I will never be able to pay back, and I will remember it. I would like to thank my brother and my sister. They have been so nice to their younger brother. I would like to thank Jerry Hager for giving such a wonderful experience in this foreign country. He is the most surprising gift that I have in my homesick life.

Finally, I would like to express my appreciation to YoonHee just for being my wife. She always believes in me and supports me. I also would like to thank Eunsoo and Eunbee for being my daughters. Although these two little girls are supposed to have zero interests in serving me, they have provided me great energy in a nontraditional way. They are the very three persons that make me want to be a better man. I dedicate this dissertation to them.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Achieving high performance on modern architectures is a demanding challenge. As the trend of multiple cores on a single chip continues and the number of cores keeps increasing to fulfill the performance improvement implied by Moore's law, this becomes even more complex, requiring parallelism in applications being exposed, and the parallel machines to be exploited, while enhancing data locality.

Many compute- and data-intensive applications spend most time on executing loops. An important class of such kernels is affine control loops. Tiling is a very useful loop transformations for high level optimization. The tiling transformation decomposes computation into a set of smaller blocks. Over the past few decades of research and practical use in high performance implementations, it has been proven to be effective for improving data locality and exposing coarse grained parallelism [28, 37, 57].

The blocks of computation, called tiles, are mapped to one of the resources in systems. The resources can be memory hierarchy, registers, functional units or cores/processors. The size of blocks is chosen to make data and/or computation fit into a certain resource, but also to provide a fine control on the ratio between computation and communication. Hence, the tile sizes have a great impact on performance, and a lot of research has addressed the problem of finding good and optimal tile sizes [1, 7, 15, 27, 50, 37, 55].

To fully exploit computation/communication/memory resources for high performance, optimizing compilers must choose optimal tile sizes based on an accurate model of target architectures. However, obtaining such models is difficult because of the complexity and diversity of modern architectures and dynamic changes in execution environment, as in cloud and mobile computing. Due to this, empirical search for optimal parameters has gained intensive research interest in recent years [13, 55, 35, 17, 54]. In this context, it is highly desirable to employ parameterized tiling where the tile sizes are not fixed at compile time, and remain symbolic constants. Although tile size selection has received a lot of attention, tiled code generation has not until recently.

In parameterized tiling tile sizes can be chosen and changed at runtime and even during a single run. Parameterization of tile sizes is quite useful for any empirical tuning/search in auto-tuners (ATLAS [55], PHiPAC [13] and SPIRAL [48]) and iterative compilers, runtime feedback systems [42, 44] and dynamic program adaption [35, 36].

High performance implementations often use multiple levels of tiling to harness the full potential of modern architectures. For example, the highly tuned matrix operation libraries generated by ATLAS [55] or PHiPAC [13] use two levels of tiling: one for caches and another for registers and instruction-level parallelism. High performance implementations for solving partial differential equations [51] use at least two levels of tiling: one for parallelism and one for caches. With advent of multi-core processors, multi-level tiling has become a design pattern for high performance implementations [12, 18].

In this dissertation we present a collection of techniques for multi-level, parameterized tiled loop generation and their parallelization. The most distinctive nature of our approach is that none of our techniques require any sophisticated machinery, such as Fourier–Motzkin elimination and projection. Rather, we only use purely syntactic processing of input loop nests, even to restructure parameterized tiled loops for wave-front scheduling. The theoretical complexity of generating tiled loop including parameter-

ized ones is reduced from the exponential complexity of FME, down to be linear in the number of loop bounds in the original loop nest. Furthermore, our approach provides multi-level parameterized tiled loops at almost the same cost of generating tiled loops for a single level of tiling. Providing three schemes and their combination, our approach provides various tiled loop structures, which can be selected depending on optimization aimed to given levels.

The remainder of this dissertation is broadly separated into three parts. Chapter 2 provides an overview of tiled loop generation problem and its related work. Chapter 3-8 provide detailed explanation of our approach. Chapter 9 concludes the discussion and presents direction of future research.

Chapter 2

Related Work

Applying tiling transformation as a complete optimization requires a dependence analysis, pre-processing loops for tiling legality, and tiled loop generation together with appropriate statement generation. The tiling transformation is an iteration (computation) reordering transformation. In general, such reordering is not always legal, i.e., does not preserve semantics of original programs. The role of data dependence analysis in the tiling optimization process is providing the constraints to be respected. Validation of the legality of tiling as well as required pre-processing of loops to ensure the legality is based on this constraint. Once the legality issue is resolved, tiled loops will be generated to realize the new ordering given by tiling transformation. Tiling as a transformation is a vehicle for other optimizations, such as data locality optimizations and coarse-grained parallelization. A specific optimization may require more than just the reordering of computation. Hence, in general appropriate statement generation is required. For example, parallelization for distributed memory systems requires an additional piece of code for communication among the processors, and data locality optimization aimed for registers requires loop unrolling with scalar promotion of array access.

This dissertation focuses on parameterized, multi-level sequential and parallel tiled loop generation from *tilable affine control loops* with *hyper-rectangular shaped tiles* and their parallelization. We assume that tiling is legal and do not address legality of

```
for (t = 1; t <= TMAX; t++)
  for (i = 1; i <= N-2; i++)
    A[i] = (A[i-1]+A[i]+A[i+1])/3.0;
```

Figure 2.1: Stencil computation on one dimensional array

tiling. Affine control loops are loop nests where loop bounds are affine expressions of outer iterators and program size parameters that are loop invariant variables. Strictly speaking, affine control loops have another restriction on statements, namely that they access array or scalar variables using affine functions of surrounding indices. However, since tiled loop generation does not change statements but only loops, it is not relevant in the context of tiled loop generation problem where tiling is assumed to be legal. The hyper-rectangular tile is the most widely used in the literature and practice. Although some loop programs do not allow rectangular tiling, any computations that can be tiled with parallelogram shape can be tiled after skewing transformation.

2.1 Background

Consider the simple, perfectly nested and doubly nested loop nest in Figure 2.1. We will apply the tiling transformation to this. The iteration space of this computation is shown in Figure 2.2. Although there is a very efficient method for tiling rectangular iteration space with rectangular tiles, the loop nest is not tilable with rectangular tile shape.

Note that the loops are not fully permutable. To make rectangular tiling legal, we apply a skewing transformation to the original iteration space by $(t, i \rightarrow t, t+i)$. The transformed iteration space is also shown in Figure 2.2 and its corresponding loops are given in Figure 2.3. The transformed iteration space is no longer rectangular and we denote it as

$$P_{iter} = \{t, i \mid 1 \leq t \leq Tmax \wedge t+1 \leq i \leq t+N-2\}$$

Now, we apply the tiling transformation to this transformed iteration space. The

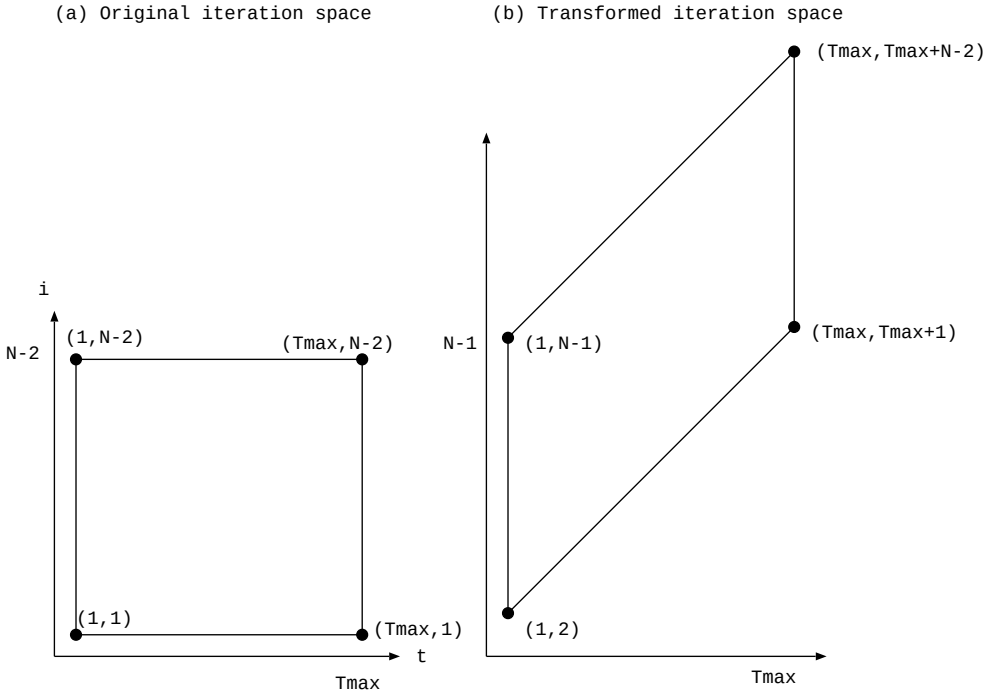

Figure 2.2: Original iteration space of the stencil computation in Figure 2.1 and its transformed iteration space by $(t, i \rightarrow t, t + i)$ to make rectangular tiling legal

```
for (t = 1; t <= TMAX; t++)
  for (i = t+1; i <= t+N-2; i++)
    A[t-i] = (A[t-i-1]+A[t-i]+A[t-i+1])/3.0;
```

Figure 2.3: Transformed tilable (fully permutable) loops of the stencil computation in Figure 2.1

tiling transformation decomposes the transformed iteration space into a set of smaller blocks as shown in Figure 2.4. Tiles are represented as dotted-line rectangles, and the lexicographically smallest points in the tiles are called *tile origins*. There are three kinds of tiles: empty tiles, partial tiles and full tiles. This classification of tiles is based on which points inside a tile belong to iteration space. When all the points belong to iteration space, a tile is called a *full tile*. When none of points belongs to iteration space, it is an *empty tile*. The tile that is neither full or empty is a *partial tile*. In the tiled loop nest, tiles are enumerated with a doubly nested loop, whose body is itself a loop that visits all the iteration points inside the tile. So, the structure of tiled loops are two outer loops enumerating tile origins and two inner loops enumerating all the points for a given iteration of the outer two loops. The loops that enumerate tiles are called *tile-loops* and the loops that enumerate iteration points in a tile are called *point-loops*.

Since there are no iteration points in empty tiles, we need to enumerate only full and partial tiles. The ideal case is that no empty tiles are enumerated. One also might want to distinguish full tiles from the others so that optimizations requiring the loop bounds to be fixed or known, such as loop unrolling, register tiling and vectorization, can be applied in later stages. When the sizes of tiles, i.e., its width and height, are (fixed) integer constants in the tiled loop nest, such tiling is called fixed tiling. In parameterized tiling, the tile sizes are not determined at compile time, but remain symbolic runtime parameters.

The space that describes the iteration space of tiled loop nest can be formulated in two ways. The first formulation is

$$\{Tt, Ti, t, i \mid t - St < St \times Tt \leq t \wedge i - Si < Si \times Ti \leq i \wedge (t, i) \in P_{iter}\}$$

where St and Si are tile sizes. We call this expanded four dimensional space as *tiled space*. Since the first two dimensions of the tiled space describes the space of tiles, they are called *tile space*.

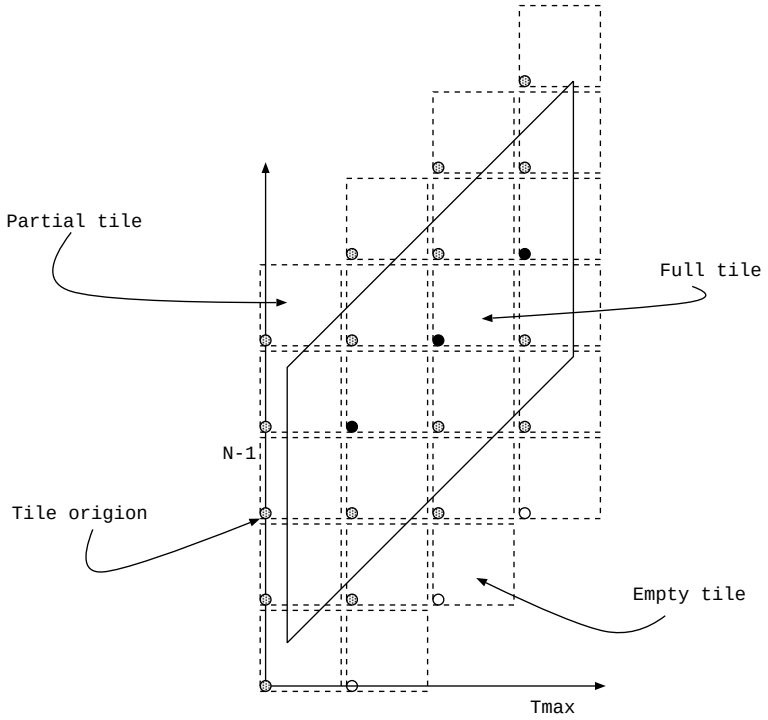

Figure 2.4: Tiled (transformed) iteration space of the stencil computation in Figure 2.1

When the tile sizes are integer constants, the tiled space is a polyhedron. For example, when $St = 3$ and $Si = 2$, it becomes

$$\{Tt, Ti, t, i \mid t - 3 < 3 \times Tt \leq t \wedge i - 2 < 2 \times Ti \leq i \wedge (t, i) \in P_{iter}\}$$

and all the constraints are linear inequalities. The tile space can be viewed as the image of this tiled space by the projection function $(Tt, Ti, i, j \rightarrow Tt, Ti)$ and can be easily computed using standard polyhedral library like Omega [47], PolyLib [45] and Parma Library[8]. In this formation, tile space is a convex (dense) set.

Another formulation is that tile origins are on a single (fixed constant or parameterized) integral lattice. Let us denote the set of all the points on this lattice as $\mathcal{L}$. For example, for $St = 3$ and $Si = 2$ all the tile origins can be expressed as $(3 \times e_1, 2 \times e_2)$ for some integer e_1 and e_2. The tile space in this formulation cannot be expressed as

$$\mathcal{L} \cap P_{iter}$$

because not all the origins of necessary tiles belong to P_{iter}. This can be easily seen in Figure 2.4. When a set contains all the necessary tile origins, i.e., the origins of all full and partial tiles, we call such set as an outset P_{out} of P_{iter} [52]. Now, the tile space can be expressed as

$$\mathcal{L} \cap P_{out}$$

Note that P_{out} is based on not only P_{iter} but also tile sizes.

The space of points inside a tile can be expressed as

$$tile(p) \cap P_{iter}$$

where p is the tile origin (Tt, Ti) and $tile(p)$ is $\{t, i \mid Tt \leq t < Tt + St \wedge Ti \leq i < Ti + Si\}$.

When tile sizes are fixed, this space can be viewed as a polyhedron parameterized by tile origin and the program size parameters.

2.2 Related Work

A considerable amount of research has been done for tiled loop generation problem. First, a significant work has addressed the case where loops are perfectly nested and tile sizes are fixed, and then there have been many advances. Roughly speaking, these advances are towards handling imperfectly nested loops, separation of full tiles, generating parameterized tiled loops, wave-front scheduling of tiled program and generating multi-level tiled loops.

2.2.1 Tiled Loop Generation with Fixed Tile Sizes

The problem of generating tiled loops has been closely related to that of generating scanning loops from polyhedra. Thanks to Irigoin and Triolet's proof that tiled iteration space is a polyhedron if tile sizes are integer constants [29], the techniques for generating

loops that scan polyhedra can be directly used when tile sizes are fixed. Ancourt and Irigoin [6] proposed a technique for scanning a single polyhedron, based on Fourier-Motzkin elimination over inequality constraints. Le Verge et al. [39, 38] 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. [32] solved this by extending the Ancourt-Irigoin technique, and together with a number of sophisticated optimizations, developed the widely distributed Omega library [47]. The SUIF [56] tool includes a similar algorithm. Quilleré et al. proposed a dual representation algorithm [49] for scanning the union of polyhedra, and this algorithm is implemented in the CLooG code generator [10] and its derivative Wloog is used in the WRaP-IT project [11]. Recently, CLooG is used in gcc Graphite project [46].

Goumas et al. [22] show situations where the generation of tiled loops may become expensive given the doubly exponential complexity of FME and propose an approach where the tiled loop generation problem is decomposed into two sub-problems: generating tile-loops and generating point-loops. Through this decomposition, tiled loop generation for loop nest of depth d is done by scanning two d-dimensional polyhedra instead of a single $2d$-dimensional polyhedron, reducing the worst-case generation time significantly.

Broadly speaking, there are two approaches for tiling imperfectly nested loops. The first approach is based on splitting iteration space into two regions based on legality of tiling and tiling can be applied only to the region where tiling is legal [16]. The second approach that is widely accepted is embedding [2, 41, 14]. In this approach, all the iteration spaces are embedded into a common space. Amhed et al. [2] proposed an embedding method based on *product space*, the Cartesian product of all the iteration spaces. They construct product space, apply techniques developed for perfectly nested

loops to obtain a legal locality-optimized loop nest while reducing the dimensionality of this space. Note that the number of dimension can be reduced up to the maximum depth of loops. Lim et al. [41] proposed a technique for the case that computation can be decomposed into a set of independent threads. They provided an algorithm for extracting the maximum number of fully permutable loops using their affine partitioning framework [40]. Bondhugula et al. [14] proposed an approach based on a formulation for relaxed pipeline scheduling and formulated an integer linear optimization problem to minimize the distance between production and consumption. The focus is not code generation, but a pre-processing for legality of tiling. Standard techniques for generating loops in polyhedral model can be directly used for tiled loop generation because tile sizes are fixed.

2.2.2 Tiled Loop Generation with Parameterized Tile Sizes

When tile sizes are not constants, but symbolic parameters, the above techniques are not directly applicable. Amarasinghe and Lam [4, 5] 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. [23] 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 generality 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.

Jiménez et al. [30] developed code generation techniques for register tiling of non-rectangular iteration spaces. They generate code that traverses the bounding box of the 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, the resulting code experiences significant code expansion.

Hartono et al. [24] presented a technique for generating parameterized tiled loops from imperfectly nested loops. They do not rely on complex polyhedral operation but use syntactic manipulation after they obtained embedded original (non-tiled) loop nest and implemented a tool call PrimeTile. However, their tiling is different from the conventional tiling in the literature where all the tile origins are on a (either fixed constant or parameterized) single lattice. For each loop, PrimeTile splits it into three loops: prologue, main and epilogue. The main loop enumerates full tile coordinates and the other loops enumerate partial tile coordinates. This process repeats until the innermost loop. The algorithm itself distinguishes full tiles from the others, but results in substantial increase in the code size. Therefore, their resulting code is complex and difficult to parallelize except computations that have already synchronization-free parallelism. Their method allows non-affine loop bounds at the price of checking all the loop iteration to extract tiled loop bounds during runtime. They focus on extracting full tiles from affine control loops, and some part of iteration space may not be tiled. They achieved great reduction of a loop overhead even for very small tile sizes.

Hartono et al. [25] proposed a solution to parallelization of parameterized tiled code based on inspector/executor. The inspector computes the range of the time steps for a given schedule and the tiles being executed at given a time step by the schedule. The executor iterates over doubly nested lists. However, they abandon their previous approach [24] for tiling on sequential machines. Rather, they use convex hull of all iteration spaces to obtain a simple tiled loops suitable parallel execution, similar to the solution proposed here.

