Reordering and Storage Optimizations for Scientific Programs

by

Geoffrey Roeder Pike

B.A. (Harvard University) 1992

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Paul N. Hilfinger, Chair
Professor Katherine Yelick
Professor Lior Pachter

Spring 2002

The dissertation of Geoffrey Roeder Pike is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2002

Reordering and Storage Optimizations for Scientific Programs

by

Geoffrey Roeder Pike

Abstract

Reordering and Storage Optimizations for Scientific Programs

by

Geoffrey Roeder Pike

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Paul N. Hilfinger, Chair

We present the design and implementation of compiler optimizations that choreograph the use of data in scientific programs. Scientific programs often include multiple loops over the same data, where completing each loop before starting the next discards opportunities for fine-grained data reuse. Interleaving parts of different loops may greatly improve performance. Our approach combines reordering optimizations such as loop fusion and tiling with storage optimizations such as eliminating or reducing the size of temporary arrays.

The programmers we have in mind are willing to spend some time tuning their code and their compiler parameters. Given that, and the difficulty in statically selecting parameters such as tile sizes, it makes sense to provide automatic parameter searching alongside the compiler. Furthermore, including automatic parameter searching logically leads one to include more aggressive and speculative optimizing transformations in the compiler. Our strategy is to optimize aggressively but to expose the compiler's decisions to external control. Since we expect to generate numerous executables during the tuning process, optimizations that *may* pay off

are relatively more important.

Our implementation is in a library invoked from tc, a compiler for the Titanium language. We give an overview of Titanium and tc, describe our reordering and storage optimizations, and present experimental results with and without parameter search. We double or triple the performance of Gauss-Seidel relaxation and multigrid, and we argue that ours is the best compiler for that kind of program.

Chair Date

For Marianne

Acknowledgements

First and foremost, thanks to friends and family. I enjoyed the great times and weathered the bad with my friends Marianne, Jeff, Bill, Webby, Laura, Todd, Jim, Franco, David, Jonah, Dave, Kim, Kirsten, Sarah, and Brandy. Professionally, Paul Hilfinger has been my mentor. I thank him and the rest of the Titanium group: Kathy Yelick, Sue Graham, Phil Colella, Alex Aiken, Luigi Semenzato, Andrew Begel, Dan Bonachea, David Gay, Arvind Krishnamurthy, Ben Liblit, C. J. Lin, Carleton Miyamoto, Simon Yau, Greg Balls, and Peter McQuorquodale. Thanks also to Rich Vuduc, Josh MacDonald, Jim Demmel, Lior Pachter, David Culler, Ed Wang, Kinson Ho, Doug Hauge, Allen Downey, Adrian Perrig, Monica Chew, Raph Levien, Kathryn Crabtree, Peggy Lau, and Mary Byrnes. Finally, thanks to John Heimbaugh and the rest of the local ultimate players—a fun, spirited group.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract F30602-95-C-0136, the National Science Foundation under grants ACI-9619020 and EIA-9802069,and the Department of Energy under contracts W-7405-ENG-48 and DE-AC03-765F00098. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Goals

Taking advantage of the hardware's features is a common goal for performance programmers. Impressive performance gains can arise from clever use of, for example, caches, vector instructions, or sticky flags to detect floating point errors. One of the goals of modern compiler technology is to thrive on ever more complex hardware without sacrificing code's readability, portability, or high-level abstractions.

We reviewed a number of some scientific programs several years ago to see what optimizations, known or unknown, would have the most impact. Unnecessary cache misses were the single largest missed opportunity. Rewriting and restructing the important loops by hand increased the performance of these programs by a factor of two or more. Multigrid [8], in particular, was an important scientific application that we found could greatly benefit from memory-hierarchy optimizations. Other hand-optimization work also suggested that compilers were doing a mediocre job on sequential multigrid (Douglas et al. [11]; Sellappa and Chatterjee [33]).

The main purpose of this document is to explain how state-of-the-art techniques in those hand-optimization experiments were automated. The resulting system is innovative for its ability to tile and fuse loops in a natural and non-restrictive way. When multiple loops are tiled and fused together, the opportunities for data reuse can be great. However, the number of possible reorderings is usually unbounded. We therefore propose some heuristics and report how they compare to each other and to previous approaches.

We rearrange code without replicating operations or performing algebraic transformations, so the optimized code and the non-optimized code perform the same operations on the same operands. The only exception is on hardware where storing a floating-point number to memory and reading it back can result in a loss of precision: in that case, we lose precision less often.

We favor trial-and-error techniques for selecting tile shapes and other parameters. We argue that to be most useful, the system must include a mechanism for choosing parameters outside the compiler itself. Projects such as PHiPAC [7], ATLAS [36], and BeBOP [6] have demonstrated that trying many different combinations of parameters often leads to a "sweet spot" in performance that never would have been predicted analytically. We describe a parameter search mechanism we have developed based on simulated annealing.

1.2 Languages

The primary computer languages of interest in this dissertation are Titanium, our source language, and a C++-like pseudocode we use to describe compiler algorithms. A self-contained (but incomplete) description of Titanium appears in Chapter 3. Hilfinger et al. [14] is the reference manual for Titanium. The main ideas of this dissertation are applicable to any language commonly used for scientific programming. The bulk of the work described below is encoded in a library that could be attached to compilers for other languages.

Pseudocode generally uses `typewriter` font but deviates from that for some variables (such as N) whose value corresponds to one that is discussed in the text. We will occasionally omit the type of a variable if it is obvious from context. A few special notations in pseudocode will be introduced later as needed.

1.3 Concepts and Notation for Tiling

Loop reordering is a potent but complex tool. Compilers reorder loops primarily to improve temporal locality of data accesses. (Vectorizing and parallelizing compilers may reorder loops for other reasons.) Typically memory accesses, cache accesses, cache misses, and number of machine instructions executed all decrease. However, register pressure and code size typically increase. Loop tiling is a well-studied reordering optimization that subsumes most other practical reordering optimizations.

We restrict our discussion to loops of the form `foreach (`p `in` D`) S`, where the iteration space, D, is an arbitrary subset of $\mathbb{Z}^N$; `S` is any statement; and N is a compile-time constant. This is the primary form of iteration in Titanium. The semantics of `foreach` specify that the body, `S`, be executed $|D|$ times with p bound to each element of D in turn. However, the order in which p iterates through D is unspecified.

Let a *tile space* T with *tile size* K be the cross product of $\mathbb{Z}^N$ and the set $\{0, \ldots, K-1\}$. The k^{th} *step* of the *tile at* x is the point $\langle x, k \rangle \in T$, where $x \in \mathbb{Z}^N$ and $0 \leq k < K$. The *tile at 0* means the tile at $[0, \ldots, 0]$. We specify a loop reordering by a total order on T and a bijection, $C\colon T \leftrightarrow \mathbb{Z}^N$. For q and q' in tile

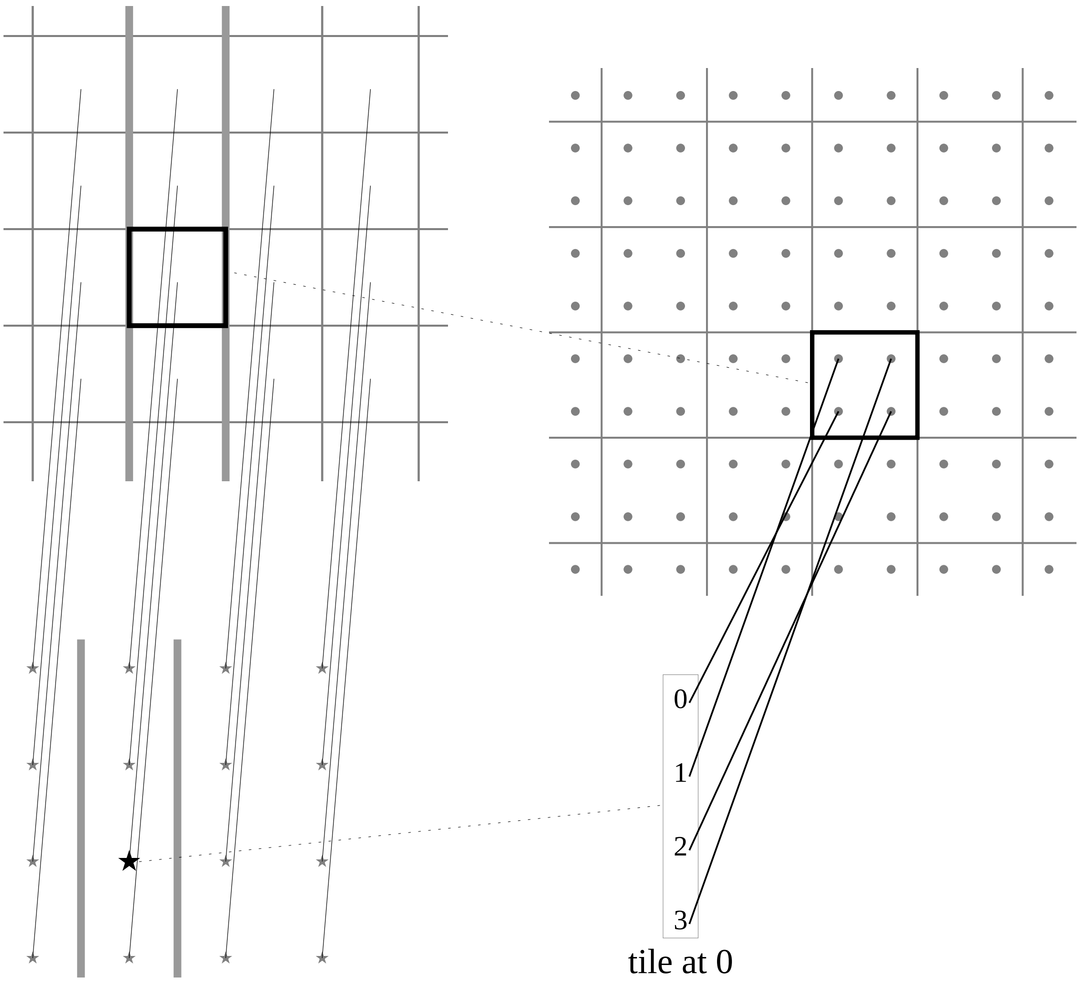

Figure 1.1: Two views of a tiling with a 2x2 square tile. The tile at 0 is highlighted throughout. (a) Top: square tiles in the iteration space. (Individual iterations not shown.) Bottom: $T = \mathbb{Z}^2 \times \{0, 1, 2, 3\}$ is shown with a star indicating each tile. The correspondence between the two spaces is roughly indicated. In both spaces, the area between the heavy gray lines is one stack of tiles. (b) A more detailed view that shows individual steps. Top, the iteration space of the loop is shown with each loop iteration drawn as a small dot. For simplicity, below we only show the tile at 0 (i.e., $\langle[0, 0], 0\rangle \ldots \langle[0, 0], 3\rangle$). Part of the correspondence, C, between the top and bottom is indicated. (Tile stack not shown.)

space and $C(q)$ and $C(q')$ in the loop's iteration space, $q \prec q'$ iff $C(q) \prec C(q')$. A *tile* is a set of points $\langle x, 0\rangle, \ldots, \langle x, K-1\rangle$ in T or the corresponding set in the loop's iteration space, $C(\langle x, 0\rangle), \ldots, C(\langle x, K-1\rangle)$. The tile at x and the tile at x' are in the same *stack of tiles* iff the first $N-1$ coordinates of x and x' are equal.

As an example, consider tiling a 2-dimensional space with 2×2 square tiles. One could use $C(\langle x, 0\rangle) = 2x$, $C(\langle x, 1\rangle) = 2x + [0, 1]$, $C(\langle x, 2\rangle) = 2x + [1, 0]$, and $C(\langle x, 3\rangle) = 2x + [1, 1]$. (We use the notation $[u_1, \ldots, u_N]$ for a vector $u \in \mathbb{Z}^N$.) This is illustrated in figure 1.1.

Our implementation always uses lexicographic order on T. In lexicographic order, $p \prec q$ if $p_1 < q_1$ or if $N > 1$ and $p_1 = q_1$ and $[p_2, \ldots, p_N]$ lexicographically precedes $[q_2, \ldots, q_N]$. Lexicographic order on $\mathbb{Z}^N \times S$ for some $S \subseteq \mathbb{Z}$ is analogous to lexicographic order on $\mathbb{Z}^{N+1}$.

In this model, no constraints are placed on the bijection C, but our implementation only can generate a subset of all the possible bijections. The details are explored in Chapter 4. Among the reorderings we generate are the standard ones for loop interchange, unrolling, reversal, and so on.

Observation 1.3.1 *For $C\colon T \leftrightarrow \mathbb{Z}^N$, any step $0 \leq \alpha < K$, and a vector $\xi \in \mathbb{Z}^N$, one may define another bijection*

$$\Gamma(\langle x, k\rangle) = \begin{cases} C(\langle x + \xi, k\rangle) & \textit{if } k = \alpha \\ C(\langle x, k\rangle) & \textit{otherwise} \end{cases}$$

to generate another (possibly identical) way to tile $\mathbb{Z}^N$. For example, if $C(\langle x, 0\rangle) = 2x$, $C(\langle x, 1\rangle) = 2x + [0, 1]$, $C(\langle x, 2\rangle) = 2x + [1, 0]$, $C(\langle x, 3\rangle) = 2x + [1, 1]$, $\alpha = 0$, and $\xi = [1, 1]$, we get $\Gamma(\langle x, 0\rangle) = 2x + [2, 2]$, $\Gamma(\langle x, 1\rangle) = 2x + [0, 1]$, $\Gamma(\langle x, 2\rangle) = 2x + [1, 0]$, and $\Gamma(\langle x, 3\rangle) = 2x + [1, 1]$. (See figure 1.2.) All the tilings we use have tiles that are parallelepipeds or can be related to such a tiling by one or more applications of this observation.

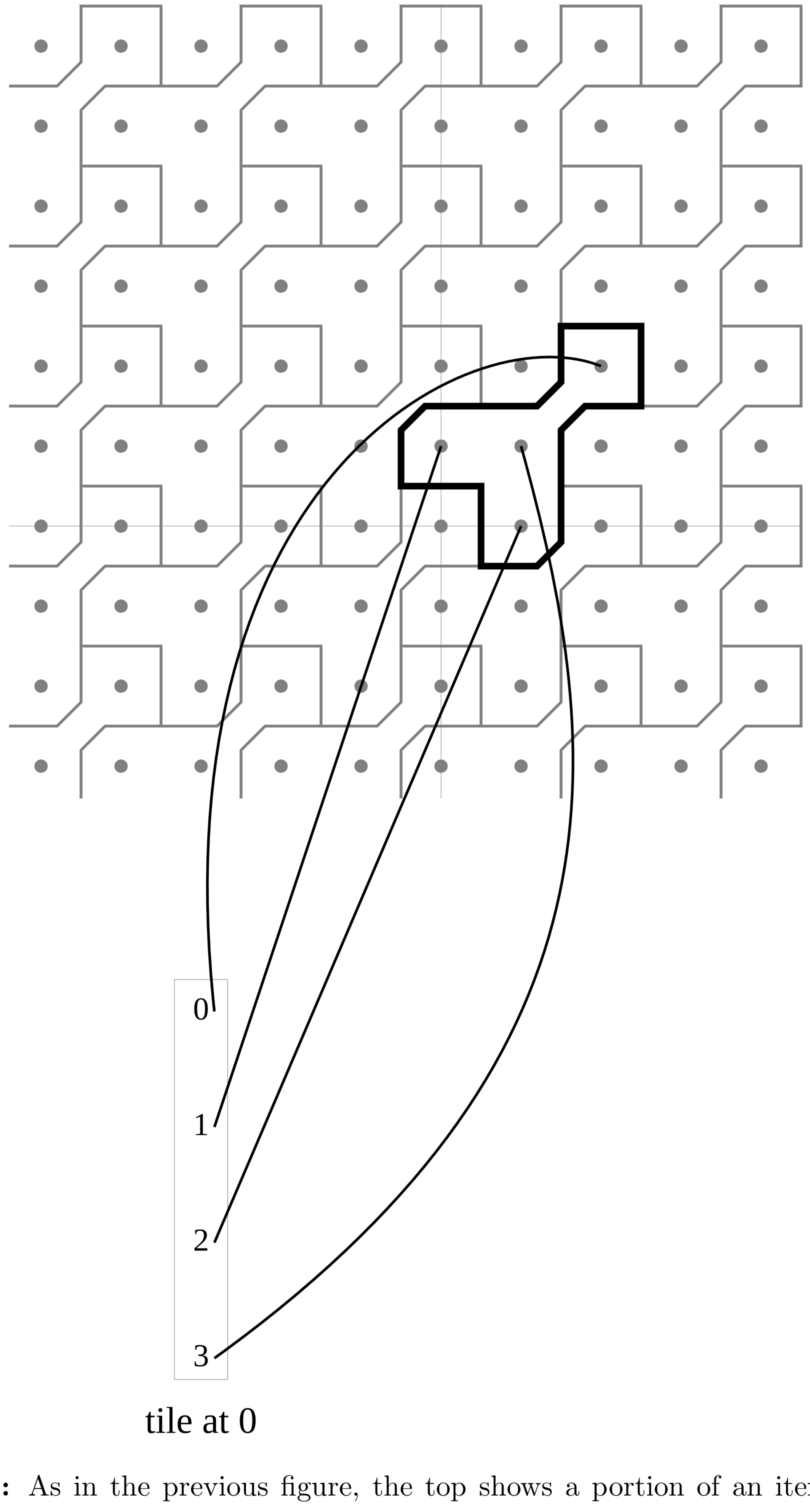

Figure 1.2: As in the previous figure, the top shows a portion of an iteration space with tiles outlined. The tile at 0 is highlighted. Below is $T = \mathbb{Z}^2 \times \{0, 1, 2, 3\}$, with only the tile at 0 shown. Part of the correspondence between the two spaces is indicated.

```
L0: foreach (p in D)
      B[p] = (A[p - [1]] + A[p] + A[p + [1]]) / 3.0;
L1: foreach (q in D)
      A[q] = B[q];
```

Figure 1.3: Sample data parallel code

1.4 A Natural Heuristic for Interleaving Execution of Loops

Tiling and interleaving the execution of multiple loops can have benefits similar to the benefits of tiling a single loop. In particular, loops may share data whose temporal locality can be improved by merging the loops.

The definitions in the previous section may be extended to the multiple-loop case. Let the loops be $L_0, \ldots, L_{n-1}$, each with an N-dimensional iteration space. Define T as before. Let $l(k) = i$ if the k^{th} step *belongs* to loop L_i. Define C_i as a bijection from $\{\langle x, k\rangle \mid l(k) = i\}$ to the iteration space of L_i. Define C as

$$C(\langle x, k\rangle) = \begin{cases} C_0(\langle x, k\rangle) & \text{if } l(k) = 0 \\ C_1(\langle x, k\rangle) & \text{if } l(k) = 1 \\ \vdots & \vdots \end{cases}.$$

Consider optimizing the 1-dimensional example in figure 1.3. An obvious reordering would be to execute each iteration of the second loop shortly after the corresponding iteration of the first loop. For example, for some value z, one could use:

$$\begin{aligned} T &= \mathbb{Z} \times \{0, 1\} \\ l(0) &= 0 \\ l(1) &= 1 \end{aligned}$$

$$
\begin{aligned}
C_0(\langle x, 0\rangle) &= x \\
C_1(\langle x, 1\rangle) &= x - z \\
\text{order on } T &= \text{lexicographic} \quad .
\end{aligned}
$$

Selecting $z = 0$ would not work as then we would be overwriting the value for `A[`x`]` in step $\langle x, 1\rangle$ (second loop, `q` $= x$) that is needed by step $\langle x+1, 0\rangle$ (first loop, `p` $= x + 1$). But any $z > 0$ is legal. Of those possibilities, the best is almost certainly $z = 1$; values are then reused in quick succession. Thus, each value produced by the first loop can be consumed by the second loop without being stored in memory at all.

The heuristic is that one should interleave loops as tightly as possible (if one is going to interleave them at all). We feel this is a very natural heuristic to use in selecting a tiling of multiple loops. We show that following the heuristic is often profitable even if it dictates creating tiles that are not parallelepipeds.

1.5 Storage Optimizations

By *storage optimizations* we simply mean optimizations that change where data is stored. Storage optimizations include basic data reuse optimizations (e.g., do not load from memory what is already in a register) as well as more complex machinations (e.g., rearrange the layout of an array). The storage optimizations we have implemented include a few of the former and one of the latter, namely, array contraction. All of the storage optimizations that we have implemented could be called memory-hierarchy optimizations.

There is no inventiveness in thinking that one should not load from memory what is already in a register. However, our version of array contraction is better than any previously available, because it allows arrays to be contracted piecewise to any combination of scalar variables and lower dimensional arrays.

Finally, as a practical matter, without storage optimizations, it would be difficult even to approach top performance. The comparison of alternative tilings is less meaningful if the low-level implementations of the register tiles are flawed.

1.6 Searching the Space of Parameters

Selecting optimal tile shapes and sizes is too hard for a compiler. The best performance seldom can be achieved in a fixed amount of programming time or compile time, especially now that even low-end CPUs have multiple floating point pipelines, at least two levels of cache, out-of-order execution, and more.

Our system does not necessarily guess parameters well, but instead provides a large number of knobs that override its guesses. The system allows an *input parameter file* alongside the source files. A question answered by the parameter file might be “How large should be the tiles for the loop at line 27?” Correct semantics of the generated code are guaranteed regardless of the parameters specified. The output of the compiler also includes an *output parameter file* indicating what questions it needed answered during compilation and what answers it used. Often a question it needs answered is not addressed in the input parameter file, in which case a default value is selected. If a question answered in the input parameter file is not relevant to the compilation then it is ignored. In our pseudocode, the notation

$$\texttt{get parameter}_d \; \texttt{"How much?"}$$

is used to indicate that the input parameter file is consulted for the answer to the question. The default is d.

Our method for achieving high performance is to test a variety of different parameter settings. The best way to take advantage of the multitude of knobs on our compiler is unclear. We describe our first attempt, a parameter-searching system based on simulated annealing. The longer a search one is willing to undertake the better the answer that will emerge (on average). One can easily imagine other methods of exploring the parameter space.

1.7 Code Bloat

The best-performing scientific programs are not succinct. The best 10,000-line program to multiply matrices will perform better than the best 100-line program to multiply matrices. However, a compiler might transform a succinct program into one approaching optimal performance if it can perform the right optimizations.

Our goal in this work to produce fast sequential programs without regard to the size of the generated code. If code bloat is a concern, there are two mitigating factors. First, the full set of optimizations we propose is typically applicable to a small subset of the code in a program. Therefore, some small sections may grow tremendously in size, but the whole program usually will not. Second, our parameter-searching mechanism incorporates a user-defined fitness criterion. The amount of time to perform some calculation is the usual criterion, but other considerations such as code size can easily be included.

1.8 Contributions

The contributions of this dissertation include:

- Better array contraction than has been done previously in a compiler—on par with the best hand-optimization techniques.

- A flexible tiling and fusion algorithm that can generate tiles in a variety of shapes and sizes.

- A small step towards a system that can equal PHiPAC or ATLAS for any source program.

- Results of searching the parameter space for multigrid and related codes.

- An elegant algorithm to simultaneously determine loop-invariant expressions and, when possible, polynomials equal to certain expressions inside a loop. The resulting information drives a number of transformations.

1.9 Outline

In this introduction we have presented a simple formalism for specifying the tiling and interleaving of any number of loops. It can express all the standard loop reordering optimizations, such as loop fusion, loop interchange, unrolling, reversal, and others.

We take an aggressive approach because we know that these optimizations, when successful, can be tremendously beneficial to performance. We are therefore quick to add transformations when favorable conditions *may* exist at runtime. We do not require complete evidence of safety at compile time if an inexpensive runtime check can enforce safety. We do not worry about array bounds or loop bounds at compile time when we can speculatively compile for some favorable situation and easily test for it at runtime. We do not worry about code bloat.

Clearly, such aggressiveness would be cavalier without the fine control afforded by the input parameter file. We cater to users who are sensitive to performance and willing to tune parameters. Since we expect users to tune the parameters, we are free to include transformations that traditional compilers disdain; even a transformation that usually is a pessimization might be worth including if it has an occasional spectacular success. Missteps are less costly when they will likely be undone by moving elsewhere in the parameter space. In short, it is better to optimize aggressively (but under the user's control) than to refuse lucrative transformations because some crucial information is not available statically.

Here ends the introduction. Chapter 2 discusses related work. Chapter 3 provides basic information on Titanium and `tc`, a compiler for Titanium. A library attached to the Titanium compiler performs the necessary analysis and transformations for our vision of loop reordering and storage optimizations. The library is called "Stoptifu," coined from the first few letters of words in "storage optimizations, tiling, and fusion." Chapters 4, 5, and 6 elaborate on the design and implementation of Stoptifu. Chapter 7 discusses our parameter-searching system. Sequential results are presented in Chapter 8, and a proof-of-concept extension to automatic parallelization is presented in Chapter 9. Finally, Chapter 10 concludes.

Chapter 2

Related Work

2.1 Introduction

This chapter puts our work in perspective amongst the many research projects related to tiling. The best optimizations currently known for multigrid are suggested by studies that rewrite the code manually (Douglas et al. [11]; Sellappa and Chatterjee [33]). Our work does the best job of performing the necessary analyses and transformations in a compiler. We therefore claim that our compiler is the best currently available for sequential multigrid. (Of course, we also can optimize other programs.)

The rest of the chapter explores related manual optimization studies, compiler technology, and parameter searching programs.

2.2 Optimization by Hand

No other compiler offers the right optimizations for stencil codes such as multigrid. We show this by comparing our full compiler with hamstrung versions, some of which correspond to competing approaches (§8). Our aggressive approach to loop fusion and array contraction offers a clear advantage over previous systems. Studies of manually rewritten source programs, however, have demonstrated optimizations that are quite similar to what we have done in Stoptifu.

Hierarchical Tiling: A Methodology for High Performance by Carter et al. [10] is a seminal work. They emphasize careful construction of tiles for register reuse, the notion that data dependences through the surface of a tile have different storage requirements than dependences within a tile, the use of non-rectangular tiles, and, of course, hierarchical tiling. Although we have not implemented hierarchical tiling (yet), we have tried to use many other ideas from their work.

Manual optimization of sequential multigrid and related codes is primarily focused on moving each datum from or to memory as infrequently as possible (Douglas et al. [11]; Sellappa and Chatterjee [33]). Most intermediate results are written once and read once, and in naïve code those data go all the way to main memory and back. Multiple passes of, for example, Gauss-Seidel relaxation, can be merged together, which allows the majority of the intermediate results never to leave the CPU. A write to memory followed much later by a read from memory becomes a write to and read from a register, thereby reducing cache and memory usage and the dynamic instruction count. Fusing loops and optimizing the use of storage is precisely the focus of our compiler work.

2.3 Compilers' Tiling and Storage Optimizations

The compilers that perform tiling are too numerous to list. The SUIF project and its relatives have contributed much over the years, including the award-winning 1991 paper by Wolf and Lam [37] and *Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning* by Lim et al. [29]. The SUIF system is one of several capable of representing the reorderings that our system can generate, but under what circumstances it would select them is unclear.

There are three main differences between the SUIF projects and our work. First, their primary focus is on automatic parallelization whereas Titanium is an explicitly parallel language. We start from the assumption that any parallel program or sequential program should have highly efficient sequential code at its heart. The key to our approach is to fuse loops as tightly as possible; the key to their approach is to minimize the degree of synchronization in automatically parallelized code [28]. Second, we do more array contraction. SUIF does array contraction only in the case where the array can be contracted to a scalar variable [29]. Both their own work and the aforementioned manual optimization work indicates that a more general form of array contraction, such as the one we have implemented, is preferable. Third, we subject many compiler parameters to external control. Parameters include tile sizes and what optimizations to apply, among others. SUIF picks parameters statically.

Strout et al. introduce *universal occupancy vectors* as a way to express equivalence classes among an array's indices [34]. If a compiler can determine that at any time only one array element per equivalence class can be live, then the array can be contracted. The fundamental difference between our approach and theirs is that they assume tiling is going to be done after array contraction. Therefore, they take pains to avoid introducing any new dependences during array contraction, and that adds extra complexity. As a result, their compiler has to solve (or approximately solve) an NP-complete problem. Even worse, their approach is limited to contracting one dimension of an array, because two array indices are in the same equivalence class if and only if they differ by some multiple of the universal occupancy vector.

Thies et al. [35] describe a unified mathematical framework for analyzing the tradeoffs between parallelism and storage allocation in a parallelizing compiler. Their work is based on Strout et al.'s universal occupancy vectors, and shares some of the same limitations. In particular, their system can at best contract one dimension of an array.

Incidentally, even in the case where contracting one dimension of an array is theoretically optimal, our system can often contract *most* of the array to a fixed number of scalars. This is explained in §6, and illustrated in figure 6.2.

Kodukula et al. describe a system that fuses loops aggressively and performs hierarchical tiling [23]. However, their system works by grouping all loop iterations that touch a given parallelogram of a given array. This can work well for dense linear algebra. Unfortunately, it relies on an assumption that one array can provide a one-to-one mapping to—and thus a tiling of—all loops' iteration spaces. Therefore, as they mention, their system cannot handle stencil codes such as Jacobi or Gauss-Seidel. Another difference is that we place more emphasize on storage optimizations.

Flynn Hummel et al. advocate "fractiling," a method that uses tilings with many different tile sizes [16]. For simplicity we have only investigated tilings that tile all space with a single tile.

2.4 Parameter Searching

A compilation system that does not expose parameters for tuning is necessarily suboptimal, because no compiler that takes finite time can always guess the best parameters for all programs. Projects that use parameter searching include PHiPAC [7], ATLAS [36], Sparsity (Im [17]; Im and Yelick [18]), BeBOP [6], FFTW ([12]), and "iterative compilation" (Kisuki et al. [21]; Kisuki et al. [22]; O'Boyle et al. [31]).

PHiPAC and ATLAS automatically generate numerous variants of matrix multiply or other kernels in an attempt to select the best one for a particular task on a particular machine. These systems are instructive for both their absolute success and their relative success. The former, now measured in GFLOPS, competes admirably with experts' hand-written assembler. The latter, their success versus traditional compilation techniques, is also striking. PHiPAC and ATLAS use hand-crafted templates for handling edge cases, copying data, prefetching, selecting regions of parameter search space, and so on. That, and the tremendous advantage in code generation speed, make it difficult for a general-purpose compiler to keep up.

As far as tuning parameters in a compiler, Bodin, Kisuki, O'Boyle, and others have demonstrated that "iterative compilation" is a valuable technique. One interesting result of theirs is that simulated annealing and at least four other search techniques that they tried all have about the same performance characteristics. We have been reasonably happy with simulated annealing and their efforts make us feel even more comfortable going forward. Their work on combining the brute force of parameter searching with modelling techniques is a possible future direction for us.

Chapter 3

Background on Titanium

3.1 Introduction

Titanium is a dialect of Java for high-performance computing (Hilfinger et al. [14]; Yelick et al. [41]). This chapter introduces some salient features of the language. We also describe basic loop analyses and optimizations in `tc`, an optimizing compiler for the Titanium language. The rest of the backend is just briefly outlined as it is only marginally relevent to this dissertation.

Titanium includes many features designed to facilitate scientific programming. Most prominent are a global address space with a distinction between local and global pointers in the type system, SPMD parallelism, unboxed objects, templates, operator overloading, new data types for points and sets in $\mathbb{Z}^N$, statically-enforced synchronization constraints [1], and region-based memory management with garbage collection [13].

The most important new types are: multidimensional arrays; `Point<N>`, an N-tuple of integers; `Domain<N>`, an arbitrary set of N-tuples of integers; and `RectDomain<N>`, a set of N-tuples of integers that can be specified by a lower bound, upper bound, and stride in each dimension. The domain of an array is a `RectDomain`, and arrays are indexed with `Point`s. The operations on `RectDomain`s and `Domain`s are typical set operations, such as `D + E` to compute the union of `D` and `E`. The components of Points may be accessed via array notation; the first component of `p` is `p[1]`.

To facilitate optimization, an array-bounds violation is a fatal error that causes the program to halt when it occurs *or before*. In particular, we need not execute any part of a loop if we detect that an array bounds violation must occur in the loop. Similarly, a number of errors that would just lead to an exception being thrown in Java may, in Titanium, cause undefined behavior or halt the program.

Titanium's memory consistency model allows reordering optimizations as long as dependencies are preserved, even if the reorderings could be observed by other threads. To preserve a stronger, sequentially-consistent semantics, we would need to add cross-thread analysis such as that described by Krishnamurthy [24].

3.2 Overview of `tc`

There exists one compiler for Titanium: `tc`. Its design goals are performance of generated code, portability, extensibility, and ease of implementation. We compile Titanium to an intermediate language that represents a program as an abstract syntax tree (AST). Most intermediate language statements resemble standard three-address codes; control constructs such as `for`, `while`, `break`, `continue`, and `?:` are rewritten to use `if` and unconditional branches. Code in intermediate form is optimized and converted to C. We rely upon the C compiler for register allocation, instruction scheduling, and some optimization. The running time of `tc` is not an issue (unless tiling with very large tile sizes) because the C compiler and linker following `tc` take much longer. Current target platforms include the IBM RS/6000 Large Scale Servers (formerly known as SP-2 and SP-3), Tera Multithreaded Architecture, Cray T3E, Sun/Solaris SMPs, IRIX SMPs (including the Origin 2000), Linux SMPs, SMP clusters, networks of workstations, and uniprocessors.

Analyses and optimizations in `tc` include inlining, control flow analysis, use/def analysis, finding loop invariant code, finding how certain integer-valued expressions change in loops, strength reduction, lifting loop-invariant code, lifting bounds checks from loops, dead-code elimination, Stoptifu, and more. The basic optimizations were selected to achieve reasonable performance. For most (ideally all) Titanium programs we want performance comparable to or better than the equivalent C or FORTRAN program. More advanced optimizations—which are not fully explained herein—were selected according to the research interests of group members. Local qualification inference (Liblit and Aiken [26]), data sharing inference (Liblit, Aiken, and Yelick [27]), statically-enforced synchronization constraints (Aiken and Gay [1]), and region-based memory management (Gay and Aiken [13]) are orthogonal to this dissertation.

The rest of this chapter describes the analysis and transformation of loops in detail (excluding Stoptifu). Of particular note is our new algorithm for simultaneously finding loop invariant code and finding how certain integer- and `Point`-valued expressions change (§3.3.1). Code transformations on loops are discussed in §3.4. An important and unusual transformation is *offset strength reduction*, or the reduction in strength of an address calculation to a constant offset plus some pointer (§3.4.3). Finally, we describe the elimation of useless assignments (§3.5).

3.3 Analysis of Loops

We want both to offer our users performance similar to that of other languages and to offer researchers a platform for implementing new, experimental optimizations. The latter goal requires accurate analysis of array indices in loops. The bulk of the optimizations we do are loop optimizations because our focus is scientific programming.

The primary form of iteration in Titanium is `foreach (`p `in` D`) S`, where the iteration space, D, is an arbitrary subset of $\mathbb{Z}^N$; `S` is any statement; and N is a compile-time constant. The semantics of `foreach` specify that the body, `S`, be executed $|D|$ times with p bound to each element of D in turn. However, the order in which p iterates through D is unspecified.

Although `foreach` loops can iterate over `Domain`s or `RectDomain`s, we often assume that iterations are over `RectDomains`. A `Domain` is currently implemented as a list of `RectDomains`. A loop

```
foreach (p in D) { ... }
```

for a `Domain D` is implemented as

```
for every RectDomain R in the list comprising D
  foreach (p in R) { ... }
```

which is eminently correct, though perhaps not best. A version of `foreach` that is guaranteed to iterate through a `Domain` in lexicographic order has been discussed but not yet added to Titanium.

Given that we compile to C one should ask why `tc` needs loop optimizations, such as strength reduction, that are contained in every C compiler. Alas, Titanium's data types are not analyzed very well by most C compilers. Furthermore, we may have more information (or can better use what information we have) than the C compiler. The complications due to Titanium's high-level array type are discussed further in the section on strength reduction (§3.4.2).

3.3.1 MIVE and Loop Invariant Expressions

We present a novel loop analysis algorithm that provides information for all of our loop optimizations. The output of the algorithm is two-fold: a set of loop-invariant expressions, and polynomials describing the value of expressions in terms of the values of induction variables and loop invariants. The former output is self-explanatory.

The purpose of each polynomial is to represent the value of some expression in a loop. We attempt to create a polynomial for every `int`-valued expression inside a `foreach` loop. All polynomials are scalar and are in terms of 32-bit constants and induction variables (`int`s). The induction variables are the enclosing loops' iteration variables, while the constants are either known integers at compile time or expressions of type `int` that are loop invariant with respect to the innermost enclosing loop. Example:

```
foreach (p in D) { a[p * [2, 3]] = 42; }
```

Here, the expression `p` has type `Point<2>` and we do a component-wise multiplication with `[2, 3]`, another `Point<2>`. As all polynomials are scalar we compute a separate polynomial for each component of a `Point`-valued expression. In the example, $2p_1$ and $3p_2$ would be the polynomials for `p * [2, 3]`.

Our representation of polynomials is straightforward. A polynomial is a sum of terms; a term is a product of an integer coefficient and zero or more factors; and a factor is a symbolic variable representing an induction variable or a loop invariant expression. Polynomials are always simplified to a canonical form.

$$
\begin{array}{rcl}
\texttt{3} & \rightsquigarrow & 3 \\
\texttt{p[1] + 3} & \rightsquigarrow & p_1 + 3 \\
\texttt{p} & \rightsquigarrow & \langle p_1, p_2, \ldots, p_n \rangle \\
\texttt{i} & \rightsquigarrow & i \text{ or } \langle i_1, i_2, \ldots, i_n \rangle \\
\texttt{p + q} & \rightsquigarrow & \langle p_1 + q_1, p_2 + q_2, \ldots, p_n + q_n \rangle \\
\texttt{q + p} & \rightsquigarrow & \langle p_1 + q_1, p_2 + q_2, \ldots, p_n + q_n \rangle \\
\texttt{p * i} & \rightsquigarrow & \langle i_1 p_1, \ldots, i_n p_n \rangle \text{ or } \langle i p_1, \ldots, i p_n \rangle \\
\texttt{p + p} & \rightsquigarrow & \langle 2p_1, 2p_2, \ldots, 2p_n \rangle \\
\texttt{p * p} & \rightsquigarrow & \langle {p_1}^2, {p_2}^2, \ldots, {p_n}^2 \rangle
\end{array}
$$

Figure 3.1: Sample MIVEs. On the left, source code; on the right, MIVEs. Assume `p` and `q` are `Points` and that `i` is either a `Point` or a scalar. Also, all variables in the examples are either induction variables or loop invariant expressions.

All of our loop optimizations use this analysis. Optimizations that reorder code work better with such detail than with summary information provided by simpler approaches such as dependence vectors.

The data structure in the compiler for handling collections of polynomials is a MIVE, an acronym for "Map from Induction Variables to Expressions." Figure 3.1 gives a few examples. A MIVE for an integer-valued expression is a scalar polynomial; a MIVE for a `Point<N>`-valued expression is an ordered sequence of N polynomials. A *MIVEcontext* is a set of variables that can appear in the polynomials. There is one MIVEcontext per loop and it initially contains only variables for each dimension of the iteration point of the loop. We will use p_k to represent the variable for the k^{th} dimension of `p` in `foreach (p in ...) { ... }`. The MIVEcontext grows as integer- or `Point`-valued loop-invariant expressions are discovered.

Although MIVEs can express exquisitely detailed information on many expressions, we did not want to limit our opportunities for invariant code motion to integer- and `Point`-valued expressions. Instead, we separately label some expressions as loop invariant by using simple rules on expression trees (e.g., the sum of invariants is invariant) and flowing results from defs to uses. Combining the calculations of MIVEs and loop invariants into a single algorithm works nicely (figure 3.2). In pseudocode we use `TreeNode` as the class of an AST node; `ForeachNode`, `ExprNode`, and so on are subclasses of `TreeNode`.