Recently, Baskaran et al. [9] proposed an approach for parallelization of parameterized tiled loop. To extract the bounds for the outermost time loop, they used parametric

integer linear programming [20]. However, given that either constraints or wave-front scheduling is not affine function of loop iterators and program parameters, it is not clear how such tools can be directly used. The exact formulation of parametric integer programming problem has not been given. They also proposed a relaxed symbolic Fourier Motzkin Elimination to optimize tile-loops, but provide neither the explanation of the effectiveness in tightening loop bounds nor experimental result for its impact on execution time.

2.2.3 Multi-level Tiled Loop Generation

For simple rectangular iteration spaces, multi-level tiled loop generation is straightforward and has been used by several tools. Jiménez et al. [31] proposed 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 an exponential complexity that grows with the number of levels of tiling.

Hartono et al. [24, 25] extended their technique for single level tiling to multi-level tiling. Baskaran et al. [9] proposed a technique for multi-level tiling based on relaxed symbolic Fourier Motzkin Elimination.

Others [53] have also looked at multi-level tiling (e.g., Mitchell et al. [43]) without addressing code generation.

2.3 Putting Our Work into Context

The unique nature of our technique is that it consists of only syntactic processing of original loop nests for all the generations and tiled loop restructuring for parallelization. Although PrimeTile [24] also uses only syntactic manipulation of the original loop nests, it does not maintain all the tile origins on a single lattice. This is different from the conventional tiling. The other difference is that their tiled loop structure are complex

and difficult to transform by repeatedly splitting loops into prologue, main and epilogue. Consequently, they do not support tiled loop structuring for wave-front scheduling of tiles.

In our approach, tiled loop generation problem is decomposed into two sub-problems similar to that of Goumas et al. [22]. The major difference is, while their technique can generate tiled loop nest with parallelogram tile shape and fixed tile sizes, our technique can produce parameterized tiled loops with hyper-rectangular shape. Their tile space is not an outset, but an image of an outset by the tiling transformation function that maps lattice points into a dense set. This results in ceilings and floor operations in loop bounds of tile-loops.

The scope of our work is the same as that of Baskaran et al. [9], parameterized multi-level tiled loop generation and their parallelization. They use the outset construction method in our work [52], but their tile space is a dense set similar to that of Goumas et al. [22]. Consequently, their tiled loop bounds have ceiling and floor operations. Another major difference is that they use a relaxed projection and symbolic Fourier Motzkin Elimination for multi-level tiled loop generation and parallelization. Also, they use an parametric integer linear programming to derive the bounds for outer most time loop in parallelized tiled loops. In contrast, we do not rely on such heavy machinery. Finally, the focus of optimization in our work is on point-loops, but their focus is on tile-loops.

Chapter 3

Parameterized Tiled Loop Generation for Perfectly Nested Loops

This chapter presents D-tiling, an approach for generating (parameterized) tiled loops dimension by dimension. The approach consists of simple *syntactic* processing that is applied *independently* and *only once* to *each* loop. For an arbitrarily polyhedral shaped iteration space, the tiled loop generation problem has been long believed to require heavy machinery like Fourier-Motzkin elimination and projection, and hence to have an exponential complexity. Avoiding any polyhedral operations, this solution makes the complexity of tiled loop generation problem linear in the number of loop bounds. Despite its simplicity, we retain several advantages of recent tiled code generation schemes—unified generation for fixed, parameterized and hybrid tiled loops, and scalability for multi-level tiled loop generation. We also provide a correctness proof of our generation algorithm. Later, this correctness proof provides a foundation of the correctness of our algorithm for generating parameterized tiled loop generation for imperfectly nested loops.

3.1 Background

The tiling transformation takes a d-depth (perfectly) nested loop and produces a loop nest of depth (at most) $2d$. The main idea is a decomposition of the iteration space into

```
for (i = 1; i <= N;i++)
  for (j = 1; j <= i;j++)
    S1(i,j);
```

Figure 3.1: Triangular iteration space: the body of the loop is represented with the macro S1 for brevity

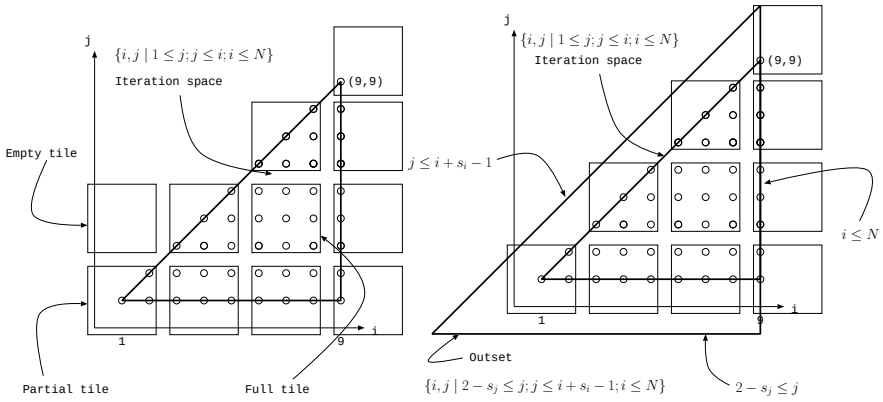

Figure 3.2: (Left) An iteration space when $N = 9$ and its 3×3 tiling, (Right) An outset of triangular iteration space when $N = 9$ (left) and its 3×3 tiling

tiles.

In the rest of this chapter we assume that rectangular tiling is legal. Also, the input program is a perfectly nested affine control loop, and tile shapes are hyper-rectangular. Affine control loops are loops where the bounds are affine functions of program parameters (loop invariant variables) and outer indices.

The iteration spaces of such loops can be expressed as a polyhedron, i.e., a set whose constraints are linear inequalities. Consider the loop nest in Figure 3.1. Its iteration space P_{ex} can be written as

$$P_{ex} = \{i, j \mid i \le N; 1 \le j \le i\}$$

Note that there are only three inequalities that define the iteration space because $1 \le i$ is redundant. A geometric representation of this iteration space is shown in Figure 3.2.

Figure 3.2 also shows a 3×3 rectangular tiling of this iteration space. Depending on the intersection of a tile and the iteration space, there are three kinds of tiles—empty

tiles whose intersection with the iteration space is empty, full tiles whose intersection is the tile itself, and partial tiles that are neither empty nor full tiles. The lexicographically smallest point of a tile is called the *tile origin*.

In general, we can express an iteration space of perfectly nested loops as

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

where $\vec{z}$ is the iteration vector of size d, Q is a $m \times d$ matrix, $\vec{q}$ is a constant m-vector, $\vec{p}$ is an n-vector containing symbolic parameters for the iteration space, and B is a constant $m \times n$ matrix.

Whenever there is no ambiguity we use z in the place of $\vec{z}$. We denote the k-th component of z as z_k and a tile size vector as s. We also denote $s - \vec{1}$ as s'. There are two important operations on polyhedra that will be used later in this chapter. The first operation is a projection of a polyhedron $\mathcal{P}$ onto first k dimensions, denoted by $Proj(\mathcal{P}, k)$.

$$Proj(\mathcal{P}, k) = \{(z_1, \ldots, z_k) \mid \exists z' \in \mathcal{P}, \forall i = 1, \ldots, k, z'_i = z_i\}$$

For example, $Proj(P_{ex}, 1)$ is $\{i \mid 1 \leq i \leq N\}$ and $Proj(P_{ex}, 2)$ is $\{i, j \mid i \leq N; 1 \leq j \leq i\}$, P_{ex} itself. Note that k is smaller than d, the number of dimensions in $\mathcal{P}$.

The other operation is an extension of $\mathcal{P}$ to k dimensions, denoted as $Ext(\mathcal{P}, k)$:

$$Ext(\mathcal{P}, k) = \{(z_1, \ldots, z_k) \mid (z_1, \ldots, z_d) \in \mathcal{P}\}$$

Here, k is greater than d.

Now, consider $Ext(Proj(P_{ex}, 1), 2)$. Since $Proj(P_{ex}, 1)$ is $\{i \mid 1 \leq i \leq N\}$, it is $\{i, j \mid 1 \leq i \leq N\}$. There are no constraints on j, so it is different from P_{ex}. More significantly, it is a superset of P_{ex}.

The essence of tiled loop generation is constructing two sets of loops—*tile-loops* that enumerate all the tiles (all the tile origins) and *point-loops* that enumerate all the points

within a tile. When tile-loops and point-loops are separately generated [22, 52, 34], the heart of the problem is to construct a tile space and generate tile-loops. The difficulty arises from the fact that not all the tile origins are in the iteration space.

3.2 Efficient Construction of Approximate Outset

In this section we present an important construction method for a superset of the outset. This technique is developed in collaboration with Lakshminarayanan Renganarayanan and Michelle Mills Strout [52].

We defined an *outset* (denoted by $outset(\mathcal{P})$) as a set containing all the full/partial tiles. An outset is precise if it does not contain any empty tile origins. The precise outset P_{out} of $\mathcal{P}$ is defined as

$$P_{out} = \{z \mid tile(z) \cap \mathcal{P} \neq \emptyset\}$$

where $tile(z)$ is a set of points in the tile whose origin is z.

We formulate an approximation of the outset that can be constructed efficiently. The construction method is based on constraint shifting. Formally, it is written as

$$\{z | Qz \geq (q + Bp) - Q^{+}s'\}$$

where

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

The term $Q^{+}s'$ can be interpreted as shift of a hyperplane. The outset of the iteration space for the example in Figure 3.1 is shown in Figure 3.2. When a hyperplane is a lower bound of the k-th dimension, it will be shifted "out" by s_k. So, $j \geq 1$ becomes $j \geq 1-(s_j-1)$ and $i-j \geq 0$ becomes $i-j \geq -(s_i-1)$. Since the constructed outset is a polyhedron, albeit parameterized by the tile sizes (d extra parameters), well established techniques and existing tools like CLooG can be used for generating code that visits all

```
// tile-loops
for (t_i = up(-s_i-s_j+3,s_i); t_i <= N; t_i+=s_i)
  for (t_j = up(-s_j+2,s_j); t_j <= t_i+s_i-1; t_j+=s_j)
    // point-loops
    for (i = MAX(1,t_i); i <= MIN(N,t_i+s_i-1); i++)
      for (j = MAX(1,t_j); j <= MIN(i,t_j+s_j-1); j++)
        S1(i,j);
```

Figure 3.3: Tiled loops from the example in Figure 3.1 from HiTLOG

the points in the outset. Then the generated loops are further processed—lower bounds are adjusted and strides are set to tile sizes—to visit only the tile origins.

Figure 3.3 shows the generated code using the open source tool *HiTLOG* [26]. The macro $up(t, s)$ in the loop lower bound gives an integer p such that $p = \lceil t/s \rceil \times s$. It adjusts the lower bounds so that the loops visit tile origins correctly.

3.3 D-tiling

In this section we first present the key intuition behind D-tiling: namely that the outsets for the projection of an iteration space are supersets of its precise outset. Then, we present an algorithm to generate parameterized tiled loops. Finally, we provide a mathematical proof of its correctness.

3.3.1 Intuition

Our main idea is that we consider a loop nest as successive projections of an iteration space on the first (outermost) few (say k) dimensions. For example, consider the iteration space P_{ex} in Figure 3.2. $Proj(P_{ex}, 1)$ is $\{i \mid 1 \leq i \leq N\}$. If we tile this with tile size of 3, the tile space $\mathcal{T}$ is $\{t_i \mid -1 \leq t_i \leq N\}$. Now, we compare $Ext(\mathcal{T}, 2)$ and the precise outset of P_{ex}. Suppose that $z \notin Ext(\mathcal{T}, 2)$ and $z \in P_{out}$. Since $z \in P_{out}$, there exists $z' \in P_{ex}$ such that $z' \in tile(z)$. But since $z \notin Ext(\mathcal{T}, 2)$, $z_1 > N$ or $z_1 < -1$. When $z_1 > N$, it implies $z'_1 > N$. The other case implies $z'_1 < 1$. This contradicts $z' \in P_{ex}$. So, $P_{out} \subseteq Ext(\mathcal{T}, 2)$. Hence, $Ext(\mathcal{T}, 2)$ is a superset of the tile space of P_{ex}.

In general, for an iteration space P_{iter} of d dimensions and for all $k = 1, \ldots, d$,

$$Outset(P_{iter}) \subseteq Ext(Outset(Proj(P_{iter}, k)), d)$$

Now, consider $Proj(P_{iter}, k)$, the projection of P_{iter} onto the first k dimensions. This information is already present in the loop nest. For instance, when a loop nest scanning P_{iter} is generated, $Proj(P_{iter}, k)$ is used for generating the loop at depth k. In this case, the iteration space of the first k loops is the same as $Proj(P_{iter}, k)$. For the case of hand-written loops, it might be different from $Proj(P_{iter}, k)$, but always a superset of $Proj(P_{iter}, k)$.

This intuition also comes from the loop generation algorithm from a polyhedron. The loop at depth k is directly derived from the projection of a d-dimensional polyhedron P into the first k dimensional space. Hence, this relation of $P \subseteq Ext(Proj(P, k), d)$ holds for any polyhedron. Furthermore, from this loop generation process we can deduce that loop nests can be viewed as successive projections and therefore all the projections are presented in original loop nests.

3.3.2 Generation of Tile-loops

Using this insight, we now present an algorithm for generating tile-loops from a perfectly nested affine control loop of depth d. In such a loop, a lower bound is either an affine expression of program parameters and outer indices or the max of finitely many such expressions. Similarly, an upper bound is either an affine expression or the min of such expressions. For the sake of simplicity in our explanation, we assume that the lower bound lb_k and upper bound ub_k of the k-th loop consist of just one affine expression. For the same reason, we also use tiling where the origin is a tile origin. Both of these are only for the sake of explanation and can be easily relaxed. When a bound has multiple affine expressions, we treat each independently and combine with the appropriate operator, max/min for lower/upper bounds. When $\vec{0}$ is not a tile origin, the generated

Input: AST - perfectly nested loops, t - a tile index name vector, s - a tile size vector

1: **for** each k-loop in AST **do**
2: $index \leftarrow t_k$
3: replace index names in lb_k/ub_k by tile index names
4: $lb_k \leftarrow up(shift\_down(lb_k)), s_k)$
5: $ub_k \leftarrow shift\_up(ub_k)$
6: $step_k \leftarrow s_k$
7: **end for**

PROCEDURE ***shift_down***

Input: af - an affine expression

1: $temp \leftarrow af$
2: **for** each outer index i in af whose coeffient c_i is negative **do**
3: $temp \leftarrow temp + c_i \times (s_i - 1)$
4: **end for**
5: $temp \leftarrow temp - (s_k - 1)$
6: return $temp$

PROCEDURE ***shift_up***

Input: af - an affine expression

1: $temp \leftarrow af$
2: **for** each outer index i in af whose coeffient c_i is positive **do**
3: $temp \leftarrow temp + c_i \times (s_i - 1)$
4: **end for**
5: return $temp$

Figure 3.4: Algorithm for generating tile-loops

loop can be easily post-processed to add an appropriate offset.

Figure 3.4 shows the algorithm for generating tile-loops. The index name and parameter names are inputs. A loop has four components: an iterator, lower and upper bound expressions, and a step size. Our algorithm consists of four rules to derive the corresponding components for tile-loops. The rules for updating index name and step size are trivial (lines 3 and 6). A lower and upper bounds of the tile-loop at depth k are derived from those of the original loop at depth k. Each bound can be written as

$$lb = \sum_{c_i<0} c_i z_i + \sum_{c_j>0} c_j z_j + \sum_h b_h p_h + c$$

where the c's are the coefficient of outer indices and b's are the coefficients of the parameters. The terms on outer indices are split into two based on signs of their coefficients.

To derive the modified lower bound, we apply the procedure called *shift_down* that adds for each negative term c_i its product with s'_i (the tile size minus 1 of that level)

and subtracts s'_k (the tile size at the given level minus 1). That is, the new bound can be written as

$$\text{shift\_down}(lb) = lb + \left(\sum_{c_i<0} c_i(s_i - 1) \right) - (s_k - 1)$$

It can also be written as

$$\sum_{c_i<0} c_i(z_i + s_i - 1) + \sum_{c_j>0} c_j z_j + \sum_h b_h p_h + c - (s_k - 1)$$

For instance, the lower bound of i loop in the figure is 1. Since there are no outer indices, we just subtract $s_i - 1$ and get $1 - (s_i - 1)$. Then, we obtain the lower bound as $\text{up}(2 - s_i, s_i)$.

To derive a upper bound, we apply the procedure called $shift\_up$ that simply adds the product of positive coefficients of the outer index term and the corresponding tile size minus one.

$$\text{shift\_up}(ub) = ub + \sum_{c_j>0} c_j(s_j - 1)$$

or

$$\sum_{c_i<0} c_i z_i + \sum_{c_j>0} c_j(z_j + s_j - 1) + \sum_h b_h p_h + c$$

The upper bound of i in Figure 3.2 is N. Since N is a parameter name, not an outer index, the upper bound of t_i remains unchanged. However, the upper bound of the j loop is i, which is an outer index. Hence, the upper bound of t_j loop becomes $t_i + (s_i - 1)$. The final generated tile loops are shown in Figure 3.6.

3.3.3 Generation of Point-loops

A point-loop is a loop nest that enumerates the iteration points within a tile. Conceptually, it visits the intersection of the iteration space and the tile with origin t. In the decomposition approach, point-loop generation is straightforward. We simply take the original loop nest and combine its lower and upper bounds with tile (box) bounds. The algorithm is given in Figure 3.5. The index name and step size are not changed.

Input: AST - perfectly nested loops, t - a tile index name vector, s - a tile size vector

1: **for** each k-loop in AST **do**
2: $lb_k \leftarrow$MAX(t_k, lb_k)
3: $ub_k \leftarrow$MIN$(t_k + s_k - 1, ub_k)$
4: **end for**

Figure 3.5: Algorithm for generating point-loops

```
// tile-loops
for (t_i = up(2-s_i,s_i); t_i <= N; t_i+=s_i)
  for (t_j = up(2-s_j,s_j); t_j <= t_i+s_i-1; t_j+=s_j)
    // point-loops
```

Figure 3.6: Tile loops obtained from the example in Figure 3.1 using dimension-by-dimension tiling

For example, in order to generate point-loops from the example in Figure 3.1, we replace the lower and upper bounds in the outermost loops with MAX$(1, t_i)$ and MIN$(N, t_i + s_i - 1)$, respectively. Similarly, the bounds of innermost point-loop are MAX$(1, t_j)$ and MIN$(i, t_j + s_j - 1)$.

3.3.4 Generation of Parameterized Tiled Loops

Tiled loops are simply a composition of tile-loops and point-loops. Again, consider the doubly nested loop in Figure 3.1 for which we just derived tile-loops and point-loops. The complete tiled loop is simply the tile loop whose body is replaced by the point-loops. The main difference between this tile-loop and the one in Figure 3.3 which was generated with the previous approach of HiTLOG is that there is no s_j in the lower bound on i. Although our lower bound is tighter, it still visits all the necessary tile origins.

Another difference is that our tile-loops visit fewer tile origins. In other words, the outset we implicitly construct is smaller than that the outset presented in Section 3.2. In this example, we do not have *any* empty tile origins. The tile-loops in Figure 3.3 scan an empty tile origin. However, our tile-loops may also visit empty tiles since the algorithm

construct an approximation of the precise outset.

3.3.5 Correctness Proof

The main idea behind our tile-loop construction is that the tile space for a projection of a polyhedron is big enough to include all the full/partial tile origins. This makes it unnecessary to use polyhedral operations of exponential complexity that have been used in all the existing techniques. We simply rely on the fact that projections onto the outer dimensions are already present in the syntactic structure of the original loop nest.