The algorithm uses a work list consisting of expression nodes that need to be visited. To visit a node means to determine its MIVE and to determine whether it is loop invariant by using a local examination of the node. Whenever information about a node is updated we add nodes that may be affected to the work list. For example, when analyzing

```
void TreeNode::loopAnal():
  for each child, C,
    C->loopAnal();

void ForeachNode::loopAnal():
  for each child, C,
    C->loopAnal();
  foreachAnal(this);

void foreachAnal(ForeachNode *l):
  Worklist W = expressions in l;
  MIVEcontext MC;
  Add to MC a MIVE for each dimension of the iteration point of l;

  while (W is not empty) {
    take t from W;
    find MIVE for t,
     determine whether t is a loop invariant expression
       with respect to l, and add nodes to w as necessary;
  }
```

Figure 3.2: Algorithm for MIVE and loop invariant analysis

```
foreach (p in D) {
    a[p] = b[p + i] * Math.pow(2.0, 3.0);
}
```

(a)

```
foreach (p in D) {
    x = p + i;
    y = b[x];
    z = Math.pow(2.0, 3.0);
    a[p] = y * z;
}
```

(b)

Initialize work list
Introduce p_1 for `p[1]`
Take `D` from worklist; invar: yes
Take `i` from worklist; invar: yes; Introduce i_1; MIVE: $\langle i_1 \rangle$
Take $\texttt{p}^1$ from worklist; invar: no; MIVE: $\langle p_1 \rangle$
Take `p + i` from worklist; invar: no; MIVE: $\langle p_1 + i_1 \rangle$
Take `x = p + i` from worklist; invar: no; MIVE: $\langle p_1 + i_1 \rangle$
Take `b` from worklist; invar: yes
Take $\texttt{x}^2$ from worklist; invar: no; MIVE: $\langle p_1 + i_1 \rangle$
Take `b[x]` from worklist; invar: no
Take `y = b[x]` from worklist; invar: no
Take `Math.pow(2.0, 3.0)` from worklist; invar: yes
Take `z = Math.pow(2.0, 3.0)` from worklist; invar: yes
Take $\texttt{z}^4$ from worklist; invar: yes
Take $\texttt{y}^4$ from worklist; invar: no
Take `y * z` from worklist; invar: no
Take $\texttt{p}^4$ from worklist; invar: no; MIVE: $\langle p_1 \rangle$
Take `a` from worklist; invar: yes
Take `a[p]` from worklist; invar: no
Take `a[p] = y * z` from worklist; invar: no

(c)

Figure 3.3: (a) source code; (b) Intermediate form; (c) list (highlights) of operations performed. Superscripts are for disambiguation and refer to line numbers (1–4) within the body of the loop.

```
b = a + 2;
c = b + 7;
```

we add the node for `b` in the second line to the work list if we discover new information about the first `b`. In addition to following def-use edges, we add the parent of a node to the work list if the parent may be affected by new information about its child. So, if the MIVE for the first `b` changes to $a + 2$ then the use of `b` will get the same MIVE, causing its parent, `b + 7`, to be added to the work list. When `b + 7` is analyzed we will assign it the MIVE $a+9$, and $a+9$ will eventually become the MIVE for `c` and for the assignment `c = b + 7`.

One could calculate loop invariant expressions first and then calculate MIVEs in separate pass, but it is not necessary. Figure 3.3 illustrates the whole algorithm on a simple example. The loop body in figure 3.3 is simple straight-line code and therefore it is only necessary to visit each node once. Even in complex programs, nodes are seldom visited more than a few times. We are able to mark `Math.pow(2.0, 3.0)` as loop invariant because we have built knowledge of certain standard libraries in to `tc`.

3.4 Loop Optimizations

Once we have analyzed a `foreach` loop and all `foreach` loops nested inside it, we are ready to do optimizing transformations. The transformations are familiar but there are some twists in Titanium and in our implementation. One common theme is that when we move code out of a loop it comes all the way out; in most other languages, Titanium's `foreach` on a n-dimensional iteration space is expressed as n nested loops and a compiler might only move code out of the innermost loop. It should be noted, however, that we tend not to move code out of multiple levels of `foreach` loops because we do not have an analysis that determines whether a nested loop has greater than zero iterations. We penalize the programming style that uses nested 1-dimensional loops when an n-dimensional loop could have been used.

Unless otherwise noted, the sections that follow assume n-dimensional `foreach` loops with iteration point `p` whose MIVE is $\langle p_1, \ldots, p_n \rangle$.

3.4.1 Lifting Loop Invariants

The Basic Idea

Even if invariant code motion is turned off, this pass transforms

```
foreach (p in D) { ... }
```

to

```
if (!D.isNull()) { foreach+ (p in D) { ... } }  ,
```

where `foreach+` denotes an iteration known to be non-empty. The `foreach+` is preceded by assignment statements that could legally be moved there, if any. We lose nothing by only moving assignment statements because nothing interesting can really happen in our intermediate form except in an assignment statement. The determination of what may legally be moved out of a loop uses standard techniques (e.g., Aho et al. [3]).

Eliminating Some Redundancy

An important twist is that we eliminate some redundant variables along the way. At present we do not implement global common subexpression elimination (CSE). However, our technique yields some of the benefit of CSE without requiring a separate pass. It was also easy to implement (figure 3.4).

```
// l is the loop; toBeMoved is the list of nodes to be moved.
StatementNode *liftInvariantCode(ForEachStmtNode *l,
                                 list<TreeNode *> toBeMoved):
  list<TreeNode *> result = empty list;
  set<TreeNode *> movedSoFar = empty set;
  bool progress;
  for each c in toBeMoved,
    set pre[c] to the set of nodes that must be moved prior to c;

  set<Patch> patches = empty set;
  do {
    progress = false;
    for each c, whose form is "LHS = RHS", in toBeMoved
      if (c is not in movedSoFar &&
          pre[c] is a subset of movedSoFar) {
        add c to movedSoFar;
        if an assignment with equivalent RHS has already been moved {
          TreeNode *previous = the LHS of that assignment;
          adjoin <LHS, previous> to patches;
          append "LHS = previous" to result;
        } else
          append c to result;
        progress = true;
      }
  } while (progress);

  remove all elements of movedSoFar from l;
  apply patches to l;
  prepend result to l;
  return l;
```

Figure 3.4: Pseudocode for invariant code motion

An important benefit of eliminating some redundancy is best illustrated by example:

```
foreach (p in D) { a[p] = (a[p + [1]] + a[p - [1]]) / 2; }
```

might become

```
foreach (i in D) {
  m = this.a; n = i + [1]; o = m[n];
  p = this.a; q = i - [1]; r = p[q];
  s = o + r;
  t = s / 2;
  u = this.a; u[i] = t;
}
```

in the unoptimized intermediate form. With lifting and redundancy elimination we rewrite it to:

```
if (!D.isNull()) {
  m = this.a;
  p = m; /* Useless. Will be eliminated in subsequent pass. */
  u = m; /* Useless. Will be eliminated in subsequent pass. */
  foreach+ (i in D) {
    n = i + [1]; o = m[n];
    q = i - [1]; r = m[q];
    s = o + r;
    t = s / 2;
    m[i] = t;
  }
}
```

which is much better because having multiple uses of the same variable, `m`, exposes opportunities for subsequent optimization. In particular, if the three array expressions did not transparently refer to the same array, then offset strength reduction (§3.4.3) would not be allowed.

3.4.2 Strength Reduction

Strength reduction is one of the most important optimizations in the history of computing because strength reduction allowed early FORTRAN codes to get performance similar to hand-coded assembly language. But for strength reduction the widespread adoption of high-level languages might have been greatly delayed. Strength reduction of array address calculations simplifies the code generated to access an array. For example, in

```
foreach (p in D) { x += a[p]; }
```

the value of `a[p]` may be compiled into a mere pointer dereference. Of course, the compiler must also insert code to initialize the pointer and update it.

The generality of Titanium's m-dimensional arrays leads inevitably to complicated address calculations. The address of `a[p]` is:

$$b + \sum_{i=1}^{m} \frac{p_i - s_i}{d_i} k_i \quad ,$$

where b and each s_i, d_i, and k_i are integer constants stored in the descriptor for `a`. We require, without loss of generality, that $d_i \geq 1$. We must have a multiplication for each dimension because the distance in memory between `a[1]` and `a[2]` could be a million bytes. We must have a division for each dimension because the distance in memory between `a[1]` and `a[1000]` could be one byte. And we must have a subtraction if we want to make the division come out evenly. (We could eliminate

```
foreach (p in R) { A[...] = 42; }
```

becomes, in C:

```
...
while (x0 != ex0) {
  int *x1 = x0;
  int *ex1 = x1 + lx1;
  while (x1 != ex1) {
    int *x2 = x1;
    int *ex2 = x2 + lx2;
    while (x2 != ex2) {
      *x2 = 42;
      x2 += Δx2;
    }
    x1 += Δx1;
  }
  x0 += Δx0;
}
```

Figure 3.5: Generic strength reduction for a 3D rectangular iteration

the subtraction if we were willing to replace normal division with division that always rounds to $-\infty$. We do the subtraction because it is cheaper on most hardware.)

The division in the address calculation will come out evenly if the index is in the domain of the array and the array descriptor is properly constructed. In fact, Titanium has bounds checking and the application programmer cannot directly construct or modify an array descriptor, so the division *will* always come out evenly. However, any straightforward translation of Titanium into C will not convince the C compiler that the division will always come out evenly. It would have to go to great lengths—beyond what is realistic—to know all that we know. As a result, we cannot rely on the C compiler to strength reduce our address calculations.

We are therefore forced to do some strength reduction of address calculations in `tc`—at least enough to remove the division from the inner loop. Given that we

```
static Point<1> find(int [1d] A, int val) {
  foreach (p in [0 : N])
    if (A[p] == val)
      return p;
  return [-1];
}
```

Figure 3.6: Strength reduction is performed on this method even though the loop is a partial domain loop. The code to set up the pointer for accessing `A[p]` and its increment must be prepared for the array being null or for the domain of the array being a proper subset of the iteration domain. The code for the body of the loop will include a bounds test.

must do that much and that we have better information, we decided to go all the way.

Our goal is to generate code similar to that in figure 3.5. What we choose to reduce in strength is dictated by our choice to require that the change to a pointer each iteration be integral and iteration-space invariant. Strength reduction is tricky for loops such as

```
foreach (p in D) { if (f()) sum += A[expr]; }
```

because `A[expr]` may be evaulated sporadically or not at all. The pointer update in every iteration might therefore slow the program down if `A[expr]` is seldom used. Furthermore, the change in the address of `A[expr]` per iteration may not be integral. If we did strength reduce the address calculation, we would have to add extra logic beyond what is shown in figure 3.5. Instead, we allow an index expression for a particular array to be strength reduced only in two cases:

1. The expression must index the array in every iteration, and every iteration must occur (barring a fatal error).

2. The expression must index the array in every iteration, except possibly the last runtime iteration, which could be cut short by a `goto`, `return`,

exception, or error.

We call these two cases the *full domain* case and the *partial domain* case; we also classify loops as *full domain* loops or *partial domain* loops, where the former must execute every iteration and cannot be cut short except by a fatal error. For the purposes of strength reduction the cases are essentially the same; they both can be compiled as shown in figure 3.5. The only difference is that in the partial domain case one must ignore certain errors while setting up pointers and increments (see figure 3.6). However, when one wants to lift array bounds checks from a loop, the partial domain case and the full domain case are completely different (§3.4.5).

The requirement that the pointer increments be loop invariant is easily checked given that we have MIVEs. For example, suppose the MIVE for an index expression e is $\langle e_1, \ldots, e_m \rangle$. Then we just check that

$$\delta^i_j(\mathtt{e}) = \frac{\partial e_i}{\partial p_j}$$

is loop invariant for each i and for each iteration space dimension $j = 1, \ldots, n$. That is, we check that

$$\forall^m_{i=1} \forall^n_{j=1} \forall^n_{k=1}, \frac{\partial \delta^i_j(\mathtt{e})}{\partial p_k} = 0 \quad .$$

3.4.3 Offset Strength Reduction (OSR)

Certain programs can benefit from a second kind of strength reduction. Compare the translations of

```
foreach (p in R) { A[p + u] += A[p + v]; }
```

in figure 3.7. In cases where standard strength reduction would lead to multiple pointers moving in lockstep, it is better to strength reduce all but one of those

```
...
while (x0 != ex0) {
  int *x1 = x0, *y1 = y0;
  int *ex1 = x1 + lx1;
  while (x1 != ex1) {
    int *x2 = x1, *y2 = y1;
    int *ex2 = x2 + lx2;
    while (x2 != ex2) {
      *x2 += *y2;
      x2 += Δx2; y2 += Δy2;
    }
    x1 += Δx1; y1 += Δy1;
  }
  x0 += Δx0; y0 += Δy0;
}
```

(a)

```
...
while (x0 != ex0) {
  int *x1 = x0;
  int *ex1 = x1 + lx1;
  while (x1 != ex1) {
    int *x2 = x1;
    int *ex2 = x2 + lx2;
    while (x2 != ex2) {
      *x2 += *(x2 + o);
      x2 += Δx2;
    }
    x1 += Δx1;
  }
  x0 += Δx0;
}
```

(b)

Figure 3.7: (a) naïve translation of code amenable to OSR, (b) translation with one address calculation strength reduced to an offset off another pointer

address calculations to a constant offset from one pointer that does move. In figure 3.7(a), all the calculations involving the $\mathtt{y}_i$'s and $\Delta\mathtt{y}_i$'s are unnecessary if we introduce `o`, the difference between the addresses of `A[p + u]` and `A[p + v]`. Fewer variables are needed, reducing register pressure. Fewer instructions are needed because fewer pointers need to updated. In addition, most architectures allow a load such as (C notation) `r1 = *(r2 + r3)` to be expressed in one instruction that is just as efficient as any other load.

We call this optimization Offset Strength Reduction (OSR). Using MIVEs, it is trivial to determine when it is legal to apply OSR. Suppose `A[e]` and `A[f]` both occur in a loop, `e` and `f` being arbitrary expressions. Let the MIVEs for `e` and `f` be $\langle e_1, \ldots, e_m\rangle$, $\langle f_1, \ldots, f_m\rangle$. OSR is legal if the difference between `e` and `f` is loop invariant and the address of `A[e]` is an available expression at the use of `A[f]`. The former condition is just

$$\forall_{i=1}^{m} \forall_{j=1}^{n}, \frac{\partial}{\partial p_j}(e_i - f_i) = 0 \quad .$$

Whether `A[e]` is strength reduced is irrelevant. To give an unlikely example, one could use OSR in a loop such as

```
foreach (p in D)
  A[p * p] += (B[p] ? A[p * p + i] : A[p]);
```

on the address of `A[p * p + i]`. One can compute the difference between that address and the address of `A[p * p]` and store it in an `int`, knowing that either the difference will be integral or it will never be used.

In our implementation we use OSR in fewer cases than we could; we require that `A[e]` be strength reduced and that `A[f]` would be strength reduced in the traditional manner if not for OSR. In practice, there is little difference between

```
// Determine whether the usage of A in Loop indicates that
// A's stride must be 1 (or -1) in the given dimension.
// S is the set of MIVEs of accesses to A that are strength
// reduced (including OSR).
bool UnitStrideInference(TreeNode *Loop, TreeNode *A, int dim,
                         set<MIVE> S):
  int g = 0;
  for every possible pair of elements in S {
    MIVE d = the absolute value of the difference between
                the pair in dimension dim;
    if (d is a known integer) {
      g = (g == 0) ? d : gcd(g, d);
      if (g == 1)
        return true;
    }
  }
  return false;
```

Figure 3.8: Pseudocode for Unit Stride Inference

our rules and the more aggressive alternatives. We might benefit slightly by using OSR more, but ratio of benefit to cost of implementation is low.

3.4.4 Unit Stride Inference

If an array `A` has unit stride in its i^{th} dimension then we can simplify its address calculations and bounds checks. We have implemented a modest algorithm for inferring unit strides at the level of a `foreach` loop (figure 3.8). For example, it will infer that `A` must have unit stride in this loop:

```
foreach (p in D) { A[p] = A[p + [2]] * A[p + [5]]; }
```

because $\gcd(2, 3, 5) = 1$. We did not bother with a polynomial gcd, though that would have allowed us to handle cases such as:

```
foreach (p in D) { A[2 * p] += A[4 * p + [1]]; }  .
```

It would be nice to transfer the knowledge gained to other uses of `A`, but that would require knowing that the loop has greater than zero iterations. If the loop has zero iterations at runtime then we cannot infer anything about the array's stride.

Let σ_i be the stride of `A` in its i^{th} dimension. When we infer from `A`'s usage in a given loop that $\sigma_i = 1$, we output code in the loop header that aborts if the loop's domain is not empty and $\sigma_i \neq 1$. Then divisions by σ_i and array bounds checks of the form "σ_i must divide x" may be omitted. (Actually, usage can at best imply $\sigma_i = \pm 1$, but strides are always positive in our implementation.)

3.4.5 Lifting Bounds Checks

Titanium's garbage collection and array-bounds checking eliminate most of the bugs that appear in typical C, C++, and FORTRAN programs. Although it can be disabled for speed, we believe that most programmers will want to enable array bounds checking most of the time.

For simplicity, the only bounds checks we optimize are on index expressions that have been strength reduced (including OSR). By our rules, these index expressions are guaranteed to appear on every loop iteration.

We move some array bounds checks to the headers of full domain loops, but each array access in a partial domain loop is individually checked. For the following discussion, assume we are optimizing an array-bounds check for an expression `A[e]` that appears inside a full domain `foreach` loop with iteration point `p` and domain `D`. Let the MIVEs of `e` and `p` be $\langle e_1, \ldots, e_m \rangle$ and $\langle p_1, \ldots, p_n \rangle$. Let the domain of the array `A` be

$$\{(x_1, \ldots, x_m) \mid \forall_{i=1}^{m} (\alpha_i \leq x_i \leq \beta_i \wedge \sigma_i \text{ divides } (x_i - \alpha_i))\} \quad .$$

Let `D`, the iteration space, be

$$\{(y_1, \ldots, y_n) \mid \forall_{j=1}^{n}(\gamma_j \leq y_j \leq \mu_j \wedge \tau_j \text{ divides } (y_j - \gamma_j))\} \quad .$$

Let the changes in `e` as `p` changes be described by

$$\delta_j^i(\mathtt{e}) = \text{the change in } e_i \text{ for a minimal change in } p_j = \tau_j \frac{\partial e_i}{\partial p_j} \quad .$$

All of the δ_j^i's are constant integers because of the requirement that `A[e]` is a strength-reduced index expression. At compile time they may or may not be known integers.

Our strategy for optimizing an array bounds check for `A[e]` is to divide the check in four parts. For each array dimension i:

1. Minimum over `D` of $e_i \geq \alpha_i$.
2. Maximum over `D` of $e_i \leq \beta_i$.
3. σ_i divides the value of $e_i - \alpha_i$ at $(\gamma_1, \ldots, \gamma_n)$.
4. $\forall_{j=1}^{n}$, if $\gamma_j \neq \mu_j$ then σ_i divides $\delta_j^i(\mathtt{e})$.

If all of the above conditions hold for all the dimensions, then `A[e]` is in bounds for the whole iteration. If any are violated, then we halt the program with an appropriate error message.

Of the four tests, we try to omit those that we can. Any dimension for which unit stride is inferred does not need the last two tests; there will just be a run-time check that $\sigma_i = 1$. We can omit one or both of the first two tests if other index expressions are known to be more extreme. For example, suppose `A[e]` and `A[e - [1, 0, 0]]` are both to be checked in the header of a loop. If the

```
// Generate max and min tests for dimension dim of A in Loop.
void generateBoundsChecks(TreeNode *Loop, TreeNode *A,
                          int dim):
  set<MIVE> s = the set of MIVEs used for strength reduced accesses
                to array A in Loop;
  generateMinBoundsTest(Loop, A, dim, mightBeMinimum(s, dim));
  generateMaxBoundsTest(Loop, A, dim, mightBeMaximum(s, dim));

void generateMinBoundsTest(TreeNode *Loop, TreeNode *A,
                           int dim, set<Polynomial> candidates):
  string minUsed = findMin(Loop, A, dim, candidates);
  Emit "assert(minUsed >= α_dim);";

// If any pair of elements differ by a known integer constant then
// the larger of the two need not be included in the result.
set<Polynomial> mightBeMinimum(set<MIVE> s, int dim):
  set<Polynomial> result = empty set;
 outer:
  for each m in s {
    Polynomial p = the polynomial for m in dimension dim;
    for each i in result {
      Polynomial diff = i - p;
      if (diff is a known integer) {
        if (diff > 0)
          Remove i from result and replace it with p;
        continue outer;
      }
    }
    adjoin p to result;
  }
  return result;
```

Figure 3.9: Pseudocode for generating minimum and maximum array bounds tests. Overbar notation indicates insertion of values into a string. We omit `generateMaxBoundsTest()` and `mightBeMaximum()`. Code for `findMin()` is on the next page.

```
// The candidate set contains the polynomials whose
// minimums we need to consider.  Generate code to find the
// minimum of those and return a string that represents the result.
string findMin(TreeNode *Loop, TreeNode *A,
               int dim, set<Polynomial> candidates):
  list<string> possibleMin = empty list;
  for each e in candidates {
    string s = "0";
    for j from 1 to n {
      string sj;
      Polynomial d = ∂/∂p_j e;
      if (d == 0)
        continue;
      else if (d is a known integer)
        sj = "d * (d > 0) ? γ_j : μ_j";
      else {
        string t = code for polynomial d;
        sj = "(t * ((t > 0) ? γ_j : μ_j))";
      }
      s = "s + sj";
      subtract p_j times d from e;
    }
    string leftover = code for polynomial e;
    string v = name of a newly-declared int-valued variable;
    Emit "v = s + leftover;";
    add v to possibleMin;
  }
  return minOfListOfVariables(possibleMin);
```

Figure 3.10: Function to generate code that finds the minimum over a loop's iteration space of all of a set of candidate polynomials. Overbar notation indicates insertion of values into a string.

minimum of $e_1 - 1$ is greater than or equal to α_1 then the minimum of e_1 must be as well. Our technique for generating the minimum and maximum tests is shown in figures 3.9 and 3.10.

Figure 3.10 shows how to generate code for the minimum over `D` of e_i. Given that all of the δ^i_j's are constants, the minimum of e_i must occur at one of the 2^j corners of the iteration space. Which corner is determined by whether e_i is increasing or decreasing with respect to each p_j. For example, the minimum of `3 * p[1] - 5 * p[2] + p[3]` is $3\gamma_1 - 5\mu_2 + \gamma_3$.

3.5 Useless Assignment Elimination

The elimination of useless assignment statements is most important when other optimizations are activated. For example, the program in figure 3.11 contains statements to calculate `v` and `w` that are useless if the array access `A[w]` is strength reduced. If we do not remove those statements then we are at the mercy of the C compiler. Surprisingly, the useless statements, when translated into C, are not always recognized as such. Our data structure for a `Point<N>` is a `struct` containing N integers; our functions manipulating `Points` are declared `static inline`. Somehow that is opaque enough to some C compilers that we must do our own useless assignment elimination.

Given that we must do some elimination of useless assignments, we implemented the simple algorithm shown in figure 3.12. Though many more clever algorithms exist, this one has served us well.

```
foreach (p in D) {
  Point<2> v = p * 2;
  Point<2> w = v + [0, 1];
  A[w] = 11;
}
```

Figure 3.11: A program fragment that can benefit from useless assignment elimination after the array access is strength reduced. Programs expressed in `tc`'s intermediate representation are full of similar examples. If this were a partial domain loop then `v` and `w` would be necessary because a bounds check would be performed on every iteration.

```
// m is a method for which we shall find useless assignments.
// We initially assume that every assignment is useless.
set<TreeNode *> uselessAssignments(TreeNode *m):
  set<TreeNode *> presumedUseless = all assignments in this;
  bool progress;
  do {
    progress = false;
    for each assignment in presumedUseless {
      if (!useless(assignment, presumedUseless)) {
        remove assignment from presumedUseless;
        progress = true;
      }
    }
  } while (progress);
  return presumedUseless;

bool useless(TreeNode *assignment, set<TreeNode *> presumedUseless):
  if (result of assignment is used by
      a statement not in presumedUseless)
    return false;
  else if (assignment has side-effects or may cause
           an exception or error)
    return false;
  else
    return true;
```

Figure 3.12: Pseudocode for finding useless assignments. It assumes we have use-def information.

Chapter 4

On Reordering Loops

4.1 Introduction

4.1.1 Purpose

We are interested in reordering loops primarily as a means to improve temporal locality. Stencil codes such as multigrid are our motivating example. In multigrid many temporary values are only used once, and good performance can be achieved only if the bulk of those temporaries are used and discarded without ever leaving the CPU. That typically requires fusing the loops that produce and consume a given stream of temporaries. The way we fuse loops is driven by dependences, which typically correspond to the flow of data. If the i iteration-space nodes in one loop produce d data that are consumed by j iteration-space nodes in another loop then our dependence driven approach usually produces tiles with a ratio of i nodes of the one loop to j nodes of the other.

The design goals of our reordering engine include generality and parameterization. We also aggressively fuse and tile loops even when runtime tests are necessary to ensure safety. Parameterization allows the aggressiveness to be tempered by striving to satisfy the programmer's fitness criterion. For example, while the methods presented below can fuse any number of loops, the parameterization allows the opportunity to fuse the "right" number.

The space of parameters is searched using a fitness criterion that is unknown to `tc` and Stoptifu. It is usually the time to perform some calculation. Regardless, we chose not to explicitly model register reuse or cache behavior or other factors that tend to correlate with the fitness of a reordering transformation. We felt it was more important to implement the transformations that might be necessary for best results and to hope that a parameter search will eventually find favorable parameters. Inadequate transformations may be impossible to be overcome, but inadequate estimation of parameters' value at worst prolongs the time to reach the best parameters. So for now, we value a set of parameters by compiling, running, and measuring the program being optimized.

4.1.2 Terminology

We often refer to *ordering constraints*. We mean this in a general sort of way, although in our implementation these can only be dependences that arise when two statements may touch the same data, at least one of them writing. The kinds of ordering constraints are therefore read-after-write, write-after-write, or write-after-read. A sample is: "iteration p of loop 0 must precede iteration $p + [1, 0]$ of loop 1 because the latter reads a value that is stored by the former." In general, we store which loops are involved, which pairs of iterations, and the kind of ordering constraint. We use the Omega library [20] to manipulate integer tuple relations and sets. In particular, the Omega library can construct relations and sets using Presburger formulas, which include affine equality and inequality constraints, logical operators $\neg$, $\wedge$, and $\vee$, and quantifiers $\exists$ and $\forall$. Our ordering constraints are usually from one iteration space to another, and although one might view each as just $\mathbb{Z}^N$, it is useful to think of them and manipulate them as separate vector spaces. For example, subtracting a point in one loop's iteration space from a point in another loop's iteration space is meaningless.

If we have executed a subset A of the iteration space of loop f then we can define the *ready set* of loop g with respect to A as the set of nodes in the iteration space of loop g that are not forbidden by any ordering constraints from loop f to loop g. (This is computed by assuming that the complement of A has not executed; only nodes that have a required precedent in $\bar{A}$ are excluded from the ready set.) Using the above sample constraint, the ready set of loop 1 with respect to $\{x \mid x_1 + x_2 < 3\}$ is $\{x \mid x_1 + x_2 < 4\}$.

The notion of ready set can be extended. The ready set of loop g with respect to A and B is the intersection of the ready set of loop g with respect to A and the ready set of loop g with respect to B. (Typically A and B are subsets of different iteration spaces.) Often we just use "the ready set" because the exact meaning will be obvious from context.

4.1.3 Outline

The rest of this chapter describes tiling in Stoptifu, our library. We begin with representation and selection of tilings of space. The crux of the chapter is the algorithm to *induce* a tiling for $n + 1$ loops from a tiling for n loops (§4.4). After exploring that algorithm and some variants, §5 begins with the Stoptifu/`tc` interface. It continues with implementation details related to tiling on the `tc` side of that interface. The detailed treatment of Stoptifu internals concludes with storage optimizations (§6).

4.2 Dividing $\mathbb{R}^N$ into an ordered set of parallelepipeds

As in §1.3 we begin by tiling an N-dimensional space. Our implementation puts no restrictions on the actual size or shape of the iteration space, which in general is not known at compile time.

Tiling a single loop in Titanium is trivial. The Stoptifu library has code to handle ordering constraints within a loop (e.g., due to data dependences). However, `foreach` loops in Titanium, which are unordered, never have such constraints. For the rest of the chapter we will discuss Stoptifu in that context.

An N-dimensional space may be tiled in any number of ways. One way is to select three points in $\mathbb{Z}^N$, ζ, ν, and σ. We will refer to ζ as the *zero*, ν as the *normal*, and σ as the *deriv*. Let $\mathbb{R}^N$ be divided by hyperplanes perpendicular to the line containing ζ and $\zeta+\nu$ through each of $\{\ldots,\zeta-\sigma,\zeta,\zeta+\sigma,\ldots\}$. (We ignore the degenerate cases, i.e., when $\nu = 0$ or all the hyperplanes coincide.)

For the purposes of ordering, a point $p \in \mathbb{R}^N$ may be classified by:

$$c(p) = \left\lfloor \frac{(p-\zeta)\cdot\nu}{\sigma\cdot\nu} \right\rfloor \quad .$$

One may think of $c(p)$ as the number of hyperplanes crossed by a line segment from from p to ζ (counting +1 for a crossing aligned with the normal vector and -1 for a crossing against it). The ordering requires that $p \prec q$ if $c(p) < c(q)$.

Selecting a set of hyperplanes in this fashion reduces the N-dimensional tiling problem to an $(N-1)$-dimensional tiling problem. A second set of parallel hyperplanes may be chosen for the reduced problem by picking a ζ', ν', and σ'. (We again ignore the degenerate cases, i.e., when the normals are not linearly independent, $\nu' = 0$, or hyperplanes coincide.) The ordering now requires that $p \prec q$ if $c(p) < c(q)$ or if $c(p) = c(q)$ and $c'(p) < c'(q)$, where

$$c'(p) = \left\lfloor \frac{(p - \zeta') \cdot \nu'}{\sigma' \cdot \nu'} \right\rfloor \quad .$$

This process may be applied repeatedly to divide $\mathbb{R}^N$ into an ordered set of parallelepipeds that implies a tiling of $\mathbb{Z}^N$ having identical size and shape for all tiles.

Observation 4.2.1 *The volume of the parallelepipeds generated by this technique is unaffected by changes to the zeroes or the normals. The volume is always equal to the volume of a parallelepiped whose edges are congruent to σ, σ', etc.*

4.3 Selecting a Tiling of One Loop

Parameters in the parameter file may be used to select different tilings for a single loop, but, for simplicity, the division of space in our implementation always uses $\zeta = 0$. The exact details are not particularly important; readers may want to skip to the next section.

Figure 4.1 shows pseudocode for picking the sets of hyperplanes. We do not allow all possible tilings because many of them are illegal or unlikely to be useful. In particular, we force each σ to be aligned with the corresponding ν. We also force the last normal selected to be perpendicular to all other normals. Perhaps that is unduly restrictive but, in practice, it works well. If a single loop may have ordering constraints among its iterations, which is not the case in Titanium, then relaxing the method for selecting the last normal is advisable.

```
Tiling pick_planes(Loop loop, list<OrderingConstraint> oc):
  queue<vector> Q = vectors in $\mathbb{Z}^N$ sorted by taxicab order;
  Discard from Q any vector that is a positive multiple of
   a vector that precedes it;
  list<set of parallel hyperplanes> hlist = empty list;
  list<vector in $\mathbb{Z}^N$> normallist = empty list;
  for i from 0 to $N$ - 2
    int skip = get parameter$_0$ "Skip how many normals?";
    do {
      do {
        pop Q;
      } until (front(Q) is not forbidden because of oc and
               front(Q) is linearly indep. of previous normals);
      skip = skip - 1;
    } until skip < 0;
    Append front(Q) to normallist;

  permutation p = get parameter$_{\text{identity}}$ "Permutation of $\{1,\ldots,N-1\}$?";
  for i from 0 to $N$ - 2
    $\nu$ = element i of p(normallist);
    int spacing = get parameter$_3$ "Spacing?";
    $\sigma$ = the pos. multiple of $\nu$ with length closest to spacing;
    h = hyperplanes with normal $\nu$ through $\{\ldots,-\sigma,0,\sigma,\ldots\}$;
    Append h to hlist;

  vector last = some vector perpendicular to all normals in hlist;
  last = last / $\gcd(\texttt{last}_1,\ldots,\texttt{last}_N)$;
  if (oc indicates that last is illegal) {
    last = -last;
    if still illegal fail;
  }
  Let $\nu$ = last;
  Let unroll = get parameter$_0$ "Unroll?";
  Let $\sigma$ = (1 + unroll) * $\nu$;
  Let h = hyperplanes with normal $\nu$ through $\{\ldots,-\sigma,0,\sigma,\ldots\}$;
  Append h to hlist;
  return tiling made from hlist;
```

Figure 4.1: Pseudocode for dividing space into parallelepipeds. N is the dimensionality of the loop being tiled. Assumes $N > 1$. The purpose of the permutation `p` is to avoid forcing taxicab order on normals in `hlist`.

Underlying `pick_planes()` is a function (not shown) to determine whether a normal vector should be allowed, given the ordering constraints. Our implementation rejects a normal, ν, if

$$\exists\sigma \text{ such that the non-degenerate partial order dictated by } c(p) = \left\lfloor \frac{p \cdot \nu}{\sigma \cdot \nu} \right\rfloor$$
$$\text{is inconsistent with the ordering constraints.}$$

Again, this never applies to an unordered `foreach`.

After `pick_planes()` determines an ordered set of parallelepipeds, an ordering of points in each parallelepiped still must be chosen. In practice, it seems to matter little for performance, but legality would be a concern if `foreach` were ordered. We use lexicographic order (default) or reverse lexicographic order (if requested in the parameter file).

If the parameters to select a tiling are left unspecified in the parameter file then we use $\zeta = 0$ and:

- in the 1-dimensional case tiles are thrice the size of trivial tiles:
  ..., [0] [1] [2], [3] [4] [5], etc., in that order;

- in the 2-dimensional case tiles are thrice the size of trivial tiles:
  ..., [0, 0] [1, 0] [2, 0], [0, 1] [1, 1] [2, 1], etc., in that order;

- in the 3-dimensional case tiles are nine times the size of trivial tiles:
  ...,
  [0, 0, 0] [0, 1, 0] [0, 2, 0] [1, 0, 0] [1, 1, 0] [1, 2, 0] [2, 0, 0] [2, 1, 0] [2, 2, 0],
  [0, 0, 1] [0, 1, 1] [0, 2, 1] [1, 0, 1] [1, 1, 1] [1, 2, 1] [2, 0, 1] [2, 1, 1] [2, 2, 1],
  etc., in that order;

- and so on.

```
foreach (p in D)
  B[p] = (4 * A[p] + A[p + [1, 0]] + A[p - [1, 0]] +
          A[p + [0, 1]] + A[p - [0, 1]]) / 8;
foreach (q in D)
  A[q] = B[q];
```

Figure 4.2: Code for running example: a 2D stencil

We chose these defaults when little data were available, hoping to balance performance gains against increased compilation time. As we gather more evidence we will not hesitate to change the defaults.

4.4 Inducing a Tiling

Given a tiling for one loop and dependence information, we may *induce* a tiling for a second loop immediately following the first. The same process may then be applied again to induce tilings for subsequent loops. As an example, consider the 2-dimensional stencil code shown in figure 4.2. Figures 4.3 and 4.4 show a possible tiling for the first loop alone and for both loops together. In the case where the loops are not adjacent in the program text, we make some effort to move the intervening code (see chapter 5).

Using the notation introduced earlier, assume we have a tiling for loops L_0 through L_{n-1} with tile size K, tile space $T = \mathbb{Z}^N \times \{0, \ldots, K-1\}$, and bijections C_0 through C_{n-1} with C_i from $\{\langle x, k\rangle \mid l(k) = i\}$ to the iteration space of loop L_i. Our goal is to induce a tiling for loops L_0 through L_n.

In figure 4.3 (top), we have $n = 1$, $K = 3$, and $C_0(\langle x, k\rangle) = [3x_1 + k, x_2]$. A line in tc's output showing "derivs" for loop i indicates how $C_i(\langle x, k\rangle)$ changes when x moves one unit in some direction. The order of the listing is $\text{deriv}_1, \ldots, \text{deriv}_N$, corresponding to directions $[1, 0, \ldots, 0], [0, 1, 0, \ldots, 0], \ldots, [0, \ldots, 0, 1]$. The K lines

```
begin tiling of 1 loop
  3 nodes from loop 0 (Sample.ti:5)
   derivs: [3, 0] [0, 1]

  From Sample.ti:5 do [0, 0]
  From Sample.ti:5 do [1, 0]
  From Sample.ti:5 do [2, 0]
end tiling

begin tiling of 2 loops
  3 nodes from loop 0 (Sample.ti:5)
   derivs: [3, 0] [0, 1]
  3 nodes from loop 1 (Sample.ti:8)
   derivs: [3, 0] [0, 1]

  From Sample.ti:5 do [0, 0]
  From Sample.ti:8 do [-1, 0]
  From Sample.ti:8 do [0, -1]
  From Sample.ti:5 do [1, 0]
  From Sample.ti:8 do [1, -1]
  From Sample.ti:5 do [2, 0]
end tiling
```

Figure 4.3: Tilings for running example: top, first loop only; bottom, both loops. This is the format that `tc` uses for printing such information. The loops are identified by number (0 or 1) and also by their position in the source code (e.g., line 5 of the file "Sample.ti"). The next figure diagrams the bottom tiling.

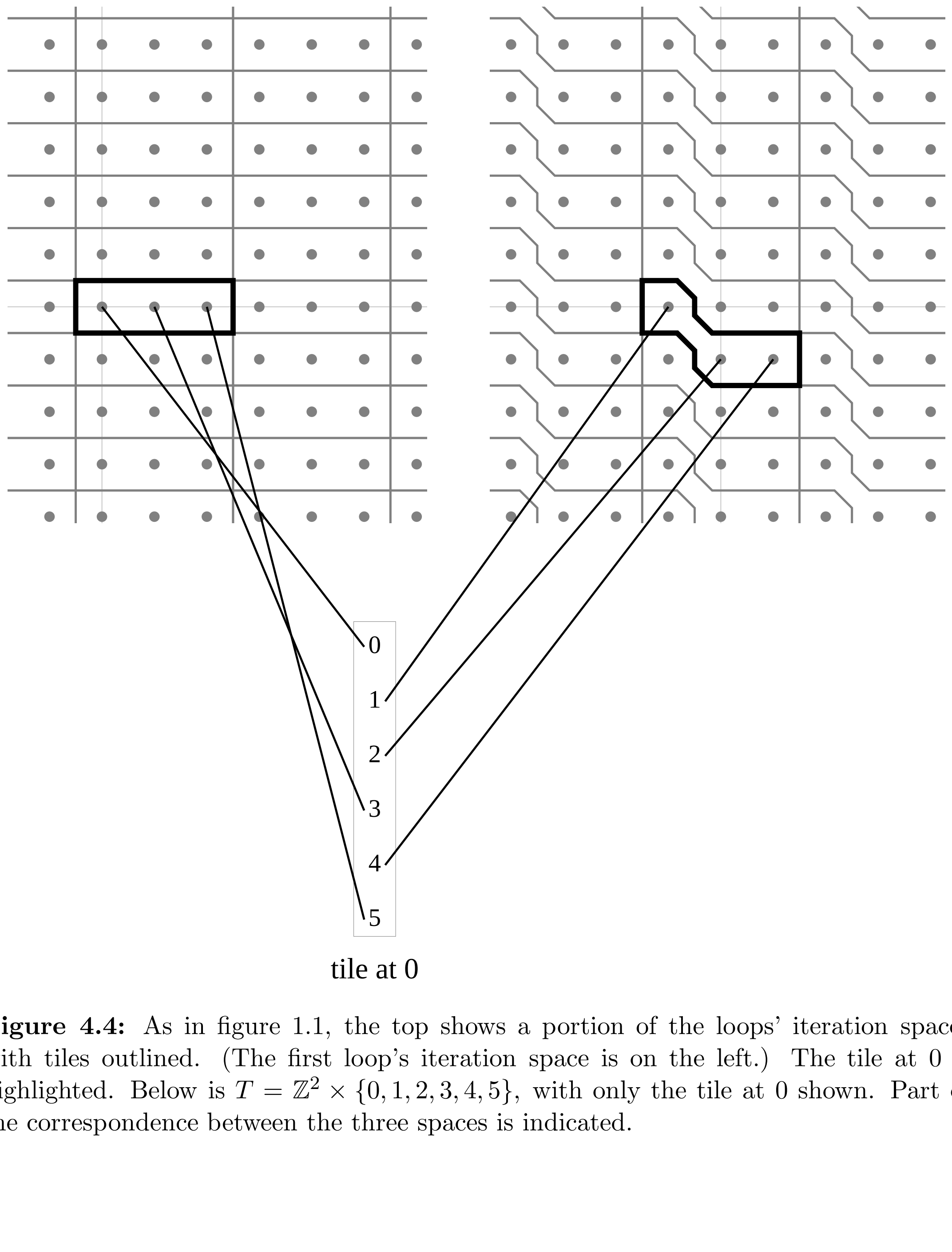

Figure 4.4: As in figure 1.1, the top shows a portion of the loops' iteration spaces with tiles outlined. (The first loop's iteration space is on the left.) The tile at 0 is highlighted. Below is $T = \mathbb{Z}^2 \times \{0, 1, 2, 3, 4, 5\}$, with only the tile at 0 shown. Part of the correspondence between the three spaces is indicated.

```
foreach (p in D)
    B[p] = (6 * A[p] + A[p + [1, 0, 0]] + A[p - [1, 0, 0]] +
            A[p + [0, 1, 0]] + A[p - [0, 1, 0]] +
            A[p + [0, 0, 1]] + A[p - [0, 0, 1]]) / 12;
foreach (q in D)
    A[q] = B[q];
```

```
begin tiling of 2 loops
  9 nodes from loop 0 (Sample3d.ti:5)
   derivs: [3, 0, 0] [0, 3, 0] [0, 0, 1]
  9 nodes from loop 1 (Sample3d.ti:9)
   derivs: [3, 0, 0] [0, 3, 0] [0, 0, 1]

  From Sample3d.ti:5 do [0, 0, 0]
  From Sample3d.ti:9 do [-1, 0, 0]
  From Sample3d.ti:9 do [0, -1, 0]
  From Sample3d.ti:9 do [0, 0, -1]
  From Sample3d.ti:5 do [0, 1, 0]
  From Sample3d.ti:9 do [-1, 1, 0]
  From Sample3d.ti:9 do [0, 1, -1]
  From Sample3d.ti:5 do [0, 2, 0]
  From Sample3d.ti:9 do [-1, 2, 0]
  From Sample3d.ti:5 do [1, 0, 0]
  From Sample3d.ti:9 do [1, -1, 0]
  From Sample3d.ti:9 do [1, 0, -1]
  From Sample3d.ti:5 do [1, 1, 0]
  From Sample3d.ti:9 do [1, 1, -1]
  From Sample3d.ti:5 do [1, 2, 0]
  From Sample3d.ti:5 do [2, 0, 0]
  From Sample3d.ti:5 do [2, 1, 0]
  From Sample3d.ti:5 do [2, 2, 0]
end tiling
```