The proof is by induction on dimension d, the depth of loops. We show that a non-empty tile origin is a valid iterator in tile-loops. We denote the original iteration space as P_{iter} and the iteration space of tile-loops as T_{iter}.

Base case: $d = 1$. This is when there is only one loop. $P_{iter} = \{z \mid lb_1 \leq z_1 \leq ub_1\}$ is a line segment (parameterized by program parameters). Let y_1 be a non-empty tile origin. Then there exists $x_1 \in P_{iter}$ such that $y_1 = x_1 - \delta$ and $0 \leq \delta \leq s_1 - 1$. Now, the loop scans $lb_1 - (s_1 - 1) \leq t_1 \leq ub_1$. So, $lb_1 \leq y_1 + \delta \leq ub_1$. So, $lb_1 - \delta \leq y_1 \leq ub_1 - \delta$. So, $y_1 \leq ub_1 - \delta \leq ub_1$ and $lb_1 - (s_1 - 1) \leq lb_1 - \delta \leq y_1$. Hence, $y_1 \in T_{iter}$. When a bound is the max/min of more than one affine expression, we can apply this reasoning to each affine expression. We get $y_1 \in T_{iter}$.

Inductive step: Suppose that this tiling scheme works for loops of depth $l - 1$, and consider P_{iter}, an l-dimensional iteration polyhedron for loops of depth l. Let y be a non-empty tile origin in l-dimensional space. By the hypothesis, the outer $l - 1$ loops are correctly tiled, which means that the tile-loops of depth $l-1$ visit the point $(y_1, \ldots, y_{l-1})$. Now, unless the innermost loop prevents y from being scanned, y will be visited by the tile-loops of depth l. Since y is a non-empty tile origin, there exists $x \in P_{iter}$ such that $y + \delta = x$ and $0 \leq \delta_k \leq s_k - 1$. Notice that δ, x and y are an l-dimensional vectors here.

Since $x \in P_{iter}$ and $x_k = y_k + \delta_k$ for all k,

$$\sum_{c_i<0} c_i(y_i + \delta_i) + \sum_{c_j>0} c_j(y_j + \delta_j) + \sum_h b_h p_h + c \leq y_l + \delta_l$$

So, we can rewrite this as

$$lb_l + \sum_{c_i<0} c_i\delta_i + \sum_{c_j>0} c_j\delta_j - \delta_l \leq y_l$$

Since $\sum_{c_j>0} c_j\delta_j$ is non-negative, we drop the term and get

$$lb_l + \sum_{c_i<0} c_i\delta_i - \delta_l \leq y_l$$

Now, we add $\left(\sum_{c_i<0} c_i(s_i - 1)\right) - (s_l - 1)$ and subtract it from the left-hand side and obtain

$$\text{shift\_down}(lb_l) + \left(\sum_{c_i<0} c_i(\delta_i - (s_i - 1))\right) + (s_l - 1) - \delta_l \leq y_l$$

Since $c_i \leq 0$ and $\delta_i \leq s_i - 1$, the term is non-negative. Similarly, $(s_l - 1) - \delta_l \geq 0$. So, we finally obtain

$$\text{shift\_down}(lb_l) \leq y_l$$

Using similar algebra, we can obtain

$$y_l \leq \text{shift\_up}(ub_l)$$

Therefore, y is a valid iteration of the tile-loops we constructed, and the tile-loops visit all the non-empty tile origins.

3.4 Experimental Results

We compare the code efficiency of the loops generated by D-tiling with TLOG (or HiTLOG). When the number of tiling levels is one, the loops generated by HiTLOG are the same as TLOG. The structures of loops are very similar to each other. The loops generated by D-tiling may visit fewer empty tiles than those by TLOG. We expect similar results from both method.

	Description	Program Parameters	Exec. time - gcc/icc (sec.)
DSYRK	Symmetric Rank k Update	N=3000	142.85 / 142.14
TriMatProduct	Triangular matrix multiplication	N=3000	93.08 / 92.65
2D3D Seidel	Gauss-Seidel Style 3D stencil computation	TMAX=2000, N=1000	51.32 / 51.67

Table 3.1: Benchmarks for evaluation of D-tiling, parameterized tiled loop generation technique for perfectly nested loops. The last column shows the execution time of original (i.e., non-tiled) program with gcc and icc.

We performed efficiency evaluation on three widely used benchmarks in Table 3.1. We compiled all the code with gcc 4.4 and Intel icc 11.1 and ran all experiments for generated code efficiency evaluation on an Intel Core 2 Duo running 2.2 GHz with 2MB L2 Cache and 1GB memory. The execution time of non-tiled version for the benchmarks is also presented with the size of program parameters.

Figure 3.7, 3.8 and 3.9 show the total execution time of the three benchmarks with two tiled versions using gcc and icc. The first tiled version is generated by HiTLOG and the second one by D-tiling. Execution time is similar except when tile sizes are very small, from 2 to 4. When tile sizes are small and icc compiler is used, the code generated by TLOG performs better that by D-tiling. In the other hand, the code by D-tiling performs better than the other when the code is compiled by gcc. The two techniques deliver comparable code quality.

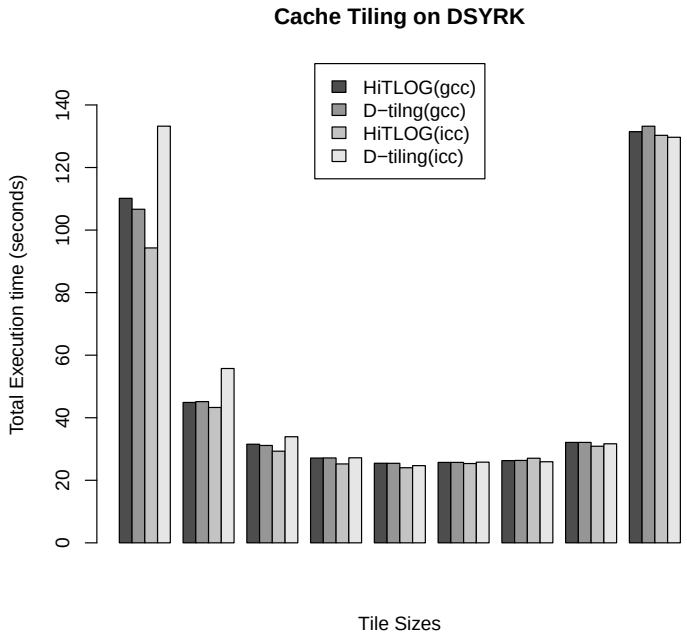

Figure 3.7: Total execution time of two tiled version of DSYRK using D-tiling and TLOG

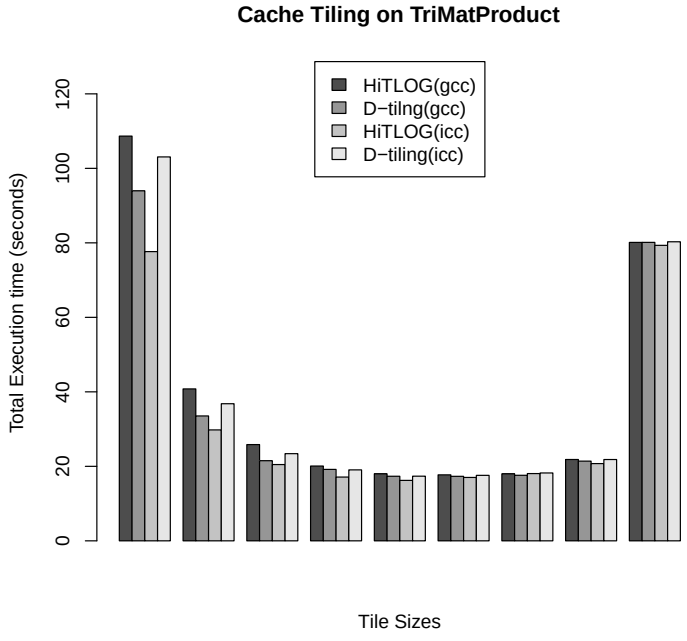

Figure 3.8: Total execution time of two tiled version of triangular matrix product using D-tiling and TLOG

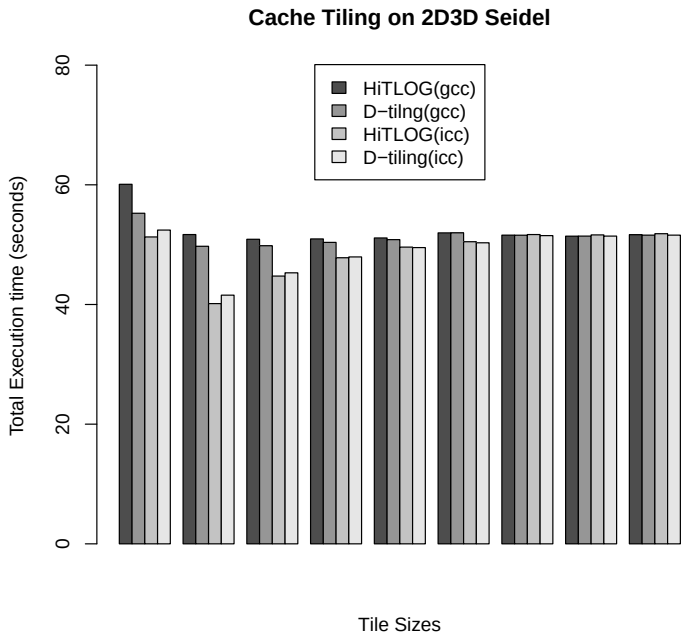

Figure 3.9: Total execution time of two tiled version of Gauss-Siedel stencil computation on 2D data using D-tiling and TLOG

Chapter 4

Parameterized Tiled Loop Generation for Imperfectly Nested Loops

In this chapter we extend the D-tiling of Chapter 3 to generate tiled loops from imperfectly nested loops. We generate parameterized tiled loops from embedded imperfectly nested loops using the combination of the direct extension of perfectly nested loop tiling and subsequent optimizations. The whole generation together with the optimizations is achieved by purely syntactic processing.

We first describe our input program specification and then provide algorithms for generating parameterized tiled loops for *embedded* imperfectly nested loops. Throughout this section, we will use, as a running example, a program for solving a lower triangular linear systems in Figure 4.1.

4.1 Input Programs

We assume that our imperfectly nested loops are *embedded*, i.e., where each assignment statement is surrounded by the *same* number of loops and these iteration spaces are pair-wise disjoint. The input is either perfectly nested loops with a sequence of affine-guarded statement blocks or embedded imperfectly nested loops . The iteration space of a guarded block must be disjoint from that of the others.

Such specifications can be obtained by embedding all the iteration spaces into a

```
for (i = 0; i < N;i++)
  for (j = 0; j < i;j++)
    x(i) -= L(i,j)*x(j);      // S1
  x(i) /= L(i,i);             // S2
```

```
for (i = 0; i < N;i++)
  for (j = 0; j <= i;j++)
    if (j<i)
      x(i) -= L(i,j)*x(j);
    if (j==i)
      x(i) /= L(i,i);
```

```
for (i = 0; i < N;i++)
  for (j = 0; j < i;j++)
    x(i) -= L(i,j)*x(j);
  for (j = i; j<=i; j++ )
    x(i) /= L(i,i);
```

Figure 4.1: A solver for lower triangular linear systems written in these styles: a simple imperfectly nested loop, a perfectly nested loop with affine-guards, and an *embedded* imperfectly nested loop

common space. Figure 4.2 shows the iteration spaces of our example. Notice that the two embedded loops in Figure 4.1 have the same iteration space. Embedded imperfectly nested loops can be obtained by generating loops from a union of polyhedra in a common space. In such loop nests, loops at the same depth have distinct loop counter ranges. For example, the ranges of the two loops in Figure 4.1, $\{j \mid 0 \leq j < i\}$ and $\{j \mid j = i\}$, are disjoint. Also, the values taken by loop counters at the same depth is in the increasing order. In our context we expect to apply the tiling to the code automatically generated by a tool like CLooG [10] or Omega [47]. Hence this requirement, *embedded* imperfectly nested loops, is not too restrictive. Other tiled loop generation tools based on the polyhedral model such as PrimeTile [24] makes the same assumptions.

In the following discussion, we assume that a statement appears only once in the input program. If the same textual statement appears more than once, we treat them as different statements. If more than one textual statements appear in the same loop, we treat them as one statement. So, the iteration spaces of statements are disjoint from each other.

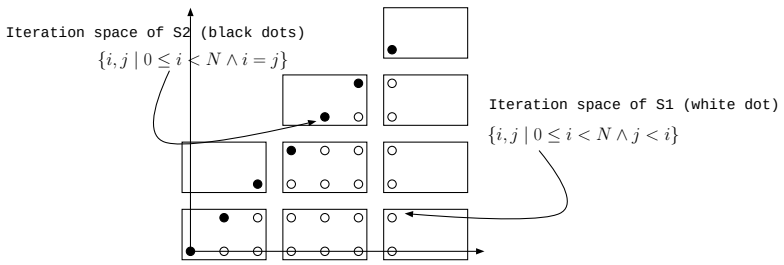

Figure 4.2: Iteration space of the solver for lower triangular linear systems

4.2 Tile-loop Generation

Generation of tile-loops for input programs with guards is the same as that for perfectly nested loops. We apply D-tiling to each loop and obtain tile-loops. That is, we treat the sequence of guarded statements as a single statement. Although treating the whole sequence as a single statement in *point-loops* might cause inefficiency, the overhead in tile-loops is not significant because the iteration spaces of statements are most likely to be "close together", especially given the fact that input loop programs are embedded into a common space. For example, the iteration space of the first statement in our example is not and cannot be far away from that of the second statement. Even when such separation is allowed, it is unlikely that programmers will write such programs; and automatic approaches for embedding based on linear programming will usually bring all the iteration spaces "close" to each other.

For embedded input programs, we take at each loop depth the lower bound of the first loop and the upper bound of the last loop. Figure 4.3 shows a simple algorithm to derive perfectly nested loops from embedded imperfectly nested loops for the tile loop generation. Note that loop counters at the same depth are in increasing order. The iteration space of the derived perfectly nested loops contains the iteration space of each statement. Then we apply D-tiling to the perfectly nested loops.

Now, we show how to apply this algorithm to our running example. Consider the last loop nest in Figure 4.1. There is one loop at depth 1, and its lower and upper bounds

Input: AST - an embedded loop nest
1: **for** each depth k in AST **do**
2: $lb_k \leftarrow$ lower bound of the first loop at depth k
3: $ub_k \leftarrow$ upper bound of the last loop at depth k
4: **end for**

Figure 4.3: Algorithm to derive perfectly nested loops from embedded imperfectly nested loops

are 0 and $N-1$. Since there is only one loop, taking the lower bound of the first loop and the upper bound of the last loop is the same as taking the loop itself. There are two loops at depth 2. By taking the lower bound of the first loop and the upper bound of the last loops, we obtain a loop whose lower and upper bounds are 0 and i respectively. This gives us a doubly nested loop, to which we apply D-tiling and obtain tile-loops.

4.3 Point-loop Generation

The basic idea for generating point-loops remains the same. In the original loop program, replace every lower bound lb_i that occurs at level i by $max(lb_i, T_i)$ and every upper bound ub_i by $min(ub_i, T_i + S_i - 1)$. The loops visit the intersection of the original iteration points and points within a given tile. For example, consider the statement $S2$ in our running example. Its iteration points are $\{i, j \mid 0 \leq i < N \wedge j = i\}$. The corresponding loop is the last loop nest in Figure 4.1, ignoring the first j loop. So, we obtain loops where the outer loop counter i iterates from $max(T_i, 0)$ to $min(T_i + S_i - 1, N)$, and the inner loop counter j from $max(T_j, i)$ to $min(T_j + S_j - 1, i)$. Similarly, we can obtain the point-loops for the first statement. Since the common loop of these two statements are identical, so are the common loops of the derived point-loops. Therefore, we can merge these two loops textually and finally obtain the loop for both statements shown in Figure 4.4 (above).

For input programs with guards, we can treat the whole sequence of guarded blocks as a single statement. In other words, the point-loops are derived from the perfectly

```
for (i = max(T_i,0); i < min(T_i+S_i,N); i++)
  for (j = max(T_j,0); j < min(T_j+S_j,i); j++)
    x(i) -= L(i,j)*x(j);
  for (j = max(T_j,i); j<=min(T_j+S_j-1,i); j++ )
    x(i) /= L(i,i);
```

```
for (i = max(T_i,0); i < min(T_i+S_i,N); i++)
  for (j = max(T_j,0); j<=min(T_j+S_j-1,i); j++ )
    if (j<i)
      x(i) -= L(i,j)*x(j);
    if (j==i)
      x(i) /= L(i,i);
```

Figure 4.4: Two point-loops for the solver for lower triangular linear system: from embedded imperfectly nested loops and from perfectly nested with affine guards

nested loop rather than the iteration space of each block. By this process, we will obtain the point-loops in Figure 4.4 (below). One may convert the condition for each guard into loop bounds to derive the other kind of point-loops. For example, we can convert $j = i$ into $j \geq i$ and $j \leq i$. We can obtain the same point-loops as that in Figure 4.4 (upper).

4.4 Parameterized Tiled Loop Generation

The final parameterized tiled loops are a simple insertion of the point-loops as the body of the tile-loops. The approach that we take in Section 4.2 and 4.3 is the most direct extension of tiled loop generation for perfectly nested loops to imperfectly nested loops.

The tiled loop nests produced by this approach have been believed to be inefficient. Nevertheless, there are a several reasons for choosing this as a starting point. First, with parameterized tile sizes and multiple iteration spaces, there is a risk of code size explosion.

When tile sizes are fixed, generated code inefficiency is resolved by splitting tile-loops into many different regions based on the sets of statements being executed, either through standard loop generation algorithms or techniques for eliminating guards. Where to split is statically known for fixed tile sizes.

For parametric tile sizes, this separation is much more complex than that for fixed tile sizes. In this situation it is possible for all the statements to be executed in a single tile for some values of tile sizes, while in extreme cases a tile may have only a statement. Furthermore, all the combinations of statements may be executed in a tile. Splitting all the cases might cause exponential growth in terms of the generation time and code size. Second, imperfectly nested tile loops are complicated to transform. Even for perfectly nested parameterized tile loops, a simple transformation like skewing for parallel execution of tiles is an open problem that has not yet been solved. Third, all the computations are tiled unlike in PrimeTime [24]. This makes the approach suitable to parallelization for machines where computing units have limited resources, like STI Cell architecture.

4.4.1 Correctness

The correctness of this direct approach is straightforward. It follows directly from the correctness of the algorithm for perfectly nested loops presented in Chapter 3. The correctness of each optimization technique to be presented later comes from the properties of the sets being used, and those properties are formally proved in Chapter 6.

4.5 Optimizations

Now we propose three optimization techniques on the generated parameterized tiled loops. At the heart of these, as well as most techniques for optimizing tiled loop nests is promotion of conditions for distinguishing the combination of statements *from point-loops to tile-loops*. The first optimization provides a condition for a particular set of statements to be executed. The second optimization exploits a condition for a tile to be full for given a statement. The last provides a guideline for which sets of statements are separated from the others to effectively improve the performance. The goal is constructing specialized point-loops and safely executing them to improve performance.