Figure 4.5: Additional example: top, code; bottom, default tiling.

```
Tiling merge(Tiling t, Loop new_loop, list<OrderingConstraint> oc):
  t' = induce_tile0(t, new_loop, oc, false);
  if (t' is empty) return t;
  t' = calculate_derivs(t', t, new_loop, oc);
  if (t' is empty or !verify(t', new_loop, oc)) return t;
  return t';
```

Figure 4.6: Pseudocode for `merge()`. Returns a new tiling that includes everything in `t` and `new_loop`, or `t` if unsuccessful.

```
Tiling induce_tile0(Tiling t, Loop new_loop,
                    list<OrderingConstraint> oc, bool only_one):
  Let I be the iteration space of new_loop;
  b = I \ {r | r forbidden by oc if p ≺ ⟨0,0⟩ have executed};
  list<steps> new_steps = empty;
  int K = number of steps in t;
  for k from 0 to K - 1
    Append step k of t to new_steps;
    s = I \ {r | r forbidden by oc if p ⪯ ⟨0,k⟩ have executed} \ b;
    if (s is infinite) return empty;
    for every element e of s
      Append e to new_steps;
      if (only_one) goto done;
      b = b ∪ {e};
 done:
  Tiling t' = t with new_steps;
  return t';
```

Figure 4.7: Pseudocode for `induce_tile0()`. Returns a new tiling if successful. The new tiling is incomplete insofar as the mapping from $\langle x, k\rangle$ in tile space to I is valid only if $x = 0$. Returns empty if unsuccessful.

of the form "From ... do ..." indicate the steps of the tile, in order. Lexicographic order on T determines execution order; in the example that yields the order mentioned in the previous section as the default for a 2-dimensional tiling. An additional example is shown in figure 4.5.

The idea of the `merge()` function (figure 4.6) is to induce a tile by stepping through our tiling at a particular position in space, calculating whether and how the induced tile should move as we move in tile space, and verifying that the whole process has resulted in a legal reordering. If so, the result will be a tiling that is similar to the one from which we began, but with one or more steps belonging to L_n intermingled.

In figures 4.7 and 4.8 we illustrate how we attempt to extend a tiling from n loops to $n + 1$ loops. The idea is the consider the tiles at 0 and at $[1, 0, \ldots, 0]$, $[0, 1, 0, \ldots, 0], \ldots,$ $[0, \ldots, 0, 1]$. Our goal when stepping through the tile at 0 is

```
Tiling calculate_derivs(Tiling t', Tiling t,
                        Loop new_loop, list<OrderingConstraint> oc):
  list<vector in $\mathbb{Z}^N$> result = empty list;
  int first = min {k | step k of t' belongs to new_loop};
  for each v in $\{[1,0,\ldots,0],[0,1,0,\ldots,0],\ldots,[0,\ldots,0,1]\}$
    Tiling shift_t = t translated by v;
    Tiling shift_t' = induce_tile0(shift_t, new_loop, oc, true);
    if (shift_t' is empty) return empty;
    int first' = min {k | step k of shift_t' belongs to new_loop};
    if (first $\neq$ first') return empty;
    Let $I$ be the iteration space of new_loop;
    Let $f$ be the function in t' that maps
     from $\langle 0,$first$\rangle$ to $I$;
    Let $f'$ be the function in shift_t' that maps
     from $\langle 0,$first$\rangle$ to $I$;
    Append $f'(\langle 0,$first$\rangle) - f(\langle 0,$first$\rangle)$ to result;

  return t' with derivs for new_loop specified by result;
```

Figure 4.8: Pseudocode for `calculate_derivs()`. Makes `t'` complete or returns empty if it fails.

to determine what points in the iteration space of the new loop become available for execution after each step. In the running example, iteration $[0, 0]$ of loop 0 is the last time that loop touches `A[`$-1, 0$`]`, so iteration $[-1, 0]$ of loop 1 may then execute. Iteration $[0, 0]$ of loop 0 is also the last time that loop touches `A[`$0, -1$`]`, so iteration $[0, -1]$ of loop 1 becomes available at the same time. (In the example, since both $[-1, 0]$ and $[0, -1]$ became available at the same time, the tie is broken by lexicographic ordering or reverse lexicographic ordering, depending on a parameter.) Similar reasoning shows that after iteration $[1, 0]$ of loop 0, iteration $[1, -1]$ of loop 1 becomes ready. However, it turns out that iteration $[2, 0]$ of loop 0 reads elements `A[`$2, 0$`]`, `A[`$2 \pm 1, 0$`]`, and `A[`$2, \pm 1$`]`, all of which will later be read again by loop 0. Therefore, no new iterations of loop 1 become ready after iteration $[2, 0]$ of loop 0.

Continuing with our example, `calculate_derivs()` (figure 4.8) finds $\text{deriv}_1 = [3, 0]$ and $\text{deriv}_2 = [0, 1]$ for loop 1. Let $s_1 = [-1, 0]$, $s_2 = [0, -1]$, and $s_4 = [1, -1]$. The subscripts were chosen to correspond to step numbers in tile space. Let C_1 from $\mathbb{Z}^2 \times \{1, 2, 4\}$ to $\mathbb{Z}^2$ be:

$$C_1(\langle [x_1, x_2], k \rangle) = s_k + x_1 \cdot [3, 0] + x_2 \cdot [0, 1] \quad .$$

The function C_1 is what the new tiling will use to map from tile space to the iteration space of loop 1. In general, if the k^{th} step selected for the tile at 0 by `induce_tile0()` is s_k in the iteration space of the new loop, we define

$$C_n(\langle [x_1, \ldots, x_N], k \rangle) = s_k + \sum_{i=1}^{N} x_i \text{deriv}_i \quad .$$

```
bool verify(Tiling t', Loop new_loop, list<OrderingConstraint> oc):
  if (size of tile at 0 is incorrect) return false;
  generic vector $a$ = $[a_1, \ldots, a_N]$;
  /* $n$ is number of loops before we added new_loop to t'. */
  int $n$ = (number of loops in t') - 1;
  for $i$ from 0 to $n$
    b[$i$] = $\{C_i(\langle x, k\rangle) \mid l(k) = i \wedge \langle x, k\rangle \prec \langle a, 0\rangle\}$;
  /* b[0..$n$] now represent the set of nodes that will
     execute before the tile at $a$.  b[$i$] is a subset
     of $L_i$'s iteration space. */
  Let $I$ be the iteration space of new_loop;
  ready = $I \setminus \{r \mid r$ forbidden by oc if b[0..$n$] have executed$\}$;
  if ($\exists a$ such that b[$n$] $\not\subseteq$ ready) return false;
  int $K$ = number of steps in t';
  for k from 0 to $K$ - 1
    $i$ = $l$(k);
    generic vector v = $C_i(\langle a, k\rangle)$;
    /* Only verify nodes from new_loop. */
    if ($i$ == $n \wedge \exists a$ such that (v $\in$ b[$i$] $\vee$ v $\notin$ ready)) return false;
    b[$i$] = b[$i$] $\cup$ $\{$v$\}$;
    ready = $I \setminus \{r \mid r$ forbidden by oc if b[0..n] have executed$\}$;

  return true;
```

Figure 4.9: Pseudocode for `verify()`. By `generic vector` we mean a vector in $\mathbb{Z}^N(a_1, \ldots, a_N)$, i.e., $\mathbb{Z}^N$ with N symbolic variables. Returns true if `t'` is a legal reordering.

4.4.1 Correctness

Have we induced a correct reordering of the program? The purpose of `verify()` is to determine whether C_n is actually a bijection and whether the new tiling respects all ordering constraints. The former is checked in two parts: we check that the tile size is correct and that $p \neq q$ implies $C_n(p) \neq C_n(q)$. The volume of the induced tile in the new loop's iteration space is the number of new steps found in the first call to `induce_tile0()`. In our example that is three. According to Observation 4.2.1, that is the correct tile size if and only if it is equal to the volume of a parallelepiped whose edges are congruent to the derivs. In our example the derivs are $[3, 0]$ and $[0, 1]$. The induced tile has the correct size.

Determining whether $p \neq q$ implies $C_n(p) \neq C_n(q)$ and determining whether ordering constraints are violated occur in a pass through a generic tile, the tile at $[a_1, \ldots, a_N]$ in tile space. We rely on the Omega library's ability to represent and manipulate symbolic variables. See figure 4.9.

4.4.2 Variations

Many possible variations spring to mind, some of which we have implemented.

Restricting the Interleaving of Steps

Normally we allow arbitrary interleaving of the steps belonging to different loops. However, a parameter setting can force the creation of a tiling where all steps belonging to L_0 come first, followed by all steps belonging to L_1, and so on. Such a tiling is useful for two reasons. First, it roughly mimics what one might get from a compiler that does tiling followed by loop fusion rather than doing the two together, allowing us to make some comparisons. Second, sometimes this approach improves temporal locality. Sometimes it is a small optimization and other times it is a small pessimization.

Reshaping Induced Tiles

After `induce_tile0()` and `calculate_derivs()` succeed, we have an optional procedure for reshaping the induced tile. See figure 4.11. The primary purpose of this is to allow comparison of our technique with alternatives that, for example, only generate tiles of certain shapes. However, as with the restriction on interleaving, sometimes this approach improves temporal locality. In other words, it could be legitimate optimization.

If a parameter turns it on (not the default), the reshaping pass constructs S, a set of points in a parallelepiped with edges congruent to the derivs calculated by `calculate_derivs()`. We are attempting to induce a tile with the shape of S and the same derivs. We select $|S|+1$ translations of S to investigate further (figure 4.10), but filter out those that do not lie completely within the ready set. (The number to investigate is arbitrary. In practice, $|S|+1$ works well.) We eventually repeat a process similar to the original induction with a goal parallelepiped enforced (figure 4.11). If `reinduce_tile0()` succeeds then we do not call `calculate_steps()` again; instead, we proceed to `verify()`.

In the running example, the induced tile at 0 contains points $[-1,0]$, $[0,-1]$, and $[1,-1]$. Reshaping, if requested, would lead to

$$\texttt{choose\_translations}(\{[-1,0],[0,-1],[1,-1]\},\{[-1,0],[0,0],[1,0]\},3,$$
$$\{[x,y] \mid x<-1 \vee (x=-1 \wedge y<1) \vee (x<2 \wedge y<0)\})$$

being called, whose result would be a list containing $[0,-1]$ and $[-2,0]$. $[0,0]$ and $[-1,0]$ are not in the result because $\{[-1,0],[0,0],[1,0]\}$ and $\{[-2,0],[-1,0],[0,0]\}$ are not subsets of the ready set. $\{[-1,-1],[0,-1],[1,-1]\}$ and $\{[-3,0],[-2,0],$

```
list<vector> choose_translations(set<vector> a, set<vector> $S_0$,
                                 int count, set<vector> ready):
  queue<vector> Q = non-zero vectors in $\mathbb{Z}^N$ sorted by taxicab order;
  list<vector> result = empty list;
  for each element o of Q {
    set<vector> c = $\{r + o \mid r \in S_0\}$;
    if ($a \cap c \neq \emptyset$) {
      Append o to result;
      if (length of result is count) goto filter;
    }
  }
 filter:
  insert the zero vector at front of result;
  for each element o of result {
    set<vector> c = $\{r + o \mid r \in S_0\}$;
    if ($c \not\subseteq$ ready) Remove o from result;
  }
  return result;
```

Figure 4.10: Pseudocode for selecting plausible placements of a reshaped tile. The first two arguments will be the points in the tile at 0 (as induced) and a set of points in the desired shape, translated to overlap. The `count` argument will be $|a|$, though it could reasonably be set to a different value. Assumes `a` and S_0 overlap. In an actual implementation, one would prefer to merge the two loops for efficiency.

```
bool reinduce_tile0(Tiling t', Tiling t, Loop new_loop,
                    list<OrderingConstraint> oc, set<vector> $S$):
  Let $I$ be the iteration space of new_loop;
  Let $C$ be the function in t' from tile space to iteration spaces;
  int $o$ = the first step of t' that belongs to new_loop;
  vector $m$ = the lexicographically least element of $S$;
  /* Make least element of $S_0$ equal to first point induced in $I$. */
  set<vector> $S_0$ = $\{p + C(\langle 0, o \rangle) - m \mid p \in S\}$;
  set<vector> induced_points = $\{C(\langle 0, k \rangle) \mid$ step $k$ belongs to new_loop$\}$;
  int $K$ = the number of steps in t;
  /* Compute the ready set after the tile at 0. */
  set<vector> ready =
     $I \setminus \{r \mid r$ forbidden by oc if $p \preceq \langle 0, K-1 \rangle$ in t have executed$\}$;
  list<vector> translations =
     choose_translations(induced_points, $S_0$, $|S|$, ready);
  if (translations is empty) return false;
  int w = get parameter$_0$ "Which translation?";
  set<vector> goal = $\{p +$ translations[w] $\mid p \in S_0\}$;
  b = $I \setminus$ goal $\setminus \{r \mid r$ forbidden by oc if $p \prec \langle 0, 0 \rangle$ in t have executed$\}$;
  list<steps> new_steps = empty;
  for k from 0 to $K$ - 1
    Append step k of t to new_steps;
    s = $I \setminus \{r \mid r$ forbidden by oc if $p \preceq \langle 0, k \rangle$ in t have executed$\} \setminus$ b;
    for every element e of s $\cap$ goal;
      Append e to new_steps;
      b = b $\cup$ {e};

  if (goal $\not\subseteq$ b) return false;

  Replace steps in t' with new_steps;
  return true;
```

Figure 4.11: Pseudocode for reshaping tiles. Returns whether it succeeds.

```
begin tiling of 2 loops
  3 nodes from loop 0 (Sample.ti:5)
   derivs: [3, 0] [0, 1]
  3 nodes from loop 1 (Sample.ti:8)
   derivs: [3, 0] [0, 1]

  From Sample.ti:5 do [0, 0]
  From Sample.ti:8 do [-3, 0]
  From Sample.ti:8 do [-2, 0]
  From Sample.ti:8 do [-1, 0]
  From Sample.ti:5 do [1, 0]
  From Sample.ti:5 do [2, 0]
end tiling
```

Figure 4.12: Tiling with reshaped induced tile. Some previous systems can only generate tiles of certain shapes, and reshaping our tiles in a post-processing pass allows us to mimic them for comparison purposes. Reshaping also may be a legitimate optimization or pessimization, determined largely by the patterns of data or cache line reuse.

$[-1, 0]\}$ are subsets of the ready set. Figure 4.12 shows the tiling after reshaping if the goal parallelepiped is set to $\{[-3, 0], [-2, 0], [-1, 0]\}$.

Unimplemented Variations

We have designed and implemented an algorithm that induces a tiling for loops $L_0, \ldots, L_n$ from a tiling for $L_0, \ldots, L_{n-1}$. One disadvantage of this design is that it steps through many different tilings as it induces tilings over more and more loops. It seems possible to construct a one pass algorithm that directly constructs the tiling for $L_0, \ldots, L_n$ from the tiling for L_0 and ordering constraints. Avoiding the construction of the intermediate tilings would speed compilation significantly, especially when n is large.

Another possible twist that we have not tried is doing the induction backwards (or in both directions). That is, instead of tiling L_0 first, one could tile L_n first. An algorithm to induce a tiling for $L_m, \ldots, L_n$ from a tiling for $L_{m+1}, \ldots, L_n$ should be fairly easy to devise. That algorithm would generate slightly different tilings than the algorithm we implemented, but we see no obvious benefit to one over the other.

Chapter 5

Implementation Details

5.1 Titanium/Stoptifu Interface

Stoptifu performs storage optimizations, tiling, and loop fusion in a library separate from `tc`. The Stoptifu library can, in theory, be attached to any compiler. Data is passed in to the library using Stoptifu's AST representation, which is distinct from `tc`'s. After Stoptifu's analysis and transformation, data is transferred back to the caller via a tree translation pass. Typically, the caller customizes the tree translation to generate code that represents the tiled, optimized program in the caller's intermediate form.

The Stoptifu library treats the bodies of loops as opaque statements. A list of ordering constraints must be provided by the caller. If storage optimizations are requested (§6) then the caller also must provide information about what data are read and written in each loop iteration.

```
(bridges)
for all i, setup for L_i;
if (can use tiling /* the precheck */)
 execute tiles ordered lexicographically in tile space;
else {
 foreach- (p_0 in D_0) S_0;
 ⋮
 foreach- (p_{n-1} in D_{n-1}) S_{n-1};
}
(bridges)
```

Figure 5.1: Outline of C code output. The setup code computes pointers and increments for strength-reduced address calculations and computes the number of points in each D_i. By `foreach-` we simply mean a `foreach` without the setup code. Each bridge now appears at the top or bottom.

5.2 Selecting Loops to Tile Together

The loops tiled together by the Stoptifu library must be full domain loops (§3.4.2) that have the same dimensionality. They must be presented to the library as a sequence of loops with no intervening code. Rarely does a program appear in that format, so `tc` makes a modest effort to reorder code for presentation to Stoptifu.

The first step `tc` takes is to find pairs of loops that are full domain loops with the same dimensionality such that all control-flow paths from the first loop inevitably lead to the second loop. We call the code between the two loops the *bridge*. Bridges can contain arbitrary code. We discard pairs for which the bridge can be entered other than from the end of the first loop.

Pairs are then strung together into longer sequences, if possible. For example, the pairs $\langle L_0\textit{(bridge)}L_1\rangle$ and $\langle L_1\textit{(bridge)}L_2\rangle$ together suggest that L_0, L_1, and L_2 might be fused. A greedy approach is `tc`'s default, but any legal sequence may be selected by using the parameter file.

Let the loops

```
foreach (p_0 in D_0) S_0;
⋮
foreach (p_{n-1} in D_{n-1}) S_{n-1};
```

be called L_0 through L_{n-1}. If they are to be tiled together then we will generate code of the form shown in figure 5.1. If any bridge cannot legally be moved forwards or backwards to accomodate that pattern then we do not attempt to tile loops $L_0, \ldots, L_{n-1}$ together.

Though not always necessary, we find it helpful to include a runtime test, the *precheck*, that determines whether a tiling should be used. The precheck includes, at the least, a test that the iteration spaces of the loops are unit-stride rectangles or a union of same. That particular test is not for correctness but for efficiency. We also check that each loop's iteration space contains at least some minimum number of nodes (default 500). That minimum may be set separately for each tiling. If the precheck fails at runtime then we fall back on a version of the code that was not optimized by Stoptifu.

5.3 Disjoint Pairs of Arrays

The ordering constraints we pass to the Stoptifu library are mostly constructed from array dependence information. (Other constraints come from scalar dependences or from method calls with unknown side-effects.) Some dependences can be profitably ignored in the sense that they are possible according to static analysis but at runtime they are present seldom or never. In order to optimize more programs we can assume that certain arrays are disjoint, i.e., their elements are not aliased. (Each such assumption can be forbidden by a parameter.) For safety, if we do optimize under one or more such assumptions then a runtime test is added to the precheck.

An assumption that two arrays `a` and `b` do not overlap is only considered if there are no assignments `a = b` or `b = a` and an ordering constraint can be removed by making the assumption. Static analysis proving `a` and `b` are disjoint is unnecessary if one is willing to bear the cost of the runtime test and the speculative compilation that may or may not bear fruit.

5.4 Generating Code for Tilings

C code to execute tiles in lexicographic order is created in three major parts: loop setup, an optimized inner loop for what we hope is the common case, and a catch-all, alternative inner loop for any nodes in tile space that cannot be handled by the optimized inner loop. A more refined approach would be to create multiple special-case inner loops (as is done in PHiPAC, for example). That may have the highest benefit-to-cost ratio of any item on our wish list.

Figure 5.2 outlines our generated C code for executing tiles in order for the 3-dimensional case (other cases are similar). As in §1, we use k to represent a step number within a tile and $l(k)$ to mean the number of the loop whose body is executed in the k^{th} step of a tile. The idea is to determine at the start what tiles are complete, i.e., what tiles must execute all of their steps. Those tiles are executed by the optimized inner loop.

At runtime it is important to represent the sets of points in tile space efficiently. When possible we use a *simple rectangle*, stored as a minimum and maximum for each dimension. A simple rectangle is a subset of $\mathbb{Z}^N$ that can be expressed as

$$\{(u_1, \ldots, u_N) \mid (l_1 \le u_1 \le h_1) \wedge \cdots \wedge (l_N \le u_N \le h_N)\} \quad .$$

```
for $k$ from 0 to $K$ - 1
  $A_k$ = $\{x \mid C(\langle x, k\rangle) \in \mathtt{D}_{l(k)}\}$;
/* $x \in$ All iff we must visit the tile at $x$. */
All = $\bigcup_i A_i$;
/* $x \in$ Best iff we must do all steps of the tile at $x$. */
Best = $\bigcap_i A_i$;
All_1_2 = $\{(x_1, x_2) \mid \exists x_3$ such that $(x_1, x_2, x_3) \in$ All$\}$;
for every $(t_1, t_2) \in$ All_1_2 {
  /* Do a stack of tiles, possibly with holes. */
  set todo = $\{t_3 \mid (t_1, t_2, t_3) \in$ All$\}$;
  set besttodo = $\{t_3 \mid (t_1, t_2, t_3) \in$ Best$\}$;
  int start, end = max(todo);
  while (true) {
    start = min(todo);
    if (start $\in$ besttodo) {
      int lastbest = max $\{t_3 \mid \{$start$, \ldots, t_3\} \subseteq$ besttodo$\}$;
      for $t_3$ from start to lastbest
        do best-case tile at $(t_1, t_2, t_3)$;
      if (lastbest == end) break;
      todo = todo $\setminus \{-\infty, \ldots,$ lastbest$\}$;
    } else {
      int lastnotbest = max $\{t_3 \mid \{$start$, \ldots, t_3\} \subseteq ($todo $\setminus$ besttodo$)\}$;
      for $t_3$ from start to lastnotbest
        do general-case tile at $(t_1, t_2, t_3)$;
      if (lastnotbest == end) break;
      todo = todo $\setminus \{-\infty, \ldots,$ lastnotbest$\}$;
    }
  }
}
```

Figure 5.2: C pseudocode for executing tiles in order for a tile space $\mathbb{Z}^3 \times \{0, \ldots, K-1\}$.

```
if ($t \in A_0$) { $\mathtt{p}_{l(0)} = C(\langle t, 0\rangle)$; $\mathtt{S}_{l(0)}$; }
if ($t \in A_1$) { $\mathtt{p}_{l(1)} = C(\langle t, 1\rangle)$; $\mathtt{S}_{l(1)}$; }
⋮
if ($t \in A_{K-1}$) { $\mathtt{p}_{l(K-1)} = C(\langle t, K-1\rangle)$; $\mathtt{S}_{l(K-1)}$; }
```

Figure 5.3: C pseudocode for executing the general-case tile at $t = (t_1, t_2, t_3)$. Both the general-case and best-case tiles are fully unrolled.

For example, if $\texttt{D}_{l(k)}$ is a simple rectangle and the derivs for $L_{l(k)}$ are axis-aligned then A_k must be a simple rectangle. `Best` must be a simple rectangle if all the A_k's are. `All` may be approximated with a simple rectangle (default) or represented exactly, depending on a parameter setting. In stencil codes, `All` often turns out to be the union of nearly-identical simple rectangles.

The type generated for `todo` (or `besttodo`) is a range of integers or an ordered list of disjoint ranges, depending on the type of `All` (or `Best`).

Figure 5.3 outlines how a general-case tile is executed. A best-case tile is similar except that the conditionals are not necessary. An assignment to $\texttt{p}_{l(k)}$ should be thought of as an update of the pointers created by strength reduction (§3.4.2) in addition to (or instead of) the update of $\texttt{p}_{l(k)}$. At an assignment to a particular $\texttt{p}_{l(k)}$ in the best-case tile we can usually optimize those pointer updates because the difference between the old value of $\texttt{p}_{l(k)}$ and the new value is known at compile time.

Chapter 6

Storage Optimizations

6.1 Introduction

Stoptifu currently performs three storage optimizations: array contraction, delaying writes to oft-written array elements, and reusing values instead of reloading them. It attempts to apply the three optimizations in that order. Many other storage optimizations are possible. We also benefit from the C compiler's optimizations, from hardware prefetching, etc.

Of particular note is our algorithm for array contraction, which is more flexible and aggressive than the versions found in other compilers. We can contract an array to any combination of lower-dimensional arrays and scalars, as appropriate. We also can defer one or more of the safety tests to runtime, thereby allowing broader application of the transformation.

To enable storage optimizations, `tc` provides the Stoptifu library with a list of all possible array reads and writes per loop iteration, all *optimizable array reads*, and all *optimizable array writes*. An optimizable array read is of the form `v = a[`$R(p)$`]`, where p is the iteration point, R is a relation from the iteration

```
B.junk();
foreach (p in D)
  B[p] = (4 * A[p] + A[p + [1, 0]] + A[p - [1, 0]] +
          A[p + [0, 1]] + A[p - [0, 1]]) / 8;
foreach (q in D)
  A[q] = B[q];
B.junk();
```

Figure 6.1: Example amenable to array contraction.

space to the array index set, $|R(p)| = 1$ for all values of p, and every iteration of the loop reads `a[`$R(p)$`]`, but no iteration of the loop writes `a[`$R(p)$`]`. Similarly, an optimizable array write is of the form `a[`$R(p)$`] = w`, where p is the iteration point, R is a relation from the iteration space to the array index set, $|R(p)| = 1$ for all values of p, the write is performed on every iteration of the loop, and the value written is the same as the value of `a[`$R(p)$`]` just after the end of iteration p.

The library returns a list of changes to the program that do not change its semantics but may improve its performance. The changes may include the allocation of temporary storage (scalars and arrays), writes and reads to and from said locations, and elimination of operations thereby made redundant. Temporary scalars are generally intended to reside in machine registers, but our implementation does not have that level of control. Allocation of temporary arrays is only necessary if array contraction is successful.

6.2 Contracting Arrays

6.2.1 Motivation

Arrays used as scratch space can sometimes be eliminated or dramatically reduced in size. The tiled version of the program in figure 6.1 is about 1.4 times faster

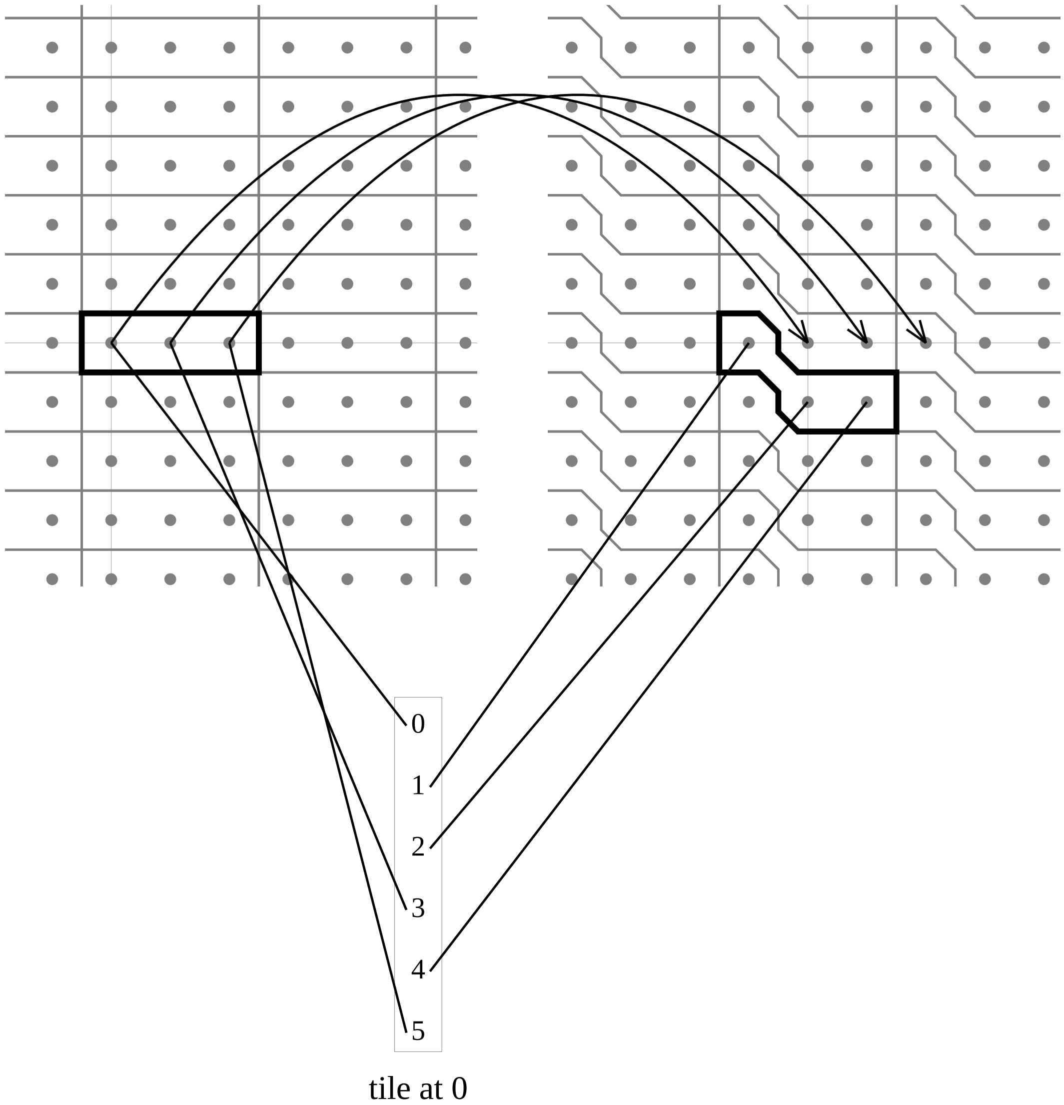

Figure 6.2: This figure is similar to figure 4.4, but it indicates the flow of temporary data via the `B` array in the source code. The top shows a portion of the loops' iteration spaces with tiles outlined. (The first loop's iteration space is on the left.) The tile at 0 is highlighted. Below is $T = \mathbb{Z}^2 \times \{0, 1, 2, 3, 4, 5\}$, with only the tile at 0 shown. Part of the correspondence between the three spaces is indicated. The curved arrows from the left side to the right side indicate the flow of data that can be optimized via array contraction. In each tile, the first two values written to `B` (in steps 0 and 3) are consumed in the very next tile (in steps 2 and 4). They can be stored in registers. The third write to `B` is consumed in the next stack of tiles, potentially a long time later. But it can be replaced with a write to a 1-dimensional compiler-generated temporary array.

with array contraction. For programs amenable to it, the running time saved from tiling with array contraction is frequently more than double the time saved from tiling alone.

In figure 6.1, the array `B` is scratch space. The programmer has helpfully invoked the `junk` method, which non-deterministically writes to all array elements. In other words, it does nothing at runtime. It signals the compiler that the contents of `B` need not be consistent across calls to `B.junk()`. Without the trailing call to `junk`, the compiler would have to worry about later reads of `B`. Although interprocedural liveness analysis is possible, `tc` does not yet implement it. Even when we do add liveness analysis, the `junk` method will remain a useful tool, because sometimes liveness analysis cannot or will not infer what one wants. The leading call to `junk` is less important, as the set of `B`'s elements read is exactly the set written.

Let the loops in our example be fused and tiled with T being the tiled iteration space. Each tile in T writes a few elements of `B` and reads the same number, from almost (but not exactly) the same places in `B`.

Consider the iteration with some $S \subset T$ completed and $\bar{S}$ not. Most elements of `B` that have been written have also been read, in which case they are now valueless. Elements of `B` not yet written are also valueless. The only elements that matter are the ones written but not yet read, and the number of them is proportional to the size of the boundary between S and $\bar{S}$.

The goal of array contraction is to replace scratch arrays with smaller scratch areas or scalar variables. In the example, optimization roughly halves the amount of data flowing between the processor and memory by replacing `B` with compiler-generated scratch space whose size is approximately the maximum number of live elements in `B`. See figure 6.2.

Of course, in a naïve implementation of the code in figure 6.1, the maximum

```
contract_array(X):
  int firstWrite = min {i | L_i may write to X};
  int lastWrite = max {i | L_i may write to X};
  int firstRead = min {i | L_i may read from X};
  int lastRead = max {i | L_i may read from X};
  if (firstRead < firstWrite || lastWrite > lastRead ||
      X's contents may be important after {L_i | L_i may write to X})
    fail;

  /* It could be a scratch array.  Check every read. */

  /* This maps a read site in some step to a write site in
     some step separated by a fixed distance in tile space. */
  map reader_to_writer;

  /* This maps a write site in some step to the distance in
     tile space over which the value written is alive. */
  map writer_to_maxdist;

  for k_r from 0 to K-1 {
    for every read site, s_r, of an element of X in step k_r {
      if (∃ a vector v, a step k_w, and a write site s_w
          s.t. ∀x, ⟨x+v, k_w⟩ ≺ ⟨x, k_r⟩ and if ⟨x+v, k_w⟩ executed
          then the read at site s_r in ⟨x, k_r⟩ must read
          the value that was written at site s_w in ⟨x+v, k_w⟩) {
        reader_to_writer[(k_r, s_r)] = (v, k_w, s_w);
        if (writer_to_maxdist[(k_w, s_w)] is not set ||
            v ≺ writer_to_maxdist[(k_w, s_w)])
          writer_to_maxdist[(k_w, s_w)] = v;
      } else fail;
    }
  }
  /* Success! */
  return reader_to_writer and writer_to_maxdist;
```

Figure 6.3: Pseudocode for array contraction. By *read site* we mean a particular textual instance of `... = X[...]`. By *write site* we mean a particular textual instance of `X[...] = ...`. Our implementation assumes `X`'s contents might be important later unless it sees a call to `X.junk()`.

number of live elements in `B` is |`D`|. We must fuse the loops to make the array contraction sensible.

6.2.2 Algorithm and Implementation

Arrays that may be aliased or may be read or written in unknown places are not contractable. (Aliased in ways not screened out in the precheck, that is.) All remaining arrays are considered candidates for contraction. Figure 6.3 shows how we decide whether it is legal to contract an array. If it is legal to contract an array then we do, unless overridden by a per-array parameter from the parameter file. Essentially, we contract an array if we can locate each textual read, and for each such read we can locate a particular textual write that supplies its value and that is a fixed distance away in T.

Conceptually, once an array is designated for contraction, each textual write becomes a write to a separate scratch area, either a scalar or an array. That scratch area is only written once per tile. (Each textual read has a corresponding textual write at a corresponding distance in T, so it always reads from only one such scratch area.) The scalar case applies if the value to be written is consumed (becomes dead) within at most some fixed number of tiles. If the array being contracted is 1D then the scalar case is the only case; otherwise the conditions for contraction would not be satisfied. (Besides, it seems silly to "contract" a 1D array to another 1D array.)

When the scalar case does not apply, the number of simultaneously live values generated by one stack of tiles is bounded by the height of the stack. (A particular textual write occurs statically once per tile, and dynamically it occurs once or not all.) We use the term *stack of writes* to refer to the values generated by a particular textual write over the course of a tile stack's execution. In the 2D case the number

```
B.junk();
foreach (p in D)
  B[p] = (4 * A[p] + A[p + [1, 0]] + A[p - [1, 0]] +
          A[p + [0, 1]] + A[p - [0, 1]]) / 8;
foreach (q in D)
  A[q] = (B[q] + B[q + [0, 2]]) / 2;
B.junk();
```

Figure 6.4: A slightly different example amenable to array contraction—a variant of figure 6.1.

of stacks of writes that are alive at once will be fixed at compiler time; otherwise the conditions for contraction would not be satisfied.

In 3D or higher, there are two interesting cases again: a fixed number of stacks of writes alive at once (i.e, when each stack of writes is fully consumed after a fixed number of subsequent tile stacks), or an arbitrary number. It is just the same thing over again. For a 3D array, contraction of each textual write is down to scalars, some fixed number of 1D stacks of writes, or an unknown number of 1D stacks of writes, which amounts to a 2D structure. (Never a 3D structure, because that would violate the conditions for contraction.) In general, an N-dimensional array can contract to any combination of scalars and lower-dimensional arrays. When discussing an individual write we say we "contract it from N dimensions to M dimensions."