4.5.1 Removing Statements from Point-Loops

Consider our running example. There are two statements. For most tiles, only the first statement will be executed. We want to execute such tiles without checking for the second statement. For this we need to know which tiles execute only the first statement and construct point-loops for first statement only. The second task, namely constructing the point-loops, is straightforward, so we focus here on the condition for such point-loops to be executed. The condition is nothing but the condition for the second statement to not be executed. We can extract the condition for the second statement to be executed from *tile-loops* of the second statement when tiling is applied *only to it*. Then we use negation of this condition for an appropriate guard for the specialized point-loops.

To illustrate this, let us apply D-tiling to the original loop after the first statement is removed. Since the first statement is removed, loops are perfectly nested and we can directly apply D-tiling. Then, we get $T_i \geq -S_i + 1$ and $T_i \leq N$ at the first depth and $T_j \geq T_i - S_j + 1$ and $T_j \leq T_i + S_i - 1$ at the second as tile-loop bounds. Here, we do not adjust the lower bounds to be aligned with a tile origin because these will not be used for loop bounds. These bounds can be viewed as a condition for a tile to include an iteration point of the second statement. Hence, if a tile origin satisfies these inequalities, the tile *may* have include an iteration of the second statement, otherwise it does not contain *any* iterations of the second statement. We can also simplify this condition to $T_j \geq T_i - S_j + 1$. An inequality can be omitted if it appear the tile-loops generated in Section 4.2. The point-loop for only the first statements can be safely placed in the guard whose condition is $T_j < T_i - S_j + 1$.

In general, we construct point-loops that execute only a given subset of statements from the general point-loops, and derive the condition for such point-loops. The condition is negation of the condition for the remaining statements to be executed. The condition for a statement to be executed is extracted from the tile-loops by applying

tiling to the statement independently, i.e., by first removing all the statements except one being considered as well as any unnecessary loops for the given statement.

4.5.2 Splitting Full Tiles

Like the first optimization, this optimization separates a set of tiles from the other tiles. We define an inset of an iteration space to be a set containing all of full tiles origins but none of empty and partial tiles. A construction algorithm and the correctness proof is presented in Chapter 6. We may further specialize point-loops by identifying the inset of a statement. In other words, we construct a guard whose condition is the inset condition and whose body is box loops containing the given statement. Note that iteration spaces of statements are disjoint from each other. A tile that belongs to the inset of a statement does not contain any other statements. Therefore, the other statements need not to be checked.

4.5.3 Selecting Iteration Space

The previous techniques provides a mechanism to distinguish particular tiles from others to specialize point-loops. However, for a program with l statements, there are $2^l - 1$ combinations of statements. The optimizations themselves do not provide information about which subset of statements leads to performance improvement. What we propose here is a guideline for selecting a set of statements and splitting full tiles for statements. First, all statements that have *equalities* in their surrounding loops will be excluded, and only the other statements are candidates to be selected. Tiles for these statements are unlikely to be executed many times and be full tiles. For our running example, we will generate an additional point loop for only the first statement, but not for the second statement only.

```
for (Ti=shift_up(-1*Si+1,Si) ; Ti<N ; Ti+=Si ) {
  for (Tj=shift_up(-1*Sj+1,Sj) ; Tj<=Si+Ti-1 ; Tj+=Sj ) {
    // For full tiles of [S1].
    if (Ti>=0 && Tj>=0 && Ti<=-1*Si+N && Tj<=-1*Sj+Ti) {
      // Box-loops for [S1]
      for (i=Ti ; i<Si+Ti ; i+=1 ) {
        for (j=Tj ; j<Sj+Tj ; j+=1 ) {
          S1(i,j);
        }
      }
    // For tiles that do not belong to the outset of [S2]
    } else if (!(Tj>=-1*Sj+Ti+1) ) {
      // Point-loops for [S1] only}
      for (i=max(0,Ti) ; i<min(N,Si+Ti) ; i+=1 ) {
        for (j=max(0,Tj) ; j<min(i,Sj+Tj) ; j+=1 ) {
          S1(i,j);
        }
    }
    // For any tiles including empty tiles
    } else {
      // Point-loops for [S1] and [S2]
      ...
    }
  }
}
```

Figure 4.5: Final structure of parameterized tiled loops from triangular solver; after optimization. For the general point-loops for both statements, see Figure 4.4

4.6 Structure of Parameterized Tiled Loops

The structure of a generated tiled loop nest consists of (i) perfectly nested tile-loops, and (ii) a sequence of if/else if/else statements where the final else clause has the most general point-loops from Section 4.3. The other conditional clauses are specialized point-loops guarded by appropriate conditions. The final structure of parameterized tiled loops for our running example is presented in Figure 4.5.

4.7 Experimental Results

We implemented the technique described in this chapter and evaluated the efficiency of our approach on four common benchmarks. We compared our technique with that of PrimeTile on generated code efficiency.

4.7.1 Experimental Setup

The four benchmarks are listed in Table 4.1. We compiled all the code with gcc 4.4 and Intel icc 11.1 and ran all experiments for generated code efficiency evaluation on an Intel Core 2 Duo running 2.2 GHz with 2MB L2 Cache and 1GB memory. The execution time of the non-tiled version for the benchmarks is also presented together with the size of program parameters.

We generated six different versions of parameterized tiled loops. First, we generated a version of using the direct extension of parameterized tiled loop generation without any optimizations. There are two kinds of point-loops: (i) perfectly nested loops with a sequence of guards and (ii) an embedded imperfectly loop nest. The first case is labelled as *naive* and the second *naive-e*. Then we construct specialized point-loops for the statements that do not have equality constraints. When this optimization is applied to *naive*, it becomes *opt*. Similarly, we have *opt-e* version. Then we further optimize this optimized code by splitting full tiles from the other tiles. They are represented by *split* and *split-e*. Finally, we have six versions: *naive*, *naive-e*, *opt*, *opt-e*, *split*, and *split-e*. In order to provide a comparison with a existing solution, we also generated parameterized tiled loops using PrimeTile [24].

4.7.2 Results

There are four charts that show the total execution time for each benchmark. The x-axis always represents tile sizes. For each tile size, there are 14 data points: 7 points using gcc and 7 points using icc. Among each set of seven data points, the first three pairs show the impact of using two different point-loops. For example, the first two data points show the execution time of naive extension with two different point-loops. The first odd/even data points (like 1st, 3rd, and 5th points) show the impact of optimizations. In the tiled loops from PrimeTile, full tiles are split from the other computation together

	Description	loop depth/ statements	Program Parameters	Exec. time - gcc/icc (sec.)
MultiTriSolver	Multiple triangular linear systems solver	$3/2$	M=2000,N=1000	6.73 / 6.79
LU	LU decomposition without pivoting	$3/2$	N=2000	12.16 / 12.42
Cholesky	Cholesky decomposition	$3/2$	N=2000	10.01 / 9.97
FDTD	Finite difference time domain on 2D data	$3/4(6)*$	TMAX=512, NX,NY=2000	146.34 / 130.96

Table 4.1: Benchmarks for evaluation of parameterized tiled loop generation technique for imperfectly nested loops. $*$There are four distinct statements in FDTD benchmark. Their iteration space can be divided into six disjoint regions. In our generation, the number of those regions is treated as the number of statements. The last column shows the execution time of original (i.e., non-tiled) program with gcc and icc.

with optimized point-loops. So, *split-e* is the most similar to *prime* in characteristics although the computation other than that captured by full tiles in *prime* may not be tiled at all.

Figures 4.6, 4.7 and 4.8 show the total execution time for cache tiling on three matrix operation kernels. For very small tile sizes (2 to 4), *prime* performs better than our proposed method. For bigger tiles (64 to 512), our optimized code performs better. For intermediate tile sizes, the difference is almost negligible. This is the general trend for the first three kernels which have few statements.

Not all optimizations lead to performance improvement. The first optimization always improves performance. However, the split optimization does not reduce the execution time in all the cases. Sometimes it even significantly decreases the running time, but for some cases like LU compiled by icc with tile size $8 \times 8 \times 8$ it also increases the execution time. Also, embedded imperfectly nested point-loops do not always outperform point-loops with guards.

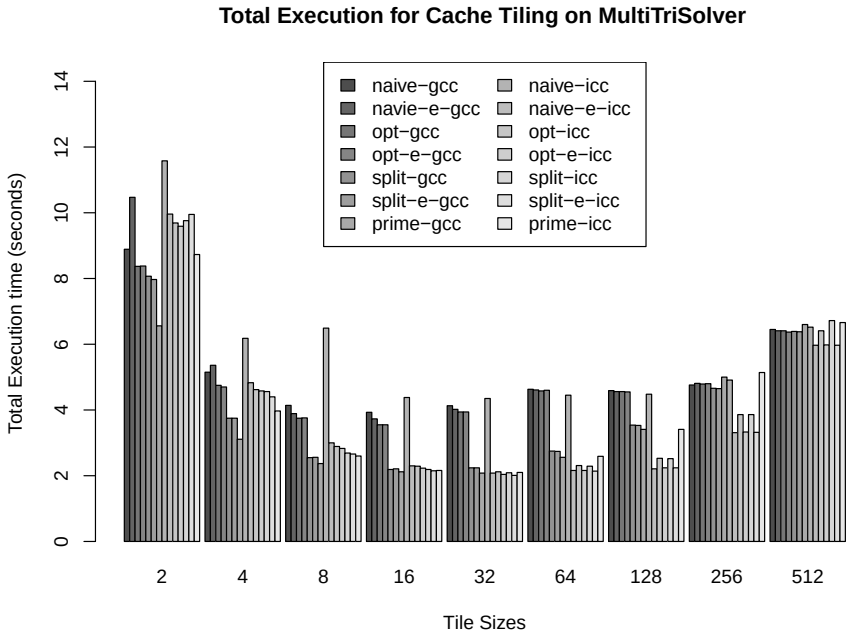

Figure 4.6: Total execution time of cache tiling on MultiTriSolver with cubic tile size

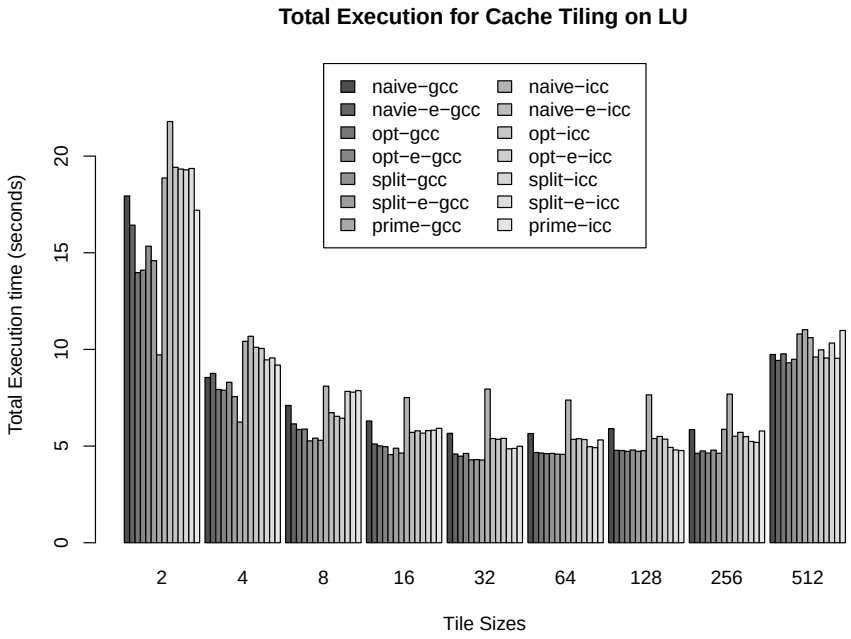

Figure 4.7: Total execution time of cache tiling on LU with cubic tile size

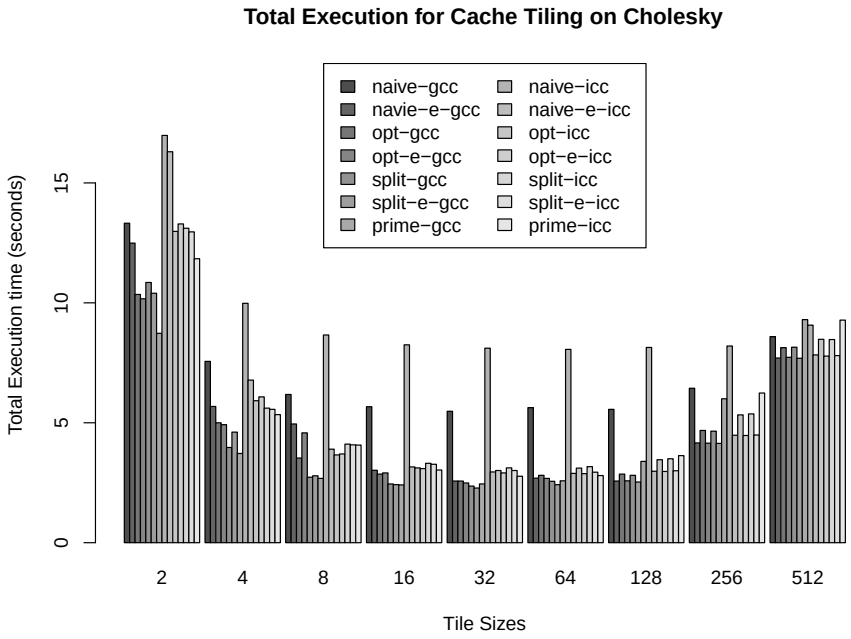

Figure 4.8: Total execution time of cache tiling on Cholesky with cubic tile size

Figure 4.9 shows the total execution time of FDTD kernel for various tile sizes. Unlike the other kernels, non-cubic tile sizes are used. The data from this kernel is a quite different from the others. The main difference in program characteristics is that there are six statements. Note that the other kernels have at most three statements. Although there are only four different textual statements, the four iteration spaces of these statements in the original program must be decomposed into six disjoint regions for generating parameterized tiled loops. So, in terms of our input program specification, there are six statements. Our optimized code outperforms that from PrimeTile even for minimum tile sizes $2 \times 2 \times 2$.

In summary, our technique generates code with performance comparable to the existing solution for the kernels when tile sizes is not very small (for cache tiling) for smaller kernels. For kernels with more statements, the code generated by our method outperforms the existing solution. Also, we were able to generate an efficient parame-

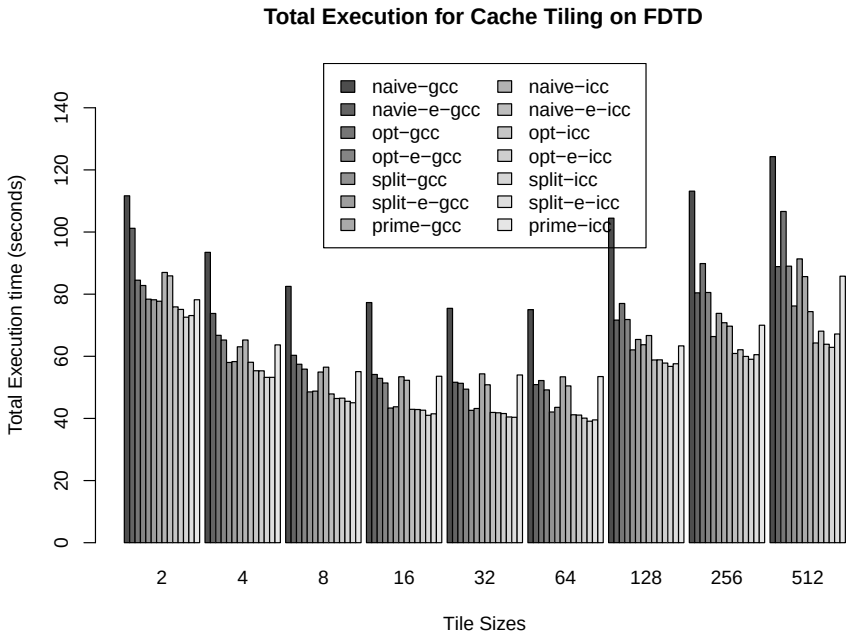

Figure 4.9: Total execution time of cache tiling on FDTD with tile size along time dimension fixed as 4

terized tiled loops from a simple loops with guards.

4.8 On the Embedding Process

One goal of embedding is to make tiling legal. A well known condition for tiling legality is full permutability of loops. Any techniques that convert imperfectly nested loops to fully permutable loops can be used as an embedding technique [41]. In the formulation based on linear algebra, the condition that is equivalent to fully permutability of loops becomes that all dependences are component-wise non-negative. This makes scheduling techniques applicable and leads to the formulation of tiling hyperplanes in Pluto [14]. Both cases (implicitly) assume that the iteration space of each statement is required to have the same number of dimensions. In other words, each statement is surrounded by the same number of loops. Existing embedding processes [2, 41, 14] uses heavy

machinery, such as projection and integer linear programming solvers. On the other hand, simple code sinking [3] works well as an embedding process for some cases. For example, the code sinking serves as embedding for the imperfectly nested loops in Figure 4.1 so that tiling is legal.

In this section we provide the precise condition when code sinking works and present an insight that may lead to development of simpler embedding process.

4.8.1 Dependence Abstraction

Since the loops are not perfectly nested, we use a more general dependence abstraction, rather than just affine functions. So, a dependence for imperfectly nested loops is a function from a pair consisting of a statement and its iteration point to another pair. We denote the depth of common surrounding loops of two statements S and T as $ncl(S, T)$.

Consider our example in Figure 4.1. There are three dependences as can be determined with an exact data-flow analysis [19].

$$\begin{aligned} dep_1: \quad & (S1,(i,j)) \rightarrow \quad (S1,(i,j-1)) \\ dep_2: \quad & (S1,(i,j)) \rightarrow \quad (S2,(j)) \\ dep_3: \quad & (S2,(i)) \rightarrow \quad (S1,(i,i-1)) \end{aligned}$$

and $ncl(S1, S2)$ is 1.

In our presentation we focus on flow dependence. However, our reasoning can naturally extended to memory-based dependence by assuming all dependences are flow dependence.

4.8.2 Embedding of Two Fully Permutable Loops

We here focus on the case where there are two statements and each statement is surrounded by fully permutable loops. In other words, any self dependence is component-wise non-negative. With this assumption, each statement can be tiled if we remove the

other statement. For this case, the dependence can be expressed as just an affine function. For example, if we remove $S2$ in our example in Figure 4.1, the remaining loop is perfectly nested and tilable. Similarly, $S2$ itself can be tiled with the same assumption.

The focus is how each iteration space is "aligned" into a common space. The key intuition is embedding along dependences. We first illustrate this idea on our running example. There are two cross-statement dependences in our example. First we take the last dependence dep_3. We embed the i iteration of $S2$ into $(i, i-1)$. We directly take the dependence function as an embedding function. Since $S1$ textually precedes $S2$, we can directly use $(i, i-1)$. Otherwise, we need to use (i, i), instead of $(i, i-1)$ by adding $(0, 1)$, the lexicographically minimum positive unit-vector. Now, let us take a look at the other dependence dep_2. Now, we replace (j) by $(j, j-1)$ according to our embedding function. Then, the dependence becomes $S1, (i, j) \rightarrow S2, (j, j-1)$. By this, we know that the second component of the dependence is non-negative. Furthermore, it also says that in order for tiling to be legal, i must be greater than or equal to j in the iteration space of $S1$. Since we can deduce this requirement from the iteration space of $S1$, tiling is legal after this embedding. We call this embedding along dependence the *natural embedding*.

Now, we apply this key intuition to the general case of two statements where each statement by itself can be tiled. Assume that there are two statements, $S1$ and $S2$. Without loss of generality, let us assume that the depth of $S1$ is no less than that of $S2$. The set of dependences from $S2$ to $S1$ is denoted as

$$\{dep_k : (S2, (z)) \rightarrow (S1, f_k(z))\}$$

where f is an affine function of z.

Then, the embedding function is the addition of the component-wise maximum of $f_k(z)$ over all k and the lexicographically minimum unit-vector in the space in which the iteration space of $S1$ lies.

```
for (i = 1; i < N;i++)
  x(i-1) /= L(i-1,i-1);
  for (j = 0; j < i;j++)
    x(i) -= L(i,j)*x(j);
x(N-1) /= L(N-1,N-1);
```

Figure 4.10: A solver for lower triangular linear systems for which code sinking does not work

On the other hand, all dependences from $S1$ to $S2$ are used to verify whether this embedding is valid. After applying the embedding function, we can verify the non-negativity of each element in the dependence. Some condition may be deduced from the iteration space of $S1$. This validation is required to ensure all the dependences to be respected. Note that not all loops are tilable. This validation may fail.

4.8.3 Discussion

The reason why simple code sinking works well is because of the fact that code sinking realizes this natural embedding. However, although it works well for some cases, its applicability is highly dependent on the structure of original loops. For example, the program shown in Figure 4.10 is the same computation as our running example, but code sinking does not work for this loop program. However, we get the same natural embedding as that from our running example. Note that the embedding itself does not change as long as dependences do not change. This natural embedding works regardless of how the loops are written. It is because although there are many ways to write loops for the same computation, dependence relation remains same.

This intuition and technique has emerged from our work [33] where we showed how some limited class of imperfectly nested loops can be tiled without embedding. The primary focus of the work is still tiled loop generation, not the legality condition nor embedding process. In order for this intuition to be useful in the context of optimizing compilers, it must be extended to handle self dependences, more than two statements and memory based dependence.

Another important extension of the natural embedding presented is scheduling. The natural embedding minimizes the distance between production and consumption while trying to satisfy tiling legality. If we take lexicographically maximum instead of component-wise maximum, we may be able to obtain a schedule of a statement while minimizing the total execution time and the distance between production and consumption. Since the lexicographically maximum represent the scheduling constraints.

Chapter 5

Multi-level Tiled Loop Generation: Successive Tiling

In this chapter we propose an approach for generating *parameterized multi-level* tiled loops from perfectly nested loops. Multi-level tiled loops are loop nests where tiling is applied more than once. When tiling is applied again, we choose tiles in the previous level as its original iteration space. We refer this approach as successive tiling. This leads to a simple and efficient technique that provides multi-level tiled loop generation at no more cost than that for *single level* tiled loop generation. We also propose a method for separating full tiles for the other tiles at any levels. We have implemented code generation techniques and the tool is available as an open source software [26].

The work presented in this chapter was done in collaboration with Lakshminaraya Renganarayana, Dave Rostron and Michelle Mills Strout, and it was presented in [34].

5.1 Successive Tiling

Tiling rectangular iteration space with hyper-rectangular tile shape is trivial and does not require any polyhedral operations that have exponential complexity. Treating tiles in the previous level as the entire iteration space for multi-level tiling allows us to exploit this simple technique for tiling all the levels, except the first level of tiling. In the first level, there are "tiles" to exploit in the previous level.

Our inputs are perfectly nested loops of depth d and their iteration space P_{iter} is represented by

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

where $\vec{z}$ is a loop iterator vector of size d, Q is a $m \times d$ matrix, $\vec{q}$ is a constant vector of size m, B is a $m \times n$ matrix, and $\vec{p}$ is a program size parameter vector of size n.

We first extend the classical formulation of (one-level) tiling for fixed tile sizes to multiple levels. With standard loop generation techniques, we may obtain highly optimized loop nest based on this formulation. However, the number of dimensions (or loop depth) increases linearly with the level of tiling, and the loop generation time grows exponentially with respect to the loop depth. Then we present our method using outset formulation based on our key observation. Our technique not only provides scalability with respect to levels of tiling, but also enables tile sizes to be symbolic parameters.

5.1.1 Multi-level Tiling for Fixed Tile Sizes

We start with the classic definition of single-level rectangular tiling [57]. 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 [32] and CLooG [10] 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 vector $\vec{s_1}, \ldots, \vec{s}_m$, a multi-level tiling can be described in a similar way.

$$\begin{aligned} P^m_{tiled} \;=\; & \{(\vec{t_1}, \ldots, \vec{t_m}, \vec{z}) \mid \forall i = 1, \ldots, m-1 : \\ & \vec{s_i} \circ \vec{t_i} \leq \vec{t_{i+1}} - \vec{o_{i+1}} \leq \vec{s_i} \circ \vec{t_i} + \vec{s_i} - \vec{1}, \\ & \vec{s_m} \circ \vec{t_m} \leq \vec{z} - \vec{o_m} \leq \vec{s_m} \circ \vec{t_m} + \vec{s_m} - \vec{1}, \vec{z} \in P_{iter}\} \end{aligned} \tag{5.1}$$

where each o_i is an offset at the i-th 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 [57], to the best of our knowledge, this is the first formalization and presentation of it—other formulations [31] 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.

5.1.2 Multi-level tiling using the outset

Another view of tiled loop generation is based on the outset formulation, 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 5.1. We start with the first level of tiling of the iteration space and each one of the first level tiles is further tiled to achieve the second level of tiling. In Figure 5.1, 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.

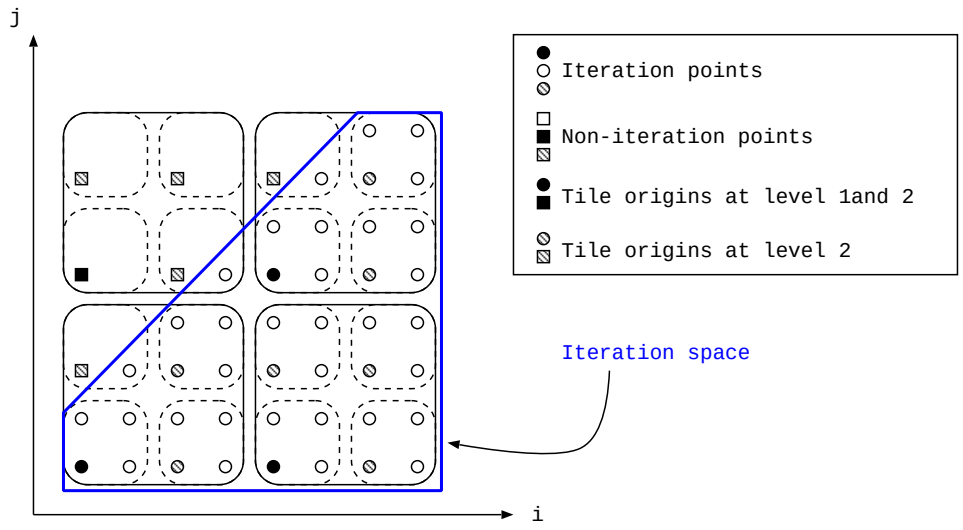

Figure 5.1: Multi-level tiling as repeatedly tiling each tile on a triangular iteration space

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 5.1. 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 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 5.2. 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 5.3. 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 as *box-tile-loops*. At the innermost level, we have the point-loops which have both the tile bounds and the iteration space bounds. We expect that 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, and is discussed in Section 5.5.2.

Multi-level tiling based on outset can be formalized as follows. Given an iteration

```
// 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 5.2: A loop nest corresponding to the multi-level tiling in Figure 5.1

space P_{iter} and a list of tile size vector $\vec{s_1}, \ldots, \vec{s_m}$, the tiled iteration space can be expressed as follows:

$$\begin{aligned}
\mathcal{P}^m_{tiled} \;=\; & \{(\vec{t_1}, \ldots, \vec{t_m}, \vec{z}) \mid \forall i = 2, \ldots, m \;: \\
& \vec{t_1} \in P_{out} \cap Lattice(\vec{s_1}, \vec{o}), \\
& \vec{z} \in P_{iter} \cap tile(\vec{t_1}, \vec{s_1}) \cap \ldots \cap tile(\vec{t_m}, \vec{s_m}), \\
& \vec{t_i} \in tile(\vec{t_1}, \vec{s_1}) \cap \ldots \cap tile(\vec{t_{i-1}}, \vec{s_{i-1}}) \;\cap \\
& \qquad Lattice(\vec{s_i}, \vec{t_{i-1}})\}
\end{aligned} \tag{5.2}$$

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 $\vec{s_i}$ is a vector of either symbolic tile size parameters, constants, or 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 super-set of P_{iter}. Further, by including the constraints of P_{iter} in the formulation, we guarantee that only valid iteration points 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

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

Figure 5.3: Structure of multi-level tiled loops generated with the outset method when partial and full tiles are not separated.

$i = 1, \ldots m-1$. The constraints of the tiled iteration space in (5.2) for this case can be simplified to:

$$\begin{aligned} \mathcal{P}^m_{tiled} \;=\; & \{(\vec{t_1}, \ldots, \vec{t_m}, \vec{z}) \mid \forall i = 2, \ldots, m \;: \\ & \vec{t_1} \in P_{out} \cap Lattice(\vec{s_1}, \vec{o}), \vec{z} \in P_{iter} \cap tile(\vec{t_m}, \vec{s_m}), \\ & \vec{t_i} \in tile(\vec{t_{i-1}}, \vec{s_{i-1}}) \cap Lattice(\vec{s_i}, \vec{t_{i-1}})\} \end{aligned} \tag{5.3}$$

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.

5.2 Separating partial & full tiles

One possible source of loop overhead occurs within the loop bounds for each tile, which contain 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 [6] suggest that tiled code may be optimized by generating different code for full tiles versus partial tiles. Jiménez et al. [30] 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. Goumas et al. [22] indicates that they differentiate between full and partial tiles, but details are not provided. Since distinguishing between full and partial tiles is important for register tiling and possibly hierarchical tiling, we present two possible approaches

for separating full tiles. 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.

Distinguishing between full and partial tiles is applicable to any of the tiled code generation techniques discussed in this dissertation. The inset can be computed as quickly as the outset. 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/empty tiles.

5.2.1 Algorithm for Computing Inset

As in Section 5.1, 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 $\vec{z}$ 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 i-th 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 $S' = diag(\vec{s} - \vec{1})$. 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,

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

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

Now, we prove that $\widehat{P_{in}} = P_{in}$.

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

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

Note that the above inequality holds 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:

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

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

5.2.2 Distinguish full and partial tiles using Inset

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 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 tile origin 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. 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 another way to enable a full versus partial tile optimization. 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.

For register tiling, it would seem that checking each tile to determine if it is full introduces higher overhead. However, splitting can result in significant blowup in code size, which can cause instruction cache problems.

5.3 Splitting Partial & Full Tiles

As discussed earlier, separation of partial and full tiles has several applications. In this section, we discuss how the inset 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

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

outer-tiles in Figure 5.1. 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 5.4. 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 5.4.

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

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

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

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

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.

5.4 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 1 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 consist 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, as shown in lines 6-7 we compute the inset of iteration space with respect to the tile sizes and indices of the desired split level (here, first). The bounds of the inset are shown below.

$$\begin{aligned} P_{in} \quad = \quad & \{(t_k, t_i) \mid 1 \leq t_k;\ t_k + s_k - 1 \leq N_k; \\ & t_k + s_k \leq t_i; t_i + s_i - 1 \leq t_k + N_i\} \end{aligned} \tag{5.4}$$

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 5.5. At line 9 we see that the guard is a direct translation from the inset in (5.4).

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 already generated the 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 5.5. Finally we compose these outermost tile-loops to obtain the complete tiled loop nest with separation of partial and full tiles.