Strout et al. [34] analyze storage requirements for an array A when

$$\exists v \text{ such that } \forall p, A[p - v] \text{ must be dead by the time } A[p] \text{ is written.}$$

The identical analysis applies to array writes that we contract from N dimensions to $N - 1$ dimensions, and an analogous analysis applies to the general case. Essentially, the only complication is when v can be expressed as tv' for some positive

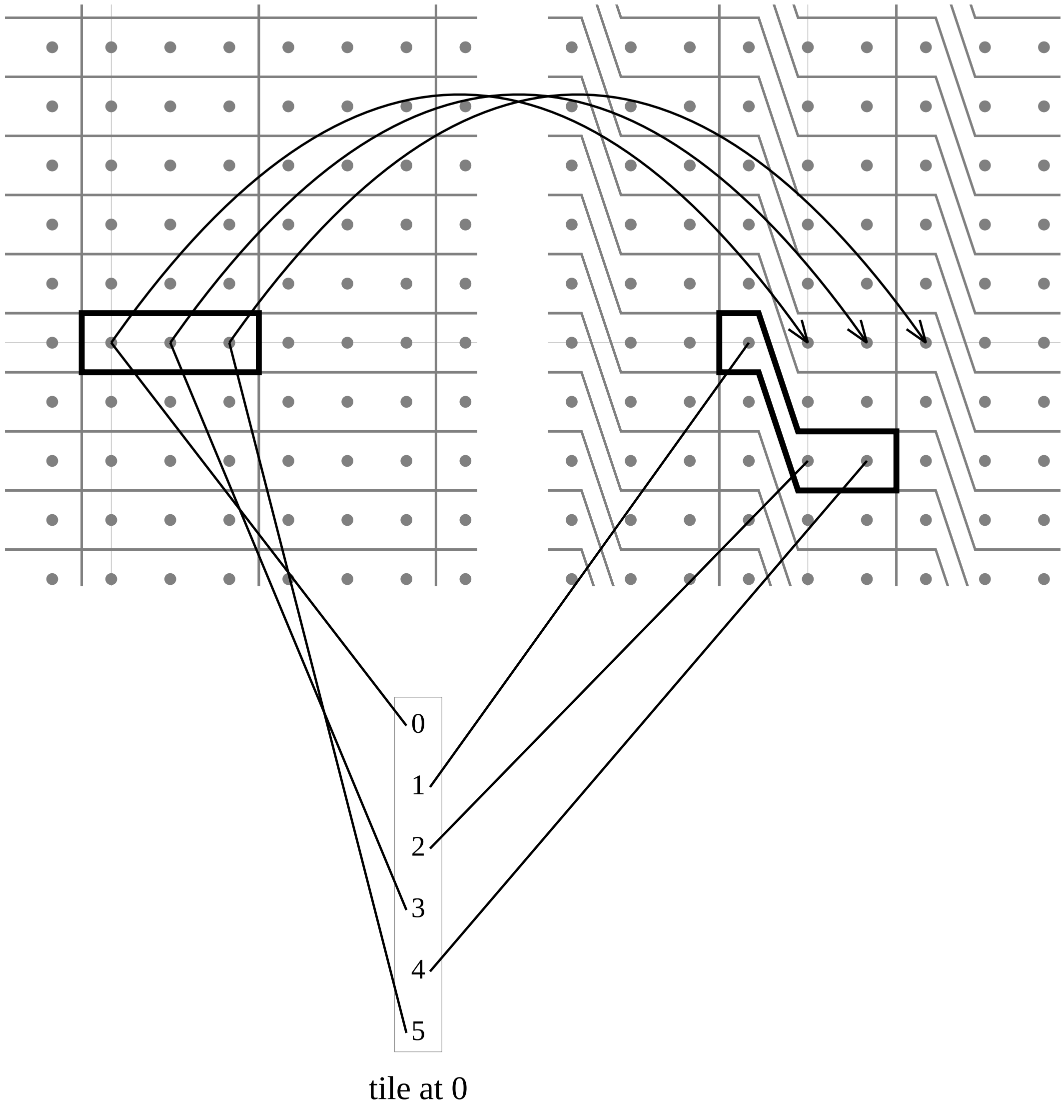

Figure 6.5: Illustration, in the same style as figure 6.2, of Stoptifu's default tiling of the code from figure 6.4.

integer, t, and $v' \in \mathbb{Z}^N$. Figure 6.4 presents a slight modification of our running example. A basic tiling of that program illustrates the complication. See figure 6.5. The array `B` can still be contracted, but the portion that is contracted to scalars requires more storage than in the previous example. Generally two values from tile step 0 and two values from tile step 3 are live at any moment. The logical way to store four values is in four registers, but there are only two program points that do the writing.

Applying Strout et al.'s analysis to our framework yields two possible solutions. First, each array write that is contracted to a scalar can use a circular buffer of t scalars if at most t values could be simultaneously live. For example, with $t = 2$,

```
B[...] = expression;
```

becomes

```
temp1 = temp2;
temp2 = expression;
```

and corresponding reads of `B` use `temp1` or `temp2` as appropriate. Alternatively, one can simply increase the tile size, as is illustrated by figure 6.6.

As a final note, we need to worry about reads of prior values of a contracted scratch array unless the array is junked, as in our example. If it is not junked, we add logic to the precheck that fails if any compiler-generated temporary value might be read before it is written. For example, if array contraction causes step 7 of the tile at $x = (x_1, x_2, x_3)$ to write `temp[`x_3`]` and step 3 of the tile at $x - v$ to read `temp[`x_3`]`, where $\langle x, 7\rangle \prec \langle x - v, 3\rangle$, then we compute

$$
\begin{aligned}
A_3 &= \left\{x \mid C(\langle x, 3\rangle) \in \mathtt{D}_{l(3)}\right\} \\
A_7 &= \left\{x \mid C(\langle x, 7\rangle) \in \mathtt{D}_{l(7)}\right\}
\end{aligned}
$$

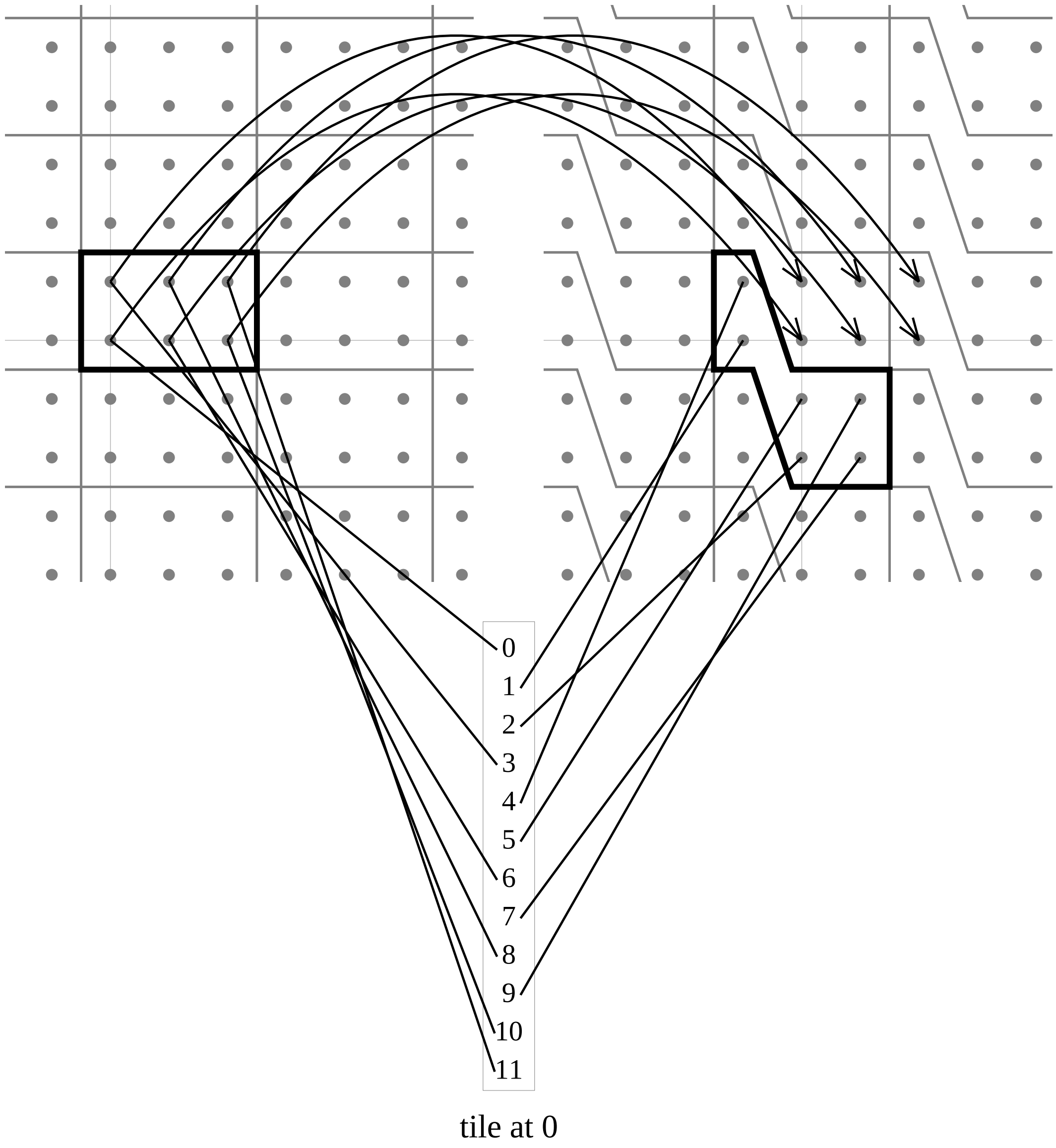

Figure 6.6: Using a bigger tile, temporary data consumed within a given tile stack always come from the immediately previous tile.

```
foreach (p in D) {
  int i = p[1], j = p[2], k = p[3];
  c[i, k] += a[i, j] * b[j, k];
}
```

Figure 6.7: Basic Titanium code for matrix multiplication.

and we require

$$\{x \mid (x - v) \in A_3\} \subseteq A_7 \quad .$$

That is not onerous since we were going to compute A_3 and A_7 anyway (figure 5.2).

As always, if the precheck fails at runtime then we use a second version of the code that is not as heavily optimized. In this case, two alternatives to adding to the precheck would be to copy potentially necessary values upon entry or to generate special startup tiles. Deferring part of the legality test to runtime increases the number of programs that we can optimize.

6.3 Delaying Writes

If the same location is written by every tile in a stack of tiles then we might benefit by loading that location to a register beforehand, using the register during the stack of tiles, and writing to the location once afterwards. This optimization is primarily for linear algebra codes such as matrix multiply (figure 6.7). In matrix multiply, we would apply this optimization if a stack of tiles in the generated code touched a fixed number of elements of `c`: each would be read into a scalar before the inner loop and written back after the inner loop (figure 6.8). All else being equal, performance gains can exceed 30%.

Theoretically this optimization could backfire if overused, due to increased register pressure. It may be wise to add parameters that allow fine control.

```
c0 = c[i, k];
c1 = c[i, k + 1];
c2 = c[i, k + 2];
c3 = c[i, k + 3];
for (j = jlo; j <= jhi; j++) {
  c0 += a[i, j] * b[j, k];
  c1 += a[i, j] * b[j, k + 1];
  c2 += a[i, j] * b[j, k + 2];
  c3 += a[i, j] * b[j, k + 3];
}
c[i, k] = c0;
c[i, k + 1] = c1;
c[i, k + 2] = c2;
c[i, k + 3] = c3;
```

Figure 6.8: Sample stack of tiles for matrix multiplication with delayed writes. There is no need to write to memory on each iteration.

```
can_read_from_register(optimizable_read O, int $k$):
  /* assume tile space $T = \mathbb{Z}^N \times \{0, \ldots, K-1\}$ */
  for (dist = 1; dist <= $K$; dist++) {
    int $k_{\mathrm{try}}$ = $k -$ dist;
    if ($k_{\mathrm{try}}$ < 0) {
      $k_{\mathrm{try}}$ += $K$;
      $\rho$ = $\overbrace{(0, \ldots, 0, -1)}^{N}$;
    } else {
      $\rho$ = $\overbrace{(0, \ldots, 0)}^{N}$;
    }
    S = the set of optimizable reads and writes in step $k_{\mathrm{try}}$;
    for every P in S {
      if (assuming the iteration space is all space,
          $\forall x$, step $k$ of the tile at $x$
          executes O to read a datum written or read by P,
          an optimizable read or write, executed at step $k_{\mathrm{try}}$ of
          the tile at $x + \rho$)
        return (P, dist);
    }

    if (any statement executed at step $k_{\mathrm{try}}$ of the tile at $x + \rho$
        could write to the location read by O)
      fail;
  }
  fail;
```

Figure 6.9: Analysis to avoid loading a recently read or written value that has not changed in the interim. If an appropriate `P` is found then we can save the value it reads or writes in some compiler-generated variable, e.g., `temp`, and change `O` from `v = A[...]` to `v = temp`.

6.4 Eliding Array Reads

6.4.1 Motivation

In tiled code, many values read from arrays have recently been read or written. That means the value is likely to be in cache, and the read is inexpensive. However, we would prefer to omit the read altogether if the value to be read is already in a register. Figure 6.9 shows our analysis to avoid loading a recently read or written value that has not changed in the interim. For each optimizable array read, we search up to a full tile backwards to see if the required value was read or written.

6.4.2 Implementation

The compile-time analysis shown reasonably assumes that the iteration space is infinite. In practice, we only perform this optimization for the loop of consecutive best-case tiles in the generated code (§5.4). A unique temporary variable is generated for each array read in the best-case tile that we choose to elide. A statement such as `v = A[...]` becomes `v = temp`. Just after the read or write returned by `can_read_from_register()` we insert an assignment to `temp`. If $\rho = (0, \ldots, 0)$ then the read of `temp` follows the appropriate write, and both always execute because they are in the same complete tile and `O` and `P` are optimizable. If $\rho = (0, \ldots, 0, -1)$ then we also insert an extra read, `temp = A[...]`, before the loop of consecutive best-case tiles. That extra read supplies the first tile in the best-case stack of tiles; the second tile in the stack gets the value of `temp` written in the first tile; and so on.

6.4.3 Register Pressure

For some programs, such as Gauss-Seidel relaxation, it turns out that most values are used multiple times in quick succession. In such cases, most possible applications of the transformation that elides array reads should be disregarded! Otherwise the number of temporaries introduced far exceeds the number of hardware registers, and performance plummets.

Figures 6.10 and 6.11 show our filtering mechanism. In short, we use a parameter (default 1) that limits the number of simultaneously live temporaries introduced

```
set<elision> filter_array_read_elisions():
  int $K$ = size of tile, count = 0;
  Let $\Omega$ = $\{0, \ldots, K-1\}$;
  for every $k \in \Omega$ {
    for every potential array read elision, e, in step $k_\mathrm{r}$
      Increment count;
      int $\delta$ = the distance in steps from the assignment of
                     the temporary to e;
      for every $b \in \Omega \cap (\{k_\mathrm{r} - \delta, \ldots, k_\mathrm{r}\} \cup \{K + k_\mathrm{r} - \delta, \ldots, K + k_\mathrm{r}\})$
        Increment step_to_num_live[$b$];
  }

  if (count is 0) return the empty set;
  int M = max {step_to_num_live[0], ..., step_to_num_live[$K-1$]};
  int max_live = get parameter$_1$ "Aggressiveness (0 to M)?";

  if (max_live >= M) return everything;
  set<elision> keep = empty set;
  if (max_live > 0) {
    map< step number, list<elision> > write_step_to_elisions;
    for every $k_\mathrm{r} \in \Omega$ {
      for every potential array read elision, e, in step $k_\mathrm{r}$
        int $\delta$ = the distance in steps from the assignment of
                       the temporary to e;
        int $k_\mathrm{w}$ = $k_\mathrm{r} - \delta$;
        if ($k_\mathrm{w}$ < 0) $k_\mathrm{w}$ += $K$;
        Add e to write_step_to_elisions[$k_\mathrm{w}$];
    }
    Reset all values in step_to_num_live to 0;
    for i from 1 to max_live
      /* Greedily select elisions, but don't allow any element of
         step_to_num_live[] to exceed i. */
      keep = keep $\cup$ filter_help(i, write_step_to_elisions,
                                  step_to_num_live);
  }
  return keep;
```

Figure 6.10: Throttling register pressure.

```
filter_help(int max_allowable,
            map< step number, list<elision> > write_step_to_elisions,
            map< step number, int > step_to_num_live):
  set<elision> selections = empty set;
 top:
  int seeking = 0;
  queue<step number> Q = empty queue;
  while (seeking < max_allowable && length(Q) < $K$) {
    for $k_w$ from 0 to $K-1$
      if (step_to_num_live[$k_w$] == seeking)
        append $k_w$ to Q;
    Increment seeking;
  }
  /* If y is early in Q and z is late in Q then
     step_to_num_live[y] <= step_to_num_live[z]. */
  while (Q is not empty)
    take $k_w$ from Q;
    for each u in write_step_to_elisions[$k_w$]
      if (suitable(u, $k_w$, max_allowable, step_to_num_live)) {
        Add u to selections;
        goto top;
      }

  return selections;

bool suitable(u, $k_w$, max_allowable, step_to_num_live):
  int $k_r$ = the step number of the read corresponding to u;
  int $\delta$ = $k_r$ - $k_w$;
  if ($\delta$ < 0) $\delta$ += $K$;
  for every $b \in \Omega \cap (\{k_r - \delta, \ldots, k_r\} \cup \{K + k_r - \delta, \ldots, K + k_r\})$
    if (step_to_num_live[$b$] + 1 > max_allowable) return false;
  for every $b \in \Omega \cap (\{k_r - \delta, \ldots, k_r\} \cup \{K + k_r - \delta, \ldots, K + k_r\})$
    Increment step_to_num_live[$b$];
  return true;
```

Figure 6.11: Helpers for filtering routine presented in the previous figure.

by this optimization. The performance of generated code when the parameter is chosen well is usually about 10–20% better than the performance when this optimization is disabled.

6.5 Conclusion

Storage optimizations are necessary for good performance. Tiling alone is not sufficient. Tilings chosen based on data dependences naturally tend to join the consumer(s) of a temporary array and the producer. Then, our aggressive and flexible array contraction algorithm might be able to remove or shrink the temporary array, compounding the gains of tiling and loop fusion.

However, storage optimizations also may backfire if we overload the faster levels of the memory hierarchy. A search of the parameter space is an excellent way to find the best performance, especially when generating C code rather than assembler.

Chapter 7

Parameter Selection

Selecting parameters such as tile sizes is difficult. Modern hardware is so complex that modelling it properly is nearly impossible.

Our method for parameter search is based on simulated annealing. The typical formulation of simulated annealing makes an analogy between minimizing the energy of a physical system and minimizing some function, $f(x)$. (In our case x is a set of parameters and choice of compilers, and $f(x)$ is a user-defined energy function.) An initial x and an initial temperature, T, are chosen. Then, at each step, x is randomly perturbed to x' and the new x' is accepted with probability $\min(1, e^{(f(x)-f(x'))/kT})$, where k is a constant. Every so often the temperature is decreased, and eventually, perhaps after some fixed number of steps, the process halts. There are many variants.

Our approach is to search with a predetermined stopping time. We let the temperature be $1-u$, where u is the fraction of time used so far. The time to perform some calculation is a typical choice of energy function. By default we scale the temperature by a moving average, $\bar{\kappa}$, of the positive values of $f(x') - f(x)$.

```
param_search(list<method> R, p_0, α, β, g, ...):
  filename ex = create an executable file from initial parameters;
  float τ̄ = τ_acc = time taken to do initial run of tc;
  parameter_vector x = g(output parameter file from tc);
  float e = get_energy(ex), κ̄ = -999;
  if (e is a NaN) fail;
  while (some time is left) {
    rule r = a randomly selected rule for perturbing parameters;
    parameter_vector x' = x perturbed according to rule r;
    if (x is not different from x') action = "misfire";
    else {
      float ξ = get_time_limit(τ̄, τ_acc);
      run tc with parameters x' and time limit of ξ;
      τ̄ = (1 - β) * τ̄ + β * time taken to do that run of tc;
      if (compile failed) action = "compiler failure";
      else if (tc tiled at least one loop in each method in R) {
        ex = create an executable from C code output by tc;
        e' = get_energy(ex);
        if (e' is a NaN) action = "runtime failure";
        else {
          if (e' > e)
            κ̄ = (κ̄ < 0) ? (e' - e) : ((1 - α) * κ̄ + α * (e' - e));
          action = accept(e, e', κ̄) ? "accept" : "decline";
        }
      } else action = "reject";
    }
    Depending on action, adjust weight for rule r;
    if (action is "accept") {
      e = e';
      x = g(output parameter file from latest run of tc);
      τ_acc = time taken to do latest run of tc;
    }
  }

bool accept(e, e', κ̄):
  if (e' < e) return true;
  float u = (time used) / (time used plus time left);
  return true with probability p_0^((e'-e)/((1-u)κ̄));
```

Figure 7.1: Pseudocode for simulated annealing to select parameters. Output parameter files are filtered by g, a user-specified function. `get_energy()`, not shown, passes the name of an executable file to a user-specified shell script. α and β default to 0.1; p_0 to 0.25.

Rule	Description
`set_param` χ v	set a parameter matching χ to v
`set_param_all` χ v	set all parameters matching χ to v
`remove_param` χ	remove a parameter matching χ from the set of parameters to be specified
`toggle_param` χ	toggle a parameter matching χ
`increase_param` χ	increase a parameter matching χ by 1
`increase_param` χ n	increase a parameter matching χ by n
`increase_param` χ l h	increase a parameter matching χ by a random element of $\{l, \ldots, h\}$
`multiple` n	recursively fire n rules
`modify_param_by_multiplication` χ n	multiply a parameter matching χ by n
`change_any`	randomly select any parameter and double it, halve it, toggle it, or add some element of $\{-5, \ldots, 5\}$
`set_compiler` n	use compiler number n

Table 7.1: Format for rules to perturb parameters: χ represents a regular expression, and other variables represent numbers. The only way to change compilers is with `set_compiler`. Other than the choice of compiler, parameters are assumed to be a set of string/value pairs.

The probability of accepting x' is

$$p_0^{(f(x')-f(x))/((1-u)\bar{\kappa})} \quad ,$$

where p_0 defaults to $\frac{1}{4}$. So, the probability of accepting a move in parameter space that is about as bad as other recent potential bad moves is $p_0^{\frac{1}{1-u}}$. Figure 7.1 is pseudocode for the whole process. The output of a parameter search is a lengthy and detailed log. Often, in practice, the parameters that resulted in the best energy are the only part of the log not discarded.

A set of rules for perturbing parameters must be provided in the format shown in table 7.1. The user gives initial weights to each rule and controls how a rule's weight changes when its invocation leads to a particular action. (By action we mean "accept", "decline," and so on, as in figure 7.1.)

Parameters are typed, so one of the ways a rule can misfire is by yielding an invalid value. For example, `change_any` might try to add 5 to a parameter whose valid range is 0 to 3. However, despite the parameters' types, rules do apply as broadly as possible. For example, `modify_param_by_multiplication` does rounding after the multiplication, and `toggle_param` works on all parameters (not just booleans) by mapping 0 to 1 and any non-zero to 0.

Each invocation of `tc` during a parameter searching run has a time limit. This is to avoid overrunning the predetermined stopping time and to allow users some control over the minimum number parameter vectors explored. Let `rand`() be a function that returns a random number between 0 and 1. A command-line argument can specify $\langle z, x, a, b \rangle$ tuples, each of which yields a cap of

$$\max \{z\tau_{\mathrm{acc}}, x, (au + b)/\texttt{rand}()\} \quad .$$

Another command-line argument specifies $\langle z, y, c, d \rangle$ tuples, each of which yields a cap of

$$\max \{z\tau_{\mathrm{acc}}, y\bar{\tau}, (cu + d)\bar{\tau}/\texttt{rand}()\} \quad .$$

In both cases, caps are expressed in seconds. A fresh set of caps is computed each time `tc` is to be invoked. If `tc` takes longer than the smallest cap, or runs until the predetermined stopping time, then it is halted.

Chapter 8

Results

8.1 Introduction

The Titanium programs that we tested on uniprocessors are `ca`, `s3`, `rb9`, `rbrb9`, and `mg`. The first two (§8.2) are 1D stencil codes, one traditional and one slightly less so. The next two (§8.3) are based on Gauss-Seidel Red-Black (GSRB) in a C++/FORTRAN implementation of Anderson's Method of Local Corrections (MLC) by Phil Colella and Paul N. Hilfinger. Their implementation performs two red-black passes in a row in several places, and `rbrb9` is just that. (According to Sellappa and Chatterjee [33], related codes do as many as eight in a row.) The red-black-red-black pattern is written as eight loops because the code uses a 9-point stencil in 2D. We merge all eight loops into one. For purposes of comparison, `rb9` is the same but performs only one red and one black pass (four loops). A V-cycle of 3D multigrid, based on Titanium code for Adaptive Mesh Refinement ([32]) by Luigi Semenzato, is our largest benchmark, `mg`. We present results for `mg` in §8.4.

Most testing was done on two PCs running Linux. One is a 866MHz Intel Pentium III Coppermine, and one is a 1.4GHz AMD Athlon Thunderbird. The

```
/* x is the input and the output; y and z are temporaries. */
/* t is a fixed table of M elements. */
foreach (p in d)
  y[p] = t[(a * x[p - [2]] + b * x[p - [1]] + c * x[p] +
            d * x[p + [1]] + e * x[p + [2]]) % M];
foreach (p in d)
  z[p] = t[(a * y[p - [2]] + b * y[p - [1]] + c * y[p] +
            d * y[p + [1]] + e * y[p + [2]]) % M];
foreach (p in d)
  x[p] = t[(a * z[p - [2]] + b * z[p - [1]] + c * z[p] +
            d * z[p + [1]] + e * z[p + [2]]) % M];
```

Figure 8.1: Pseudocode for `ca`.

latter uses both gcc and icc (Intel's C compiler) while the former uses gcc only. We also present a few results on Sun UltraSPARCs using gcc. (`tc` generates C code.) All problem sizes were chosen to fit comfortably in main memory but not in cache.

8.2 1D Benchmarks

We begin with two programs that operate on 1-dimensional arrays. Red-black relaxation is useful in any number of dimensions; `s3` applies a 3-point stencil first to the red points of an array of doubles, and then to the black points. Only one array is necessary, and half its values are updated by each of the two loops. A program to simulate a cellular automaton is our other 1-dimensional benchmark, `ca`. Its basic operation is to calculate an array index via a 5-element integer dot product modulo `M`. Pseudocode is shown in figure 8.1. Stoptifu can fuse the three loops and contract the temporary arrays `y` and `z` to scalars.

Results for `s3` are presented in table 8.1. All running times are presented to three significant figures, and are the minimum wall-clock time of five runs. Millions of double-precision floating-point operations per second (MFLOPS) are

	Pentium III		UltraSPARC	
Effort	runtime (s)	MFLOPS	runtime (s)	MFLOPS
baseline	0.790	50.6	1.84	21.7
0h	0.504	79.4	2.36	16.9
1h	0.413	96.9	1.70	23.5
2h	0.394	102	1.67	24.0
4h	0.329	122	1.59	25.2
8h	0.324	123	1.49	26.8

Table 8.1: Results for `s3` on 866MHz Pentium III and on 167MHz UltraSPARC. Each result represents an independent search that started from scratch, so it is possible for an unlucky long search to underperform a lucky short search. The baseline is compiling without Stoptifu but with all other optimizations. The 0h line shows running times after compiling with Stoptifu's default parameters. Longer searches yielded no further improvement.

also presented to three significant figures. The baseline results, by which we mean results achieved without Stoptifu but with all other optimizations, are 0.790s for the 866MHz Pentium III and 1.84s for the 167MHz UltraSPARC. The results are as expected, except that, on the UltraSPARC, the default Stoptifu parameters led to a time worse than the baseline. Our system improves the runtime of `s3` by a factor of 2.44 on the Pentium machine and 1.23 on the Sun. Parameter searches longer than eight hours did not yield any further improvement.

The benefits of decreasing memory traffic are more pronounced on the Pentium because its processor speed to memory speed ratio is higher. The Sun's CPU is clocked only twice as fast as its memory bus; the Pentium III's CPU is clocked 6.5 times as fast as its memory bus. (The Sun is several years older.) The Sun also searches parameter space relatively slowly because equivalent compilations take longer.

Results for `ca` are presented in table 8.2. Without Stoptifu but with all other optimizations, the baseline times are 7.66s for the 167MHz UltraSPARC and 2.09s for the 866MHz Pentium III. The Pentium data are for a problem size five times

	Pentium III		UltraSPARC	
Effort	runtime (s)	op/μs	runtime (s)	op/μs
baseline	2.09	158	7.66	8.62
0h	1.41	234	6.86	9.62
1h	1.08	306	6.79	9.72
2h	1.04	317	6.81	9.69
4h	1.02	324	6.78	9.73
8h	0.983	336	6.49	10.2

Table 8.2: Results for `ca` on 866MHz Pentium III and on 167MHz UltraSPARC. The 0h line shows running times after compiling with Stoptifu's default parameters. Longer searches yielded no further improvement.

as large. Once again, a two hour search on the Pentium doubles the baseline performance. Improvement on the UltraSPARC is noticeable but less spectacular.

8.3 Gauss-Seidel Relaxation in 2D

Table 8.3 presents the results for `rb9`. All results for `rb9` come from the minimum wall-clock time of five runs, presented to three significant figures. Running with Stoptifu's default parameters, a time of 1.92 seconds was achieved. The baseline time is 3.38 seconds (all of `tc`'s optimizations except Stoptifu). The primary results here are:

- `tc` with Stoptifu's defaults yields code 1.76 times faster than the baseline.
- `tc` with better Stoptifu parameters (`rb9`$\bar{s}i$55) yields code another 1.52 times faster, i.e., 2.68 times faster than the baseline.
- The best we were able to do with array contraction enabled was 1.25 times faster than the best we were able to do without it.
- Allowing or disallowing any tile shape matters little (a few percent).

Name	Allow any shape	Coarse inter-leaving	Array con-traction	effort	Runtime after search with gcc (s)	MFLOPS
baseline	N/A	N/A	N/A	N/A	3.38	49.7
no search	yes	yes	yes	0h	1.92	87.5
rb9$s\imath\bar{a}$72	yes	yes	no	72h	1.69	99.4
rb9$\bar{s}\imath\bar{a}$72	no	yes	no	72h	1.71	98.2
rb9$s\bar{\imath}\bar{a}$72	yes	no	no	72h	1.60	105
rb9$\bar{s}\bar{\imath}\bar{a}$72	no	no	no	72h	1.66	101
rb9$s\imath a$72	yes	yes	yes	72h	1.32	127
rb9$\bar{s}\imath a$72	no	yes	yes	72h	1.37	123
rb9$s\bar{\imath}a$72	yes	no	yes	72h	1.34	125
rb9$\bar{s}\bar{\imath}a$72	no	no	yes	72h	1.37	123
rb9$s\imath$55	yes	yes	both	55h	1.28	131
rb9$\bar{s}\imath$55	no	yes	both	55h	1.26	133
rb9$s\bar{\imath}$55	yes	no	both	55h	1.34	125
rb9$\bar{s}\bar{\imath}$55	no	no	both	55h	1.38	122

Table 8.3: Results for `rb9` on 866MHz Pentium III. See §4.4.2 for an explanation of "allow any shape" and "coarse interleaving." Array contraction was free to be enabled or disabled during the last four searches, but all of them settled on parameters that use it.

- Allowing or disallowing fine interleaving of nodes in a tile matters little (a few percent).

Table 8.4 presents the results for `rbrb9`. All results for `rbrb9` come from the minimum wall-clock time of five runs, presented to three significant figures. During searches the backend compiler was free to switch between gcc and icc, but at the end of each run we tried both, using that search's best reported `tc` parameters. The baseline times, without Stoptifu, are 3.73 seconds (gcc) and 3.70 seconds (icc). With Stoptifu and its default parameters, that improves to 2.48 seconds (gcc) and 2.36 seconds (icc). Nine of ten results favor icc, so here we summarize just the highlights of the icc results:

- `tc` with Stoptifu's defaults yields code 1.57 times faster than the baseline.

Name	Allow any shape	Coarse inter-leaving	Array con-traction	effort	Runtime after search: with gcc	icc	best MFLOPS
baseline	N/A	N/A	N/A	N/A	3.73	3.70	90.8
no search	yes	yes	yes	0h	2.48	2.36	142
`rbrb9`sia55	yes	yes	yes	55h	1.56	1.48	227
`rbrb9`$\bar{s}ia$55	no	yes	yes	55h	1.59	1.44	233
`rbrb9`$s\bar{\imath}a$55	yes	no	yes	55h	1.71	1.45	232
`rbrb9`$\bar{s}\bar{\imath}a$55	no	no	yes	55h	1.68	1.61	209
`rbrb9`$si\bar{a}$55	yes	yes	no	55h	1.81	1.73	194
`rbrb9`$\bar{s}i\bar{a}$55	no	yes	no	55h	1.49	1.45	232
`rbrb9`$s\bar{\imath}\bar{a}$55	yes	no	no	55h	1.83	3.21	184
`rbrb9`$\bar{s}\bar{\imath}\bar{a}$55	no	no	no	55h	1.71	1.68	200
`rbrb9`a120	both	both	yes	120h	1.42	1.34	251
`rbrb9`$\bar{a}$120	both	both	no	120h	1.49	1.44	233
`rbrb9`120	both	both	both	120h	1.30	1.22	275

Table 8.4: Results for `rbrb9` on 1.4GHz Athlon. For all ten searches, the C compiler used was free to switch between gcc and icc. At the end, whatever parameters were selected were used with each C compiler, for comparison purposes.

- `tc` with better Stoptifu parameters (`rbrb9120`) yields code another 1.93 times faster, i.e., 3.03 times faster than the baseline.
- The best we were able to do with array contraction enabled was 1.18 times faster than the best we were able to do without it.
- Allowing or disallowing any tile shape matters little (a few percent).
- Allowing or disallowing fine interleaving of nodes in a tile matters little (a few percent).

8.4 Multigrid

There are numerous variations on multigrid (e.g., Briggs [8]), and many of them are amenable to our system of optimization. Multigrid algorithms that spend the majority of their time performing GSRB or other linear relaxation methods are common. Sellappa and Chatterjee [33] show a multigrid program that spends 80% or more of its running time doing GSRB. Using our results from the previous section, we would improve the performance of a program that spends 80% of its time in GSRB by more than a factor of two.

The program `mg` is interesting because it contains several different loops that have different opportunities for optimization. The majority of the time is spent on GSRB with a 7-point stencil in 3D, for which no temporary storage is needed. In fact, a naïve compiler does relatively well on this code. But loop fusion, tiling, and storage optimizations still improve `mg`in interesting ways.

We can contract only one array in `mg`. A residual is calculated and immediately used to correct the right-hand side of the next coarser level. To expose the temporary residual to contraction we manually inlined part of the recursive call

Effort	runtime (s)	MFLOPS	ratio to baseline
baseline	2.72	100	1
0h	2.62	104	1.04
120h	2.17	125	1.25
120h+8h	2.01	135	1.35

Table 8.5: Results for `mg`.

in the V-cycle. As expected, our heuristic for inducing the fusion of loops (§4.4) combines "coarse" and "fine" loops in the necessary 1:8 ratio to allow contraction. Even better, it is able to find one set of three loops that it combines in a 1:8:64 ratio.

While not as spectacular as some of the other results, both array contraction and parameter search were necessary to do that well. The best results were obtained by doing a 120 hour search that was unconstrained, then manually profiling the code and adding a further 8 hour search that only modified parameters for the most important Titanium method in the source code. The latter search used the best result from the 120 hour search as its initial position in parameter space.

8.5 Discussion

The Stoptifu library is capable of merging any number of loops by the methods described in §4. Furthermore, by using an algorithm that is driven by data dependences, the loops will usually be merged in a way that increases both temporal locality and the opportunity for storage optimizations such as array contraction. Stoptifu allows the contraction of an array to scalar(s), to lower-dimensional array(s), or to both, as necessary. The combination of these properties makes our compiler's output resemble the state-of-the-art in hand-optimization of multigrid algorithms. No other compiler can do as well.

The ability to automatically search a space of compiler parameters is also crucial to our performance results. The difference between good parameters and indifferent ones often makes a 50% difference in the performance of generated code.

Interestly, the best result obtained with coarse interleaving of nodes from different loops roughly equalled the best result obtained with fine interleaving. That is somewhat surprising because toggling that one decision in any particular parameter vector frequently has a noticeable effect. The same is true of allowing any tile shape. Overall, it appears that having those options available is worthwhile. There certainly exist local performance maxima in parameter space that require coarse interleaving, or that require fine interleaving. Similarly, there exist local performance maxima in parameter space that require allowing any tile shape, or that require disabling that option.

Finally, it is interesting to note the drawbacks of a parameter search that constrains array contraction to be perpetually enabled or disabled. In the latter case the performance boost of array contraction is not realized, while in the former case compile times are higher, so less of the parameter space is explored. (The same pitfall can apply to any beneficial transformation that increases compile time.) This explains how, for example, `rb9`$\bar{s}i55$ found better parameters than either of two more CPU-intensive competitors, `rb9`$\bar{s}ia72$ and `rb9`$\bar{s}i\bar{a}72$.

Chapter 9

Parallel Execution

9.1 Introduction

Given a tiling, it is easy to reason about the ordering constraints among tiles. Then the tiles can be assigned to different processes for parallel execution, or the tile space can be reordered (yielding a hierarchical tiling), or both.

Hierarchical tiling is future work for us, but we have implemented a limited form of automatic parallelization. In the context of Titanium, an explicitly parallel language, this is slightly odd. However, we felt it important to show how to extend our sequential optimizations to the ever more important realm of parallel computing. This chapter is essentially a proof-of-concept.

9.2 Implementation

Most stencil codes cannot be trivially parallelized: a pipeline or wavefront scheme must be employed. The wavefront scheme divides tile space with a set of parallel hyperplanes, and does each slice of computation between adjacent planes in parallel. In fused and tiled GSRB, for example, it starts at one corner of the bounding box of the runtime tile space and proceeds to the opposite corner. Global barriers periodically synchronize all processes so that ordering constraints between tiles are not violated. Initially only one process is busy, but after a few barriers all are usually busy (assuming a large, rectangular tile space). The pipeline scheme is similar but it uses point-to-point communication to synchronize processes working on adjacent stacks of tiles. The pipeline scheme is generally preferable (according to Lim and Lam [28]), so we have not yet implemented the wavefront scheme.

We have implemented basic automatic parallelization for shared-memory machines at the level of a tile space. A tile space to be executed in parallel is divided into stacks of tiles as usual, but the stacks are assigned cyclically to processes. For simplicity and to prevent deadlocks, each process executes all its tiles in standard order. Ordering constraints between different stacks, if any, are enforced with point-to-point communication. This amounts to the pipeline scheme except, of course, communication is not necessary in the embarrassingly parallel case.

To allow the programmer to indicate loops to be parallelized, we introduce a new syntax to Titanium: `foreach (p in D) parallel` B, where B is a block of zero or more statements. (This syntax was chosen to avoid adding a keyword. The syntactic position of `parallel` does not require that it be a keyword.) It is an error if one process arrives at a particular textual instance of said construct but another does not, and it is also an error if the processes do not agree on the value of `D`. In Titanium jargon, `D` must be single-valued, and the construct has global effects (Aiken and Gay [1]).

Inside the Stoptifu library, tiling proceeds as usual except that we never tile together `parallel` and non-`parallel` loops. If one or more `parallel` loops have been mapped to a tile space, $T = \mathbb{Z}^N \times \{0, \ldots, K-1\}$, we apply the algorithm of figure 9.1 to determine how to proceed. If `parallelize()` succeeds then it returns

```
set<vector> parallelize(vector z, list<OrderingConstraint> oc, ...):
 Let $s(k)$ = $\{x \mid$ tile at z $+\, x$ must precede $\langle \mathtt{z}, k \rangle$ according to oc$\}$;
 set<vector> required = $\bigcup_{k=0}^{K-1} s(k)$;
 if (iteration is 1D and required is non-empty) fail;
 /* required may be infinite.  Discard all but the
    last element of each stack of tiles. */
 Let same_stack = $\{(0,\ldots,0,0),(0,\ldots,0,\pm 1),(0,\ldots,0,\pm 2),\ldots\}$;
 subsumed = $\{q \mid \exists p \in \mathtt{required} \wedge q \prec p \wedge (q-p) \in \mathtt{same\_stack}\}$;
 required = required $\setminus$ subsumed;
 if (required is infinite) fail;
 /* The elements of required may be viewed as dependence vectors.
    At runtime, the union of partial tile stacks ending at each
    element are a sufficient condition for the tile at z to execute.
    Do those conditions suffice for a generic tile at $x$ as well? */
 Let $f(x)$ = $\bigcup_{r \geq 0} \{x + p - (0,\ldots,0,r) \mid p \in \mathtt{required}\}$;
 Let $\Lambda$ = $\{(a,b) \mid$ an ordering constraint in oc requires $a \prec b\}$;
 if ($\exists x \exists y$ such that $y \notin f(x) \wedge (y,x) \in \Lambda$) fail;
 return required;
```

Figure 9.1: Pseudocode for computing dependences in tile space relevant to automatic parallelization. We use $\mathtt{z} = 0$, and we filter the ordering constraints because they should not contain constraints within a stack of tiles.

a list of dependence vectors in $\mathbb{Z}^N$. If the list is empty then no synchronization is necessary. Otherwise, at runtime, a spin lock before each tile prevents it from prematurely executing.

Once proper synchronization is enforced, the only other necessary change is to array contraction. The number of simultaneously live values in a contracted array depends on whether multiple processes are working in parallel. Arrays values contracted to scalars cannot be live outside of a tile or stack of tiles and are therefore unaffected. Values in contracted arrays that would (in the sequential case) not be stored in scalars need special treatment because those data are moving from one stack of tiles to another. We simply assume that all stacks of tiles might be simultaneously in progress. Separate storage is therefore allocated for each stack. As a result, array contraction in automatically parallelized codes does not reduce the amount of temporary