```
1  // Outermost tile loops that scan the outset
2  T1kLB = -S1k+2; T1kLB = LB_SHIFT(T1kLB,S1k);
3  for (T1k = T1kLB; T1k <= Nk; T1k += S1k) {
4    T1iLB = T1k-S1i+2; T1iLB = LB_SHIFT(T1iLB,S1i);
5    for (T1i = T1iLB; T1i <= T1k+Ni+S1k-1; T1i += S1i) {
6      // Is (T1k,T1i) a full tile at level 1?
7      if ( T1k-1 >= 0 \&\& -T1k+Nk-S1k+1 >= 0 \&\&
8        -T1k+T1i-S1k >= 0 \&\& T1k-T1i+Ni-S1i+1 >= 0 ){
9       // Box-loops scanning origins of level 2 tiles.
10      for (T2k = T1k ; T2k<=T1k+S1k-1 ; T2k += S2k )
11       for (T2i = T1i ; T2i<=T1i+S1i-1 ; T2i += S2i )
12        // Box-loops scanning points in level 2 tiles.
13        for (k = T2k ; k<=T2k+S2k-1 ; k++ )
14         for (i = T2i ; i<=T2i+S2i-1 ; i++ )
15               S1 ;
16     } else { // (T1k,T1i) is a partial tile at level 1
17     // Point loops scanning partial tiles at 1st level.
18     for (k= max(T1k,1);k<=min(T1k+S1k-1,Nk);k++)
19      for (i= max(T1i,k+1);i<=min(T1i+S1i-1,k+Ni);i++)
20           S1 ;
21     }
22   }
23 }
```

Figure 5.5: A multi-level tiled loop for the 2D Stencil. The body of the loop is by S1.

5.4.1 Complexity & scalability of the algorithm

Let us first consider the case where no full vs. partial tile separation is performed. At the high level, the key steps are computing the outset to generate the outermost tile-loops and constructing all 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* and 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 5.5.1 confirm this, and 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 levels in multi-level tiling. For an original loop nest of depth d, the number of dimensions and constraints increase by d and $2d$, respectively, as the level of tiling increase (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 5.5.1 show how this exponential growth with respect to number of levels renders the technique inapplicable beyond two levels of tiling. In the multi-level tiled loop generation method proposed by Jiménez et al. [31] its generation time grows linearly with the number of levels. In other words, the complexity of their method is exponential for a single level, and it spends the same cost for each level.

5.5 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 5.1.1. The second generator is capable of generating tiled code with tile sizes that are fixed or parameterized or mixed and is based on the method discussed in Section 5.1.2. 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 [10] loop generator is used internally to generate the

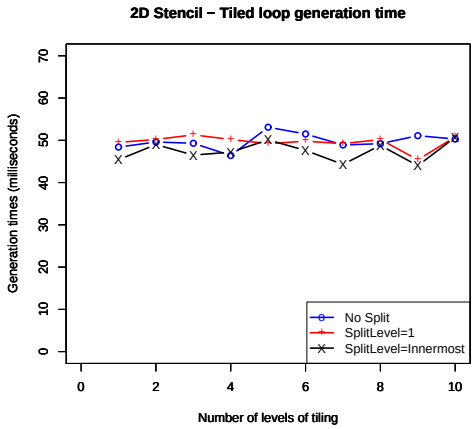

Figure 5.6: Generation time for multi-level tiling of 2D Stencil.

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 [32] instead of CLooG. We chose CLooG for its robustness across several benchmarks and its code generation speed (up to $4\times$faster than OMEGA [10]).

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 5.1. The benchmarks 2D Stencil and 3D Stencil correspond to a Gauss-Seidel 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 5.1 and for the experiments, all the loops are tiled at all the levels for all the benchmarks. 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., the

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

Table 5.1: Benchmarks used for evaluating generation efficiency and code quality.

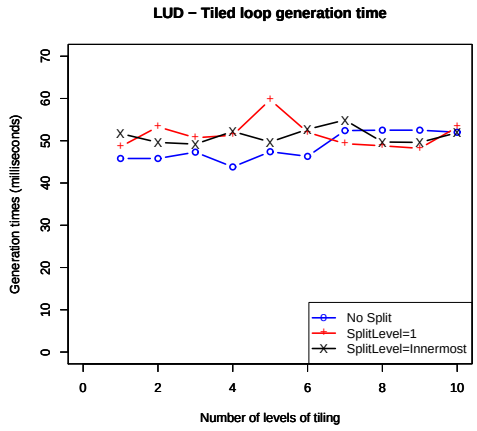

Figure 5.7: Generation time for multi-level tiling of LU decomposition.

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.

5.5.1 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 5.1.2). 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++`

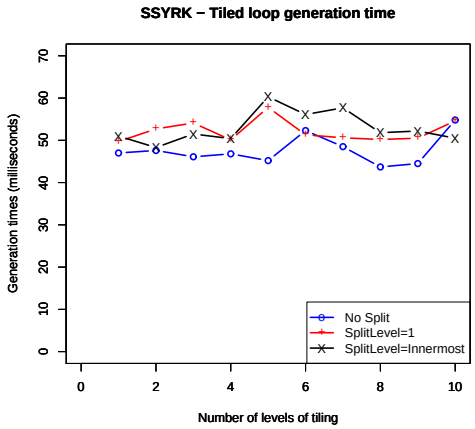

Figure 5.8: Generation time for multi-level tiling of symmetric rank k update (SSYRK).

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 5.6,5.7, 5.8,5.9, and 5.10. 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 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 our previous work [52] 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

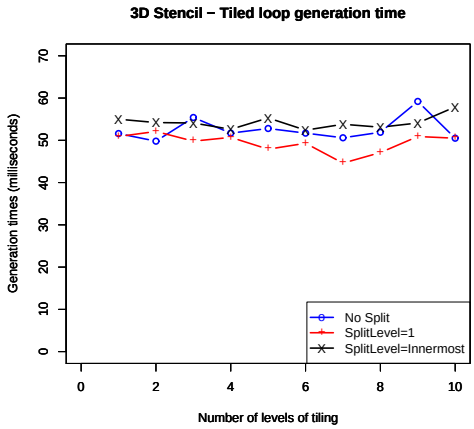

Figure 5.9: Generation time for multi-level tiling of 3D Stencil.

that splitting does not introduce any additional cost.

The generation times for the classic method for fixed tile sizes is shown in Figure 5.11. 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.

5.5.2 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-

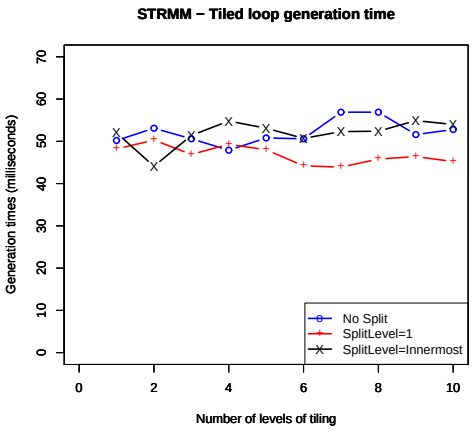

Figure 5.10: Generation time for multi-level tiling of triangular matrix multiplication (STRMM).

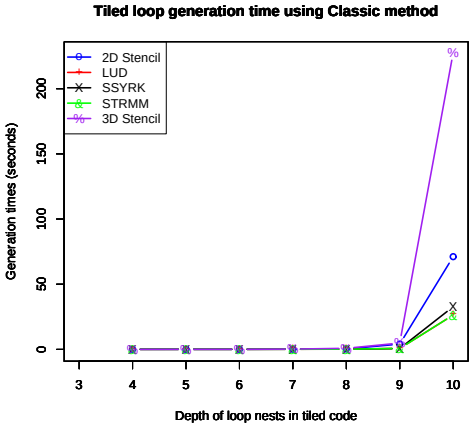

Figure 5.11: Generation time for multi-level tiling of classic method. The *x*-axis of the graph is the number of loops in the tiled loop nest. The *y*-axis is the code generation time in seconds.

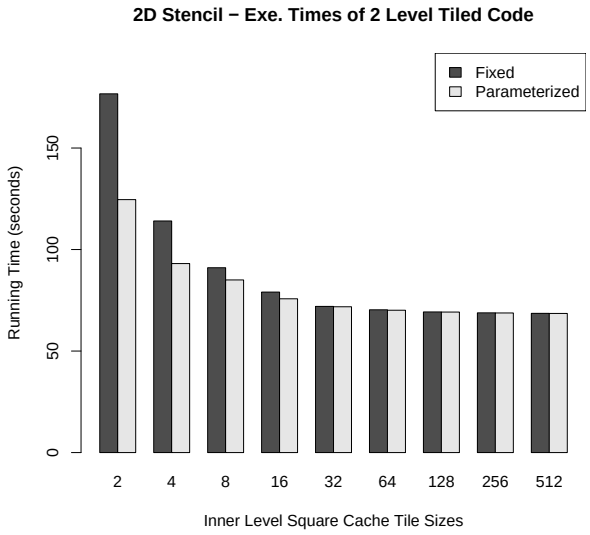

Figure 5.12: Total execution time for 2D Stencil on a data array of size 65536. The x-axis shows the inner (cache) cubic tile sizes. The outer (TLB) tile size is fixed at 512.

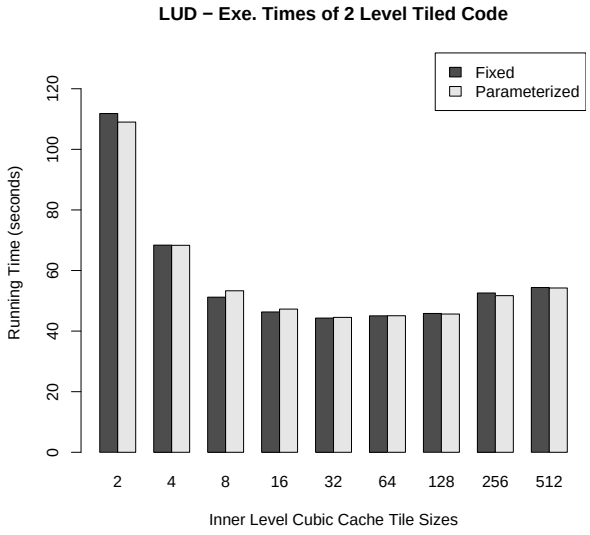

Figure 5.13: Total execution time for LU decomposition on a matrix of size 2048×2048. The x-axis shows the inner (cache) cubic tile sizes. The outer (TLB) tile size is fixed at 512.

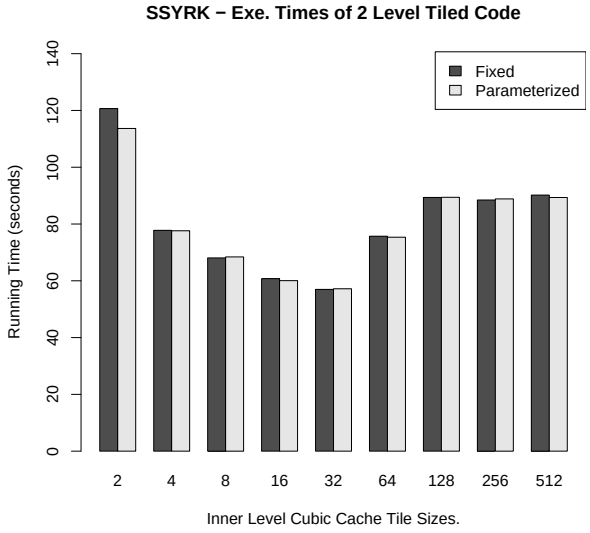

Figure 5.14: Total execution time for symmetric rank k update (SSYRK) for matrix of size 2048×2048. The *x*-axis shows the inner (cache) cubic tile sizes. The outer (TLB) tile size is fixed at 512.

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

Figures 5.12, 5.13, 5.14 and 5.15 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 shown in the graphs, first 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 (omitted for brevity) 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.

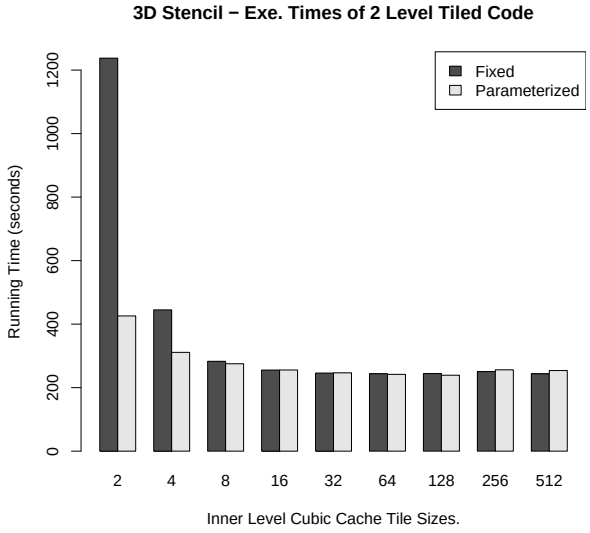

Figure 5.15: Total execution time for 3D Stencil for a data array of size 2048×2048 over 2048 time steps. The x-axis shows the inner (cache) cubic tile sizes. The outer (TLB) tile size is fixed at 512.

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.

5.5.3 Effect of separation level

We evaluate the effect of separating partial and full tiles at different levels tiling on the benchmarks. We tiled them two levels: one for cache and another for registers. The register tiles were 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 5.16-5.19. Also shown is the running time for one level of tiling for caches. There is no clear indication on "better splitting level" across all the benchmarks. It depends on the benchmarks as well as the sizes of tiling. For example, in Figure 5.16 there is 159% difference in execution time depending

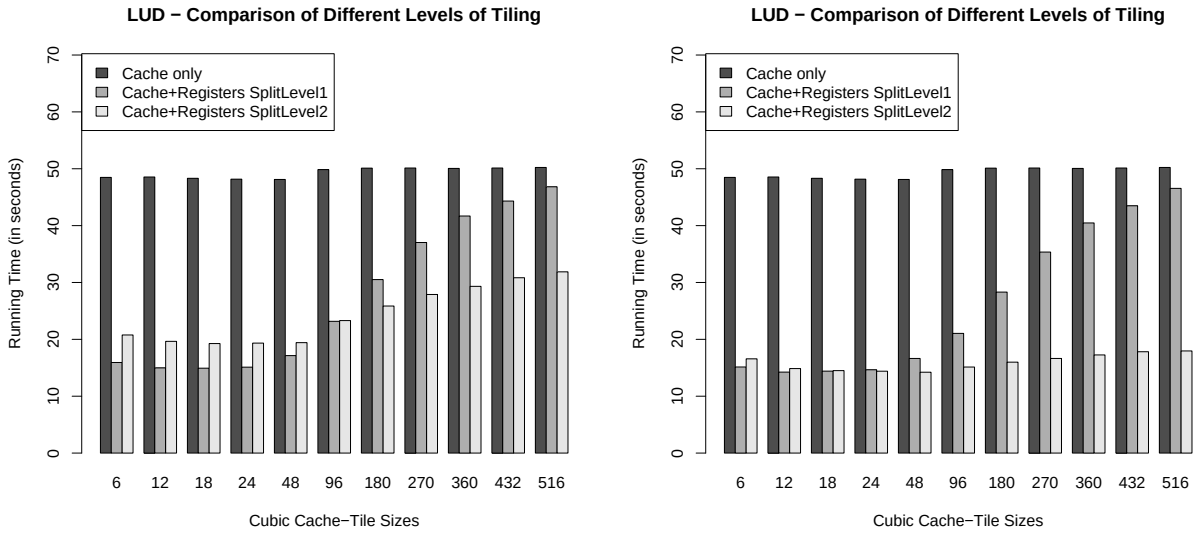

Figure 5.16: Total execution time for LU decomposition for matrices of size 2048×2048. Two levels of tiling for cache and registers is used. The *x*-axis shows the cubic cache-tile sizes. The graph on the left is for a register-tile size of $2 \times 2 \times 2$ and the one on the right is for $3 \times 3 \times 3$.

on splitting levels when cache tile sizes are $516 \times 516 \times 516$ and register tile sizes are $3 \times 3 \times 3$. On the other hands, there is only 0.6% difference when cache tile sizes become $18 \times 18 \times 18$. When we look only at the best tile sizes for each benchmark, there is up to 2-33% difference between the best points of different splitting levels. Overall, the results clearly show that splitting level has impact on performance.

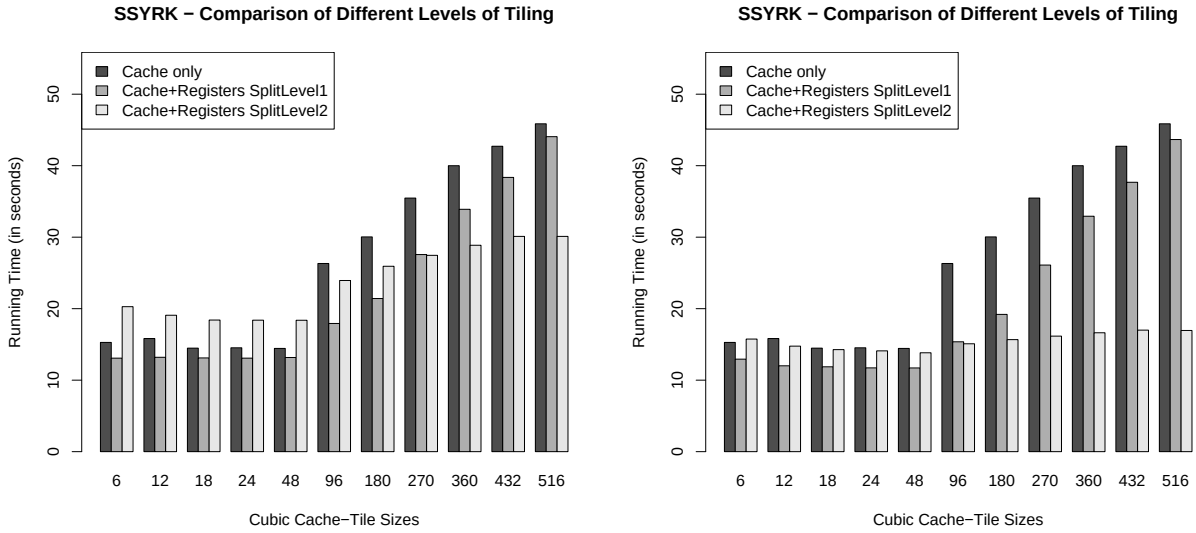

Figure 5.17: Total execution time for symmetric rank k update (SSYRK) for matrices of size 2048×2048. Two levels of tiling for cache and registers is used. The x-axis shows the cubic cache-tile sizes. The graph on the left is for a register-tile size of $2 \times 2 \times 2$ and the one on the right is for $3 \times 3 \times 3$.

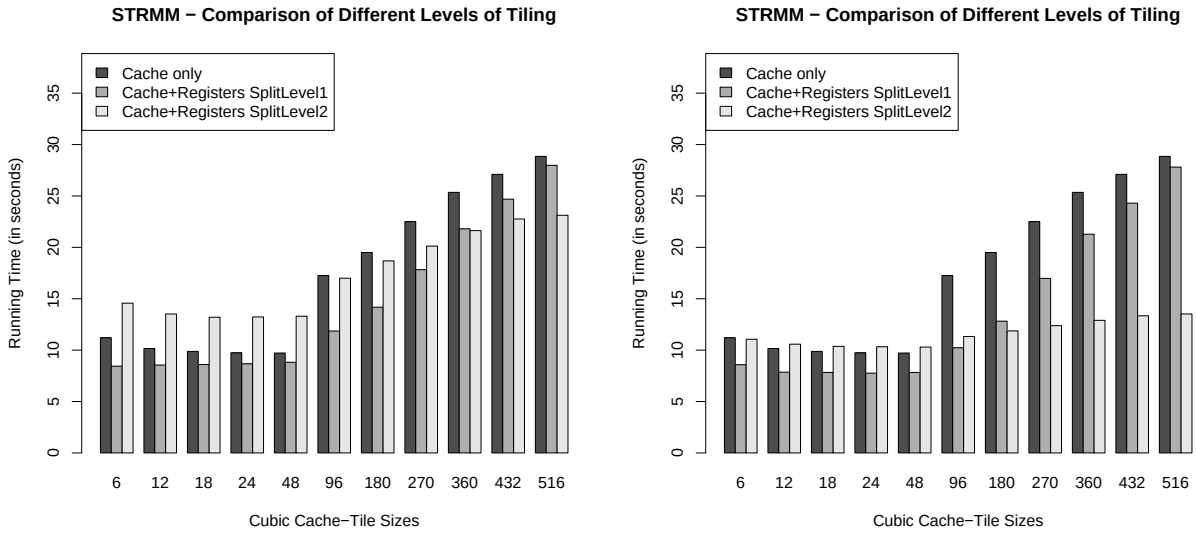

Figure 5.18: Total execution time for triangular matrix multiplication (STRMM) for matrices of size 2048×2048. Two levels of tiling for cache and registers is used. The x-axis shows the cubic cache-tile sizes. The graph on the left is for a register-tile size of $2 \times 2 \times 2$ and the one on the right is for $3 \times 3 \times 3$.

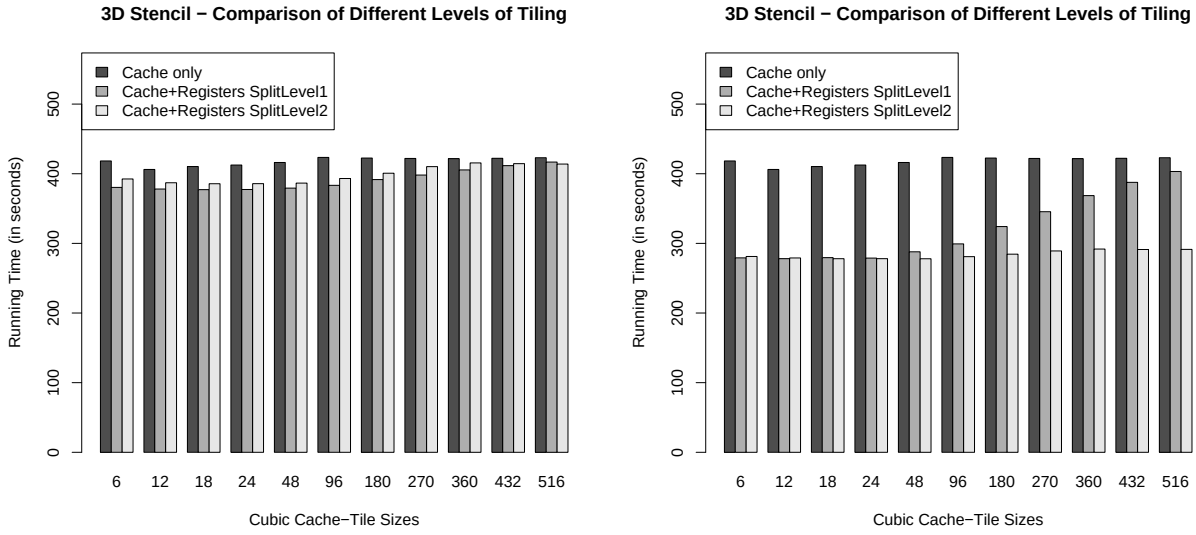

Figure 5.19: Total execution time for 3D Stencil for a data array of size 2048×2048 over 3000 time steps. Two levels of tiling for cache and registers is used. The x-axis shows the cubic cache-tile sizes. The graph on the left is for a register-tile size of $2 \times 2 \times 2$ and the one on the right is for $3 \times 3 \times 3$.

Algorithm 1 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

Chapter 6

Multi-level Tiled Loop Generation: Multiple Tiling

In this chapter we provide another approach for parameterized multi-level tiling for perfectly nested loops. In this approach we always take the original iteration space as "iteration space" for a given level. We refer this as *multiple multi-level tiling*. In the successive tiling, the approach of Chapter 5, we take tiles in the previous level as the iteration space in the current level. We provide the detailed explanation on the difference between these two approach and how these approach can be combined to achieve better performance. We also provide an technique for *syntactically* extracting the condition for a tile to be full from *the original loop nest*. Finally, we provide experimental results on these two approaches as well as their combinations.

6.1 Motivation

In successive tiling, we take tiles in the previous level as the iteration space for a given level and use the bounding box approach. Although this simplifies the tiled loop generation process, there is possibility that the generated tiled loops suffer from poor code quality. This comes from the combination of the well-known inefficiency of the bounding box approach and tiles being used as the iteration space. First, the bounding box strategy may result in poor code quality because many empty tiles will be visited and

```
// tile-loops
  // box-loops
    ...
    // box-loops
      // point-loops
        // Statements
```

```
// tile-loops
  // tile-loops + box-loops
    ...
    // tile-loops + box-loops
      // point-loops
        // Statements
```

Figure 6.1: Loop structure of multi-level tiling as successive tiling (left) and multiple tiling(right)

the emptiness will not be determined until loop bounds of the innermost loop are tested. Second, there are three kinds of tiles: full, partial and empty tiles. When an empty or partial tile is used for an iteration space for the next level tiling, many empty tiles at the next level of tiling will be enumerated. Although tiled loops produced by our technique do not enumerate many empty tiles, a single empty tile leads to enumerating many empty tiles at the next level. For example, when a cubic empty tile of size $64 \times 64 \times 64$ is tiled with $2 \times 2 \times 2$ tiles in the next level, it results in enumerating 2^{15} empty tiles. Similarly, partial tiles can lead to a significant number of empty tiles in the next level.

6.2 Multiple Tiling

In multiple tiling, we consider the original iteration space as well as tiles at the previous level. In other words, we take the intersection of the iteration space and tiles in the previous level. Although this seems a complex process involving polyhedral operations, it is achieved with just simple syntactic processing. Like successive tiling, we obtain the bounds for tiles in the previous tiles using the bounding box approach, which consists of only tile indices and tile size parameters. For the original iteration space, we apply D-tiling directly to the original loops with different tile size parameters. Then we combine these two sets of loop bounds appropriately. The structure of tiled loop nest in the multiple multi-level tiling is shown in 6.1 together with that produced by successive tiling. Although it may reduce enumerating empty tiles, tiled loops generated by this method have more complex loop bounds.

Input: AST - perfectly nested loops, t - a tile index name vector, s - a tile size vector

1: $INSET \leftarrow \emptyset$
2: **for** each k-loop in AST **do**
3: replace index names in lb_k and ub_k with tile index names
4: $lbcon \leftarrow shift\_up(lb_k)$
5: $ubcon \leftarrow shift\_down(ub_k)$
6: add $lbcon$ and $ubcon$ to $INSET$
7: **end for**

Figure 6.2: Algorithm for deriving the test for full tiles. *Shift_down* and *Shift_up* are defined in Chapter 3

6.3 Separation of Full tiles

Separating full tiles from the other tiles is an important optimization for register tiling and vectorization. Successive tiling of full tiles is trivial and multi-level tiled loops can be efficiently generated. This enables us to exploit vectorization that most current architectures support.

An algorithm for deriving the condition for testing whether a tile is full is given in Figure 6.2. One may split each loop or have one guard at the end of the tile-loops. We provide a correctness proof below. It parallels the proof and correctness of D-tiling.

It suffices to show that if $z \in INSET$ and $0 \leq \delta \leq s'$, then $z + \delta \in P_{iter}$. Suppose $z \in INSET$. Then, for $l = 1, \ldots d$

$$shift\_up(lb_l) \leq z_l$$

We can write this as

$$lb_l + \sum_{c_j > 0} c_j(s_k - 1) \leq z_l$$

Let $x = z + \delta$. Now, we replace z with $x - \delta$, i.e., z_k with $x_k - \delta_k$. We obtain

$$lb_l - \sum_{c_i < 0} c_i \delta_i + \sum_{c_j > 0} c_j(s_j - 1 - \delta_j) + \delta_l \leq x_l$$

Note that $-\sum_{c_i<0} c_i\delta_i$, $\sum_{c_j>0} c_j(s_j - 1 - \delta_j)$ and δ_l are all non negative. So, we derive $lb_l \leq x_l$.

Similarly, we can derive $x_l \leq ub_l$ from $z_l \leq shift\_down(ub_l)$. Therefore, all the points in a tile whose origin is z belong to P_{iter}.

6.4 Implementation and Experiments

We implemented our tiled loop generation algorithms as a simple walk on a loop AST (Abstract Syntax Tree) generated by the SableCC [21].

We evaluated the efficiency of our technique using three common benchmarks: Gauss-Seidel stencil code (Seidel) on a two-dimensional data array, a triangular matrix multiplication (DTRMM), and symmetric rank k update (DSYRK). We compared our implementation with PrimeTile and HiTLOG on generation efficiency and generated code efficiency. We compiled with gcc 4.4 and Intel icc 11.1 and ran all experiments for generated code efficiency evaluation on an Intel Core 2 Duo running 2.2 GHz with 2MB L2 Cache and 1GB memory.

6.4.1 Evaluation

All three code generators reported similar generation time. Given that depths of loops in the benchmarks are small, this was expected. Due to the fact that these code generators are written in different languages and all the generation time are still less than 0.1 seconds, we do not report these data here. The generation time of PrimeTile with boundary tiles fully recursively tiled (represented as Prime(f)) grows faster than the others and its code size grew exponentially with tiling levels.

Figure 6.3 and 6.4 show the execution time of eight different tiled loops for two levels of tiling: one for cache and one for registers—unroll-jam and scalar promotion. For bigger tile sizes, levels of splitting full tiles have more impact, but for smaller tile sizes the tiling schemes have more impact on execution time. For most cases, mixed tiling with splitting at the innermost level performs better than the others.

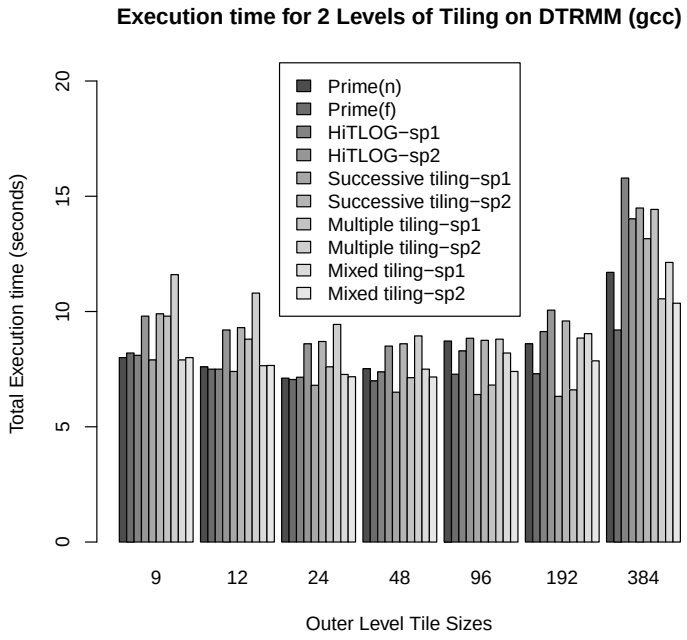

Figure 6.3: Total execution time for DTRMM with two levels of tiling, one for cache and one for registers, and compiled with gcc. The register tile size are fixed at $3 \times 3 \times 3$.

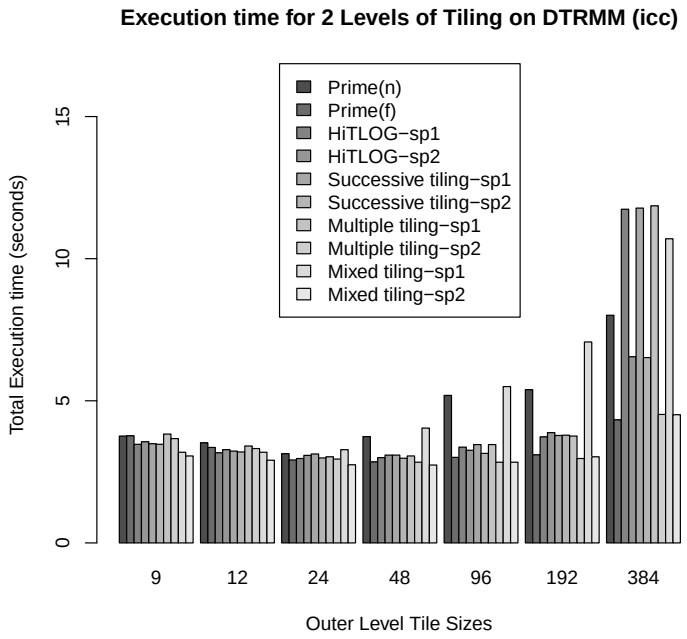

Figure 6.4: Total execution time for DTRMM with two levels of tiling, one for cache and one for registers, and compiled with gcc. The register tile size are fixed at $3 \times 3 \times 3$.

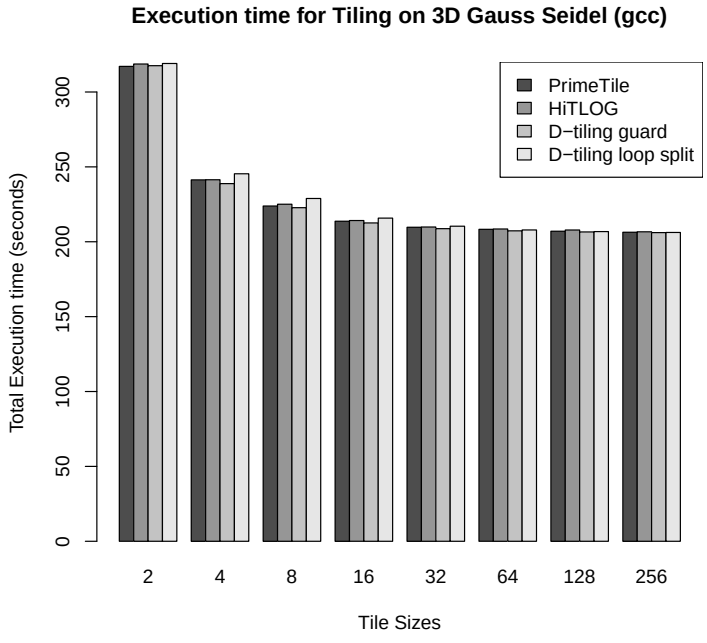

Figure 6.5: Total execution time for Seidel with separating full tiles, compiled with gcc

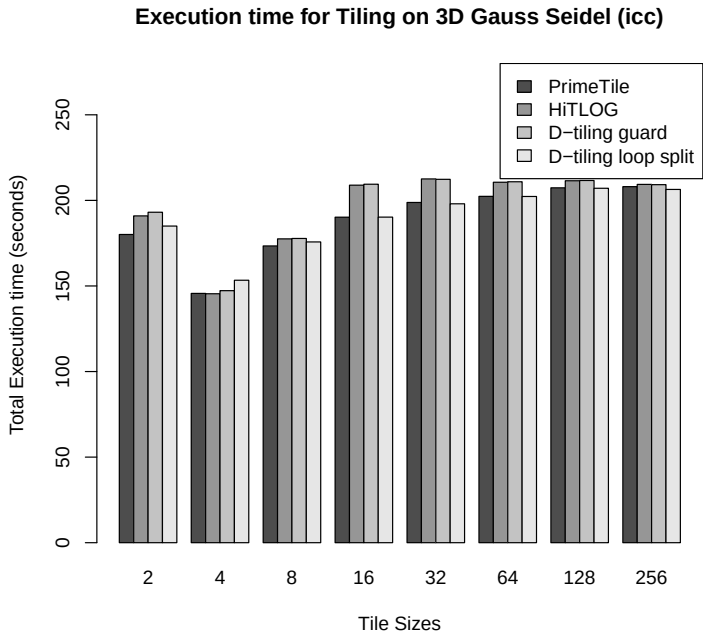

Figure 6.6: Total execution time for Seidel with separating full tiles, compiled with icc

Figure 6.5 and 6.6 show the total execution time of 3D Gauss Seidel computation with one level of cache tiling with separating full tiles using various methods. Since the condition for full tiles is also constructed dimension by dimension, we can easily split each loop into three sections. Then, code size increases significantly like Prime(f). Note that the best code depends on even tile size. Splitting in D-tiling is comparable to the others. In summary, tiling schemes have an impact of generated code.

Chapter 7

Multi-level Tiled Loop Generation for Imperfectly Nest Loops

We solve this problem by simply composing our techniques for generating single-level tiled loops for imperfectly nested loops and multiple-level tiled loops for perfectly nested loops. From the techniques presented so far, there are three ways to generate multi-level parameterized tiled loops from imperfectly nested loops. The first method is applying our tiling scheme for imperfectly nested loops to the point-loops produced. The second and third are applying successive and multiple tiling to the tile-loops. Notice that the tile-loops produced by our scheme is perfectly nested, so these two methods for perfectly nested loops can be directly applicable. These three schemes provide various loop structures that can be selected for various optimization and parallelization schemes.

7.1 Successive/Multiple Tiling on Tile-loops

In our scheme for imperfectly nested loops, we preserve tile-loops to be perfectly nested for subsequent processes, such as parallelization. Another advantage of perfectly nested tile-loops is that we can directly apply multi-level tiling methods developed for perfectly nested loops to these perfectly nested tile-loops. We first extract perfectly nested loops from embedded imperfectly nested loops and use them to generate all the tile loops. For point-loops, we apply our scheme for imperfectly nested loops with the tile sizes at the

last level, and obtain point-loops. Finally, we insert the point-loops as the body of the last tile-loop.

7.2 Imperfectly Nested Loop Tiling on Point-loops

For a given loop nest, we apply our scheme for imperfectly nested loops. It produces perfectly nested tile-loops and a set of point-loops from embedded imperfectly nested loop. Each set of point-loops is again an embedded imperfectly nested loop. While applying tiling to point loops, we have two choices: taking only tile bounds like successive tiling and taking both iteration bounds and tile bounds like multiple tiling. In any cases, we do not change tile bounds. We apply D-tiling only to the original loop bounds.

7.3 Mixture of various approach

The two approaches presented in Section 7.1 and 7.2 are extreme cases. The approaches can be applied whenever it is applicable. Note that our scheme for imperfectly nested loops generates a set of point-loops some of which are perfectly nested. We apply either successive or multiple tiling to those point-loops. We can always apply our scheme for imperfectly nested loops to any point-loops. We may also choose to apply tiling to only a subset of these point-loops to achieve better performance.

Perfectly nested tile-loops are preferable for parallelization, although they might have more loop overhead in sequential execution. Particularly, a successive tiling is suitable for MPI parallelization because each processor will enumerate the same number of tiles. However, a detailed discussion on how to use tiled loop generation schemes to a particular optimization is beyond the scope of this thesis.

Chapter 8

Parallel Execution of Parameterized Tiles

In this chapter we propose a technique for executing tiles in a wavefront fashion. Consistent with our overall approach, this will be done syntactically by restructuring loops so that they enumerate tiles in an order different from the original.

Traditionally, this restructuring has been realized through a skewing transformation on tile space followed by loop generation.

Consider the iteration space $\mathcal{I}$ in Figure 8.1:

$$\mathcal{I} = \{i, j \mid 0 \leq i \leq M + N \wedge max(0, i - M) \leq j \leq min(N, i)\}$$

Its tile space can be formulated as the canonical projection onto the first two dimensions of $\{t_i, t_j, i, j \mid s_i \times t_i \leq i \leq s_i \times t_i + s_i - 1 \wedge s_j \times t_j \leq i \leq s_j \times t_j + s_j - 1 \wedge (i, j) \in \mathcal{I}\}$. So, when tile sizes, s_i and s_j, are fixed (i.e., constant integers), the tile space remains a polyhedron. Therefore, a skewing transformation such as $(t_i, t_j \rightarrow t_i + t_j, t_j)$ can be applied. The scanning loop for this skewed space allows the tiles to be executed in a wavefront fashion. Figure 8.1 also conceptually shows how these loops enumerate the tiles. However, this approach works only when tile sizes are fixed and hence the tile spaces are formulated as polyhedra.

Now, consider the tile space as an intersection of polyhedra and an integral lattice,

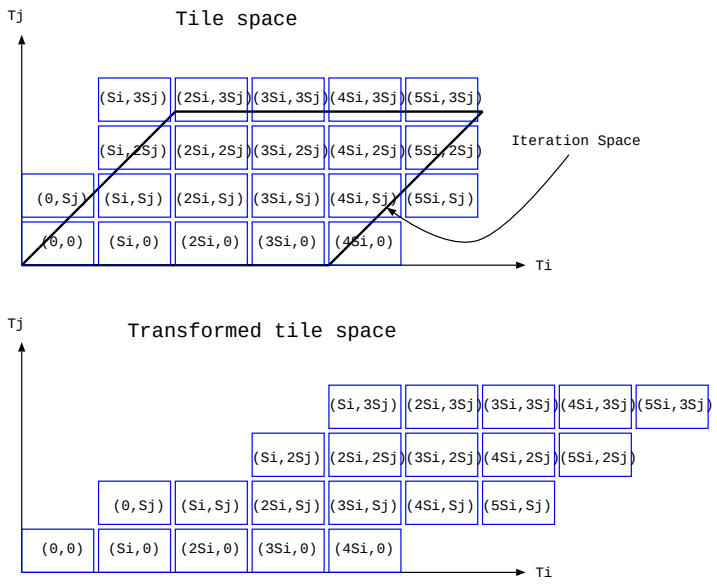

Figure 8.1: Tile space and its transformed space; each tile is denoted by its origin coordinate; note that even in the transformed space tiles are denoted by their origin coordinates in the *original* space

i.e., tile sizes are fixed. For instance, consider the integral lattice $(3, 2)$ where all the tile origins are a form of (t_i, t_j) where t_i is a multiple of 3 and t_j is a multiple of 2. The precise transformation for skewing tiles is not $(t_i, t_j \rightarrow t_i + t_j, t_j)$. Note that a tile $(0, 2)$ must be mapped to $(3, 2)$ by the skewing transformation, not $(2, 2)$. So, the skewing transformation for such formulation of tile space is $(t_i, t_j \rightarrow t_i + \frac{3}{2}t_j, t_j)$. Not all the coefficients are integers.

Now, consider a tile space where tile sizes are parameterized and the space is viewed as an intersection of polyhedra and a *parameterized* integer lattice. Figure 8.1 shows an example of parameterized tiling and skewing transformation. Note that figures are not a precise representation for parameterized tiling. The figure gives an impression that the width of a tile is larger than its height, even though such assumption is not made. In this case, transformation for skewing tiles is formulated as $(t_i, t_j \rightarrow t_i + (s_i/s_j) \times t_j, t_j)$. To the best of our knowledge, polyhedral framework does not support this transformation. Hence, neither transformation nor code generation technique in the polyhedral model

can be applied.

We present a technique for restructuring parameterized tiled loops so that tiles will be executed in wave-front parallel scheduling. Our technique is simple, but based on two key intuitions that at first glance seem to be in conflict with.

8.1 Basic Ideas Illustrated

One of two main ideas behind our approach is that we know all the tiles that need to be visited for a given time stamp. Consider that we find $(S_i, 2S_j)$ at the original tiled space in Figure 8.1. Without transforming the tiled loops, we know that $(2S_i, S_j)$ and $(3S_i, 0)$ tiles need to be executed with $(S_i, 2S_j)$. All valid tiles that can be written as $(S_i, 2S_j) + k \times (S_i, -S_j)$ for some k need to be visited with $(S_i, 2S_j)$. In other words, $(S_i, 2S_j)$ will be executed at $time = 3$ computing from $(S_i)/S_i + (2S_j)/S_j$. Any tiles (t_i, t_j) where $t_i/S_i + t_k/S_j = 3$ will be executed with $(S_i, 2S_j)$ at $time = 3$. The wave-front schedule of parameterized tiled loops for this example can be expressed as $t_i/s_i + t_j/s_j$.

In general, the wave-front schedule is

$$time = \sum_{k=1}^{d} t_k/s_k$$

The other key idea is that we cannot change the original scanning order without a new projection even though we know what tiles need to be visited and a set of parameterized spanning vectors whose linear combination can express any point in a space. Note that this is true even with conventional loop generation algorithms. We do not have a mechanism to compute a projection by a non-integer matrix. Therefore, we can specify the time at which a tile needs to be executed but the schedule itself cannot be used to transform the space.

Our approach uses the original scanning order, and computes the last tile index using the schedule. For our running example, we use the outer loop of the original tile-loops,

and compute t_j using $t_j = (time - t_i/S_i) * S_j$. The bounds of t_j are used for checking the validity of a tile. Now, the only necessary information is its time stamp. We use the time stamp of the first tile as starting time and that of the last tile as ending time stamp. We iterate over time and the original tile-loops with appropriate processing will visit all the tiles that need to be executed for a given time stamp.

8.2 Generation Algorithm

The main target input to our algorithm is a set of perfectly nested parameterized tile-loops. This is not a severe restriction and the algorithm can be be applied to any sets of perfectly nested affine loops. A loop at depth k is denoted by an iterator t_k, a lower and upper bounds lb_k and ub_k, respectively, and a step size s_k. The maximum depth of the loops being considered is d. Note that we are interested only in tile-loops, not point loops. We first compute the first tile by

$$t_1 = lb_1, t_2 = lb_2(t_1), \cdots, t_d = lb_d(t_1, \ldots, t_{d-1})$$

Similarly, we compute the last tile

$$t_1 = ub_1, t_2 = ub_2(t_1), \cdots, t_d = ub_d(t_1, \ldots, t_{d-1})$$

Now, we compute the first and last time stamp from these two tiles using the schedule. We assume that the first and last tile in the original tile-loops will be executed at the first and last time stamp, respectively. Most applications that require wave-front scheduling, such as LU and Cholesky decomposition and stencil computations, satisfy this assumption. Also, it is unlikely for this assumption to be violated from the fact that the loops need to be fully permutable for parameterized tiling with hyper-rectangular shape.

Now, we construct a loop that scans all the time stamps between the first and last time stamps. We denote this loop as $(time, start, end, 1)$. We directly use the input loop

Input: AST - perfectly nested loops up to depth d, a loop L_k at depth k consists of iterator t_k, lower bound lb_k, upper bound ub_k and step size s_k

1: **for** k= 1 to d **do**
2: print "$t_k \leftarrow lb_k$"
3: **end for**
4: print "$start \leftarrow t_1/s_1 + \cdots + t_d/s_d$"
5: **for** k= 1 to d **do**
6: print "$t_k \leftarrow ub_k$"
7: **end for**
8: print "$end \leftarrow t_1/s_1 + \cdots + t_d/s_d$"
9: print $loop(time, start, end, 1)$
10: **for** k=1 to d-1 **do**
11: print L_k
12: **end for**
13: print "t_d = $(time - (t_1/s_1 + \cdots + t_{d-1}/s_{d-1})) \times s_d$"
14: print a guard whose condition is $(lb_d \leq t_d) \wedge (t_d \leq ub_d)$ and whose body of L_d as its body

Figure 8.2: Generation algorithm for wave-front scheduling of parameterized tiled loops

nest except the innermost loop. The innermost loop will be replaced by the following assignment statement

$$t_d = \left(time - \sum_{k=1}^{d-1} t_k/s_k \right) \times s_d$$

and a guard whose condition is

$$lb_d \leq t_d \wedge t_d \leq ub_d$$

and where body is the loop body of the original loop nest.

The algorithm is presented in Figure 8.2. One can easily replace the scheduling equation with a different one, if so desired.

The outermost loop in the original parameterized tiled loops can be marked as a parallel loop given that wave-front scheduling is legal. Furthermore, any loops that are from the original tile-loops can be marked as parallel.

```
for (t_i = 0; t_i ≤ M + N; t_i += s_i)
  for (t_j = max(0, shift_up(t_i − M − s_j + 1, s_j)); t_j ≤ min(t_i + s_i − 1, N); t_j += s_j)
    for (i = max(0, t_i); i ≤ min(M + N, t_i + s_i − 1); i++)
      for (j = max(0, i − M, t_j); j ≤ min(i, N, t_j + s_j − 1); j++)
        S1(i, j);
```

Figure 8.3: A simplified parameterized tiled loop for the example in Figure 8.1; the actual loop body is replaced by a macro for brevity; the loop body is irrelevant to the generation algorithm

8.3 Algorithm Walk-through

The simplified tiled loops for the example in Figure 8.1 is given in Figure 8.3. Now, we apply our algorithm to these loops. Note that we are interested in only tile loops, i.e., up to depth 2 for this loop nest.

First we compute the first tile by assigning the lower bound for each loop. So, $t_i = 0$ and $t_j = max(0, shift\_up(t_i - M - s_j + 1, s_j))$. For this, we construct a statement computing the first time stamp $start$ by $t_i/s_i + t_j/s_j$. Note that we are generating code, not executing code. Similarly, we generate the assignment for the last tile origin by taking upper bounds by

$$t_i = shift\_down(M + N, s_i), t_j = shift\_down(min(t_i + s_i - 1, N), s_j)$$

Note that the upper bounds may not be a multiple of tile size for a given depth. Similarly, we create a statement for assigning end by the time stamp of this last tile. These are from line 1 to 8 in Figure 8.4.

Now, we construct a loop using a new iterator $time$. Its lower bound is $start$, and its upper bound end. The step size is 1. This loop iterates over "time" of the wave-front schedule. Then, we construct a loop nest that scans all the tiles for a given time. We take the outermost loop directly, but the t_j loop becomes a statement and a guard. The scheduling equation provides an equation for t_j from a given t_i and $time$. The loop bounds of t_j in the original loop give the precise condition for the valid range of t_j. The body of t_j loop will be the body of this new guarded statement.

```
t_i = 0; t_j = max(0, shift_up(t_i − M − s_j + 1, s_j));
start = t_i/s_i + t_j/s_j;
t_i = shift_down(M + N, s_i); t_j = shift_down(min(t_i + s_i − 1, N), s_j);
end = t_i/s_i + t_j/s_j;
for (time = start ; time ≤ end ; time++)
  for (t_i = 0; t_i ≤ M + N; t_i += s_i)
    t_j = (time − t_i/s_i) × s_j
    if (max(0, shift_up(t_i − M − s_j + 1, s_j)) ≤ t_j ∧ t_j ≤ min(t_i + s_i − 1, N))
      for (i = max(0, t_i); i ≤ min(M + N, t_i + s_i − 1); i++)
        for (j = max(0, i − M, t_j); j ≤ min(i, N, t_j + s_j − 1); j++)
          S1(i, j);
```

Figure 8.4: Transformed loop nest for wave-front scheduling from the parameterized tiled loops in Figure 8.3

The final code produced by our algorithm is given in Figure 8.4.

8.4 Experimental Results

We first measure the loop overhead of the restructured tiled loops against the original tiled loops. For this, we replace the loop body by simple counters. We do this because the data locality aspect does not affect the loop overhead. Due to parallel execution, data locality of the restructured program is worse than that of the original program.

Then, we evaluate the efficiency of our generated code using OpenMP on multi-core machines. Since there are no available tools for executing parameterized tiled program with wave-front scheduling, we compare with tiled programs with constant tile sizes.

We performed our experimentation on four well-known kernels. The characteristics of these four kernels are given in Table 8.1. We compiled all the code with gcc 4.4.2 and ran all the code on an Intel eight-core (two Intel Xeon E5450 Quad Core) machine running at 3.0 GHz with 6MB L2 Cache and 16GB memory.

Figure 8.5 shows the loop overhead of three kernels between before and after loop restructuring for wave-front scheduling. The overhead due to transformations is not significant.

Figure 8.6 shows the total execution time of MultiTriSolver and Gauss-Seidel stencil computation. MultiTriSolver consists of independent triangular system solvers, so it has

	Description	Perfectly nested	Program Parameters	Wave-front needed
MultiTriSolver	Multiple triangular linear systems solver	No	M=3000,N=2000	No
LU	LU decomposition without pivoting	No	N=3000	Yes
Cholesky	Cholesky decomposition	No	N=3000	Yes
Seidel	Gauss Seidel style stencil computation on 2d data	Yes	TMAX=1000, NX/NY=2000	Yes

Table 8.1: Four benchmarks for the evaluation of the restructured parameterized tiled loop nest for parallel execution on shared memory systems

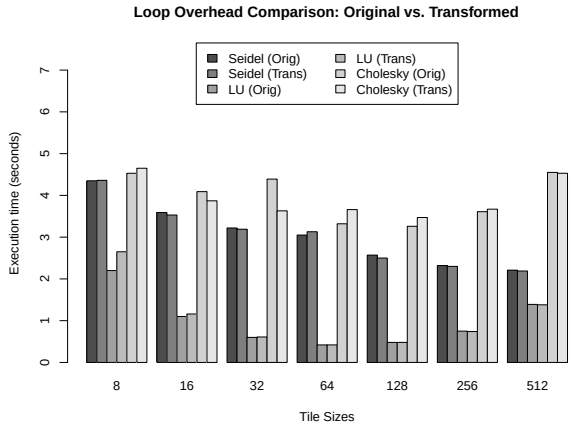

Figure 8.5: Loop overhead comparison between before/after loop restructuring

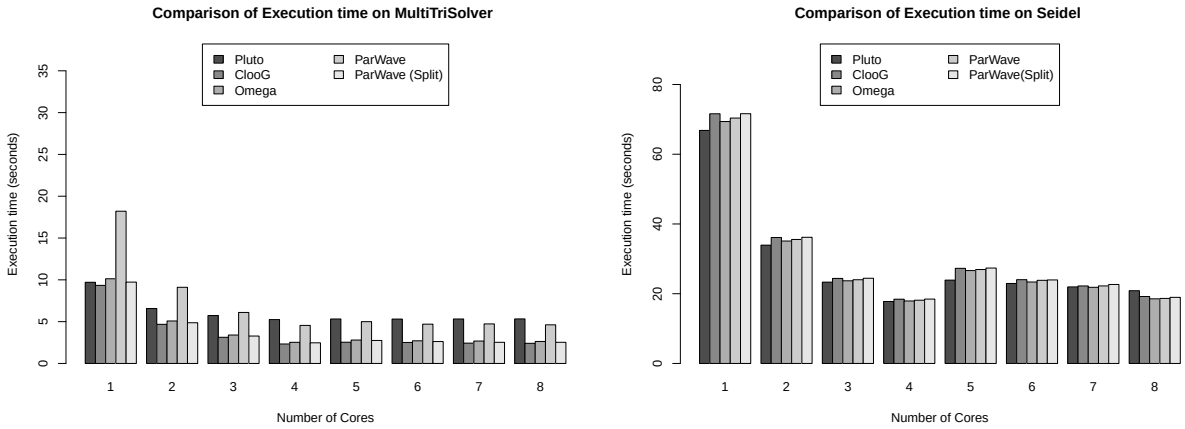

Figure 8.6: Total execution time of Multiple triangular solver and Gauss-Seidel stencil computation. All the loops in fixed tile size code are tiled with $16 \times 16 \times 16$. ParWave is parameterized tiled code without splitting full tiles, and ParWave(Split) with splitting full tiles

synchronization-free parallelism. So this kernel does not require the wave front scheduling, and therefore the original parameterized tiled loops are used without restructuring. We generated three reference codes using Pluto [14], CLooG, Omega. We generated code from Pluto with option -tile and -parallel, and tile sizes were fixed to $16 \times 16 \times 16$. We directly used the generated code without any modification. Note that Pluto does not always choose wave-front scheduling for parallelization. In order to obtain code with wave-front scheduling, we generated code using CLooG and Omega with the same tile sizes. We used $(t_1, t_2, t_3 \rightarrow t_1+t_2+t_3, t_1, t_2)$ as a transformation on tiles, and this transformation provides the most similar code to our restructured parameterized tiled code. We generated two versions of code using our technique: one without full-tile splitting (ParWave) and the other with splitting (Parwave(Split)). The restructuring process is same, but the starting parameterized tiled loops are different. Our approach provides comparable performance to the fixed size tiling with skewing transformation, but the tile sizes remain as runtime parameters.

Figure 8.7 shows the total execution time of LU and Cholesky. We generate code

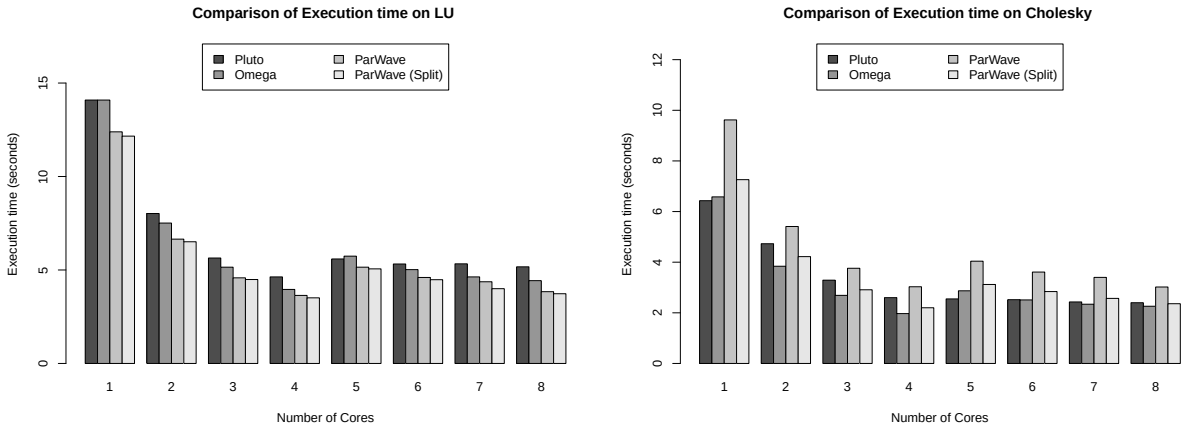

Figure 8.7: Total execution time of Cholesky and LU. All the loops in fixed tile size code are tiled with $48 \times 48 \times 48$. ParWave is parameterized tiled code without splitting full tiles, and ParWave(Split) with splitting full tiles

from Omega using option 2 for LU. Omega does not generate a correct code with option 3 from our specification of LU.

Overall, the wave-front scheduling of parameterized tiled code delivers comparable performance and even better for some cases. Our technique not only enables parallelization but also provide competitive performance.

Chapter 9

Conclusions and Future Work

Tiling transformation facilitates many important optimizations: data locality optimization, coarse-grained parallelism, and vectorization. Existing solutions extract a mathematical representation of loops, compute either its tile space or its tiled space, and finally regenerate loops from those space. Heavy machinery for generating tiled loop nests and the complexity of the generated tiled loops has limited its use to optimization experts and advanced optimizing compilers. We achieve a solution where tiled loops are produced by processing original loop nests *directly* and *syntactically*. The techniques presented in this thesis provide well-structured and hence human-readable parameterized tiled loops with performance comparable to highly optimized complex tiled loops without any sophisticated mathematical libraries. In addition to the ability of restructuring parameterized tiled loops for wave-front scheduling, our collection of techniques retains all the positive attributes of the simple technique that is used to generate tiled loop nested from rectangular iteration space with hyper-rectangular tile shape: generation of parameterized, fixed and mixed—where some tile sizes are fixed and others remain parameters—tiled loops within a single framework, generation efficiency (no worst-case exponential complexity), scalable multi-level tiled loop generation, and separation of full tiles from the others.

Although we achieve a solution for tiled loop generation problem without any poly-

hedral operations, our techniques greatly benefit from many intuitions and concepts based on the polyhedral model, specially from those in the techniques for generating loops from polyhedra.

The problem addressed in this thesis is limited to the tiled loop generation, not the whole aspect of tiling. One important direction of future research is generating complete tiled code with both the tiled loops and the appropriately transformed loop body for particular optimization, such as register tiling, vectorization and parallelization using accelerators. Another interesting direction is developing an auto-tuning framework for tile size selection and incorporation of this technique into iterative compilers.

REFERENCES

[1] Anant Agarwal, David A. Kranz, and Venkat Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 6(9):943–962, 1995.

[2] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *Int. J. Parallel Program.*, 29(5):493–544, 2001.

[3] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence Based Approach*. Morgan Kaufman, San Francisco, 2002.

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

[5] Saman P. Amarasinghe and Monica S. Lam. 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*, pages 126–138, New York, NY, USA, 1993. ACM Press.

[6] Corinne Ancourt and Francois Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[7] Rumen Andonov, Stephan Balev, Sanjay V. Rajopadhye, and Nicola Yanev. Optimal semi-oblique tiling. *IEEE Trans. Parallel Distrib. Syst.*, 14(9):944–960, 2003.

[8] R. Bagnara, P. M. Hill, and E. Zaffanella. 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.

[9] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, April 2010.

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

[11] C. Bastoul, A. Cohen, A. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. Technical Report 4902, INRIA Rocquencourt, 2003.

[12] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, Maria J. Garzaran, David Padua, and Christoph von Praun. 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*, pages 48–57, 2006.

[13] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347. ACM Press, 1997.

[14] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical and fully automatic polyhedral program optimization system. In *ACM SIGPLAN PLDI*, June 2008.

[15] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (pen)-ultimate tiling? *Integr. VLSI J.*, 17(1):33–51, 1994.

[16] Steve Carr and R. B. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Trans. Math. Softw.*, 23(3):336–361, 1997.

[17] J. Dongarra, G. Bosilca, Z. Chen, V. Eijkhout, GE Fagg, E. Fuentes, J. Langou, P. Luszczek, J. Pjesivac-Grbovic, K. Seymour, et al. Self-adapting numerical software (SANS) effort. *IBM Journal of Research and Development*, 50(2/3):223, 2006.

[18] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of international conference on Supercomputing SC*, page 83, 2006.

[19] P. Feautrier. Dataflow analysis of array and scalar references. *Int. Journal of Parallel Programming*, 20(1):23–53, 1991.

[20] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Op'erationnelle*, 22, 1988.

[21] Etienne Gagnon and Laurie Hendren. An object-oriented compiler framework. In *In Proceedings of TOOLS*, pages 140–154, 1998.

[22] Georgios Goumas, Maria Athanasaki, and Nectarios Koziris. An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10), October 2003.

[23] Armin Größlinger, Martin Griebl, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. In Michael Gerndt and Edmond Kereku, editors, *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, Research Report Series, pages 1–12. LRR-TUM, Technische Universität München, July 2004.

[24] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 147–157, New York, NY, USA, 2009. ACM.

[25] Albert Hartono, Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Dyntile: Parametric tiled loop generation for effective parallel execution on multicore processors. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

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

[27] Karin Hogstedt, Larry Carter, and Jeanne Ferrante. Determining the idle time of a tiling. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 160–173, New York, NY, USA, 1997. ACM Press.

[28] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM Press, 1988.

[29] F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM Symposium on Principles of Programming Languages*, pages 319–328. ACM, Jan 1988.

[30] Marta Jiménez, José M. Llabería, and Agustín Fernández. Register tiling in non-rectangular iteration spaces. *ACM Trans. Program. Lang. Syst.*, 24(4):409–453, 2002.

[31] Marta Jiménez, José M. Llabería, and Agustin Fernández. A cost-effective implementation of multilevel tiling. *IEEE Trans. Parallel Distrib. Syst.*, 14(10):1006–1020, 2003.

[32] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, 1995.

[33] DaeGon Kim and Sanjay Rajopadhye. Parameterized tiling for imperfectly nested loops. Technical Report 09-101, Colorado State University, 2009.

[34] Daegon Kim, Lakshminarayanan Renganarayana, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. 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.

[35] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. 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*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[36] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Iterative compilation. In *Embedded processor design challenges: systems, architectures, modeling, and simulation-SAMOS*, pages 171–187. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[37] Monica S. Lam and Michael E. Wolf. A data locality optimizing algorithm (with retrospective). In *Best of PLDI*, pages 442–459, 1991.

[38] H. Le Verge, V. Van Dongen, and D. Wilde. 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.

[39] H. Le Verge, V. Van Dongen, and D. Wilde. Loop nest synthesis using the polyhedral library. Technical Report PI 830, IRISA, Rennes, France, May 1994. Also published as INRIA Research Report 2288.

[40] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *International Conference on Supercomputing*, pages 228–237, 1999.

[41] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *PPoPP '01*, pages 103–112, New York, USA, 2001. ACM Press.

[42] David K. Lowenthal. Accurately selecting block size at runtime in pipelined parallel programs. *Int. J. Parallel Program.*, 28(3):245–274, 2000.

[43] N. Mitchell, N. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.

[44] Dimitrios S. Nikolopoulos. Dynamic tiling for effective use of shared caches on multithreaded processors. *International Journal of High Performance Computing and Networking*, pages 22 – 35, 2004.

[45] PolyLib: A library of polyhedral functions. Available at: `http://icps.u-strasbg.fr/PolyLib/`.

[46] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for gcc. In *Proc. of the 4th GCC Developper's Summit*, pages 179–198, June 2006.

[47] W. Pugh. Omega test: A practical algorithm for exact array dependency analysis. *Comm. of the ACM*, 35(8):102, 1992.

[48] Markus Puschel, Jose M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. SPRIALl: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):21–45, 2004.

[49] Fabien Quilleré;, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal Parallel Programming*, 28(5):469–498, 2000.

[50] J. Ramanujam and P. Sadayappan. Tiling multidimensional itertion spaces for multicomputers. *J. Parallel Distrib. Comput.*, 16(2):108–120, 1992.

[51] Lakshminarayanan Renganarayanan, Manjukumar Harthi-kote, Rinku Dewri, and Sanjay Rajopadhye. Towards optimal multi-level tiling for stencil computations. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS) (to appear)*, 2007.

[52] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *PLDI '07: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 405–414, New York, NY, USA, 2007. ACM Press.

[53] Gabriel Rivera and Chau-Wen Tseng. Locality optimizations for multi-level caches. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 2, New York, NY, USA, 1999. ACM Press.

[54] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[55] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.

[56] R. P. Wilson, Robert S. French, Christopher S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S-W. Liao, C-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[57] Jingling Xue. *Loop Tiling For Parallelism*. Kluwer Academic Publishers, 2000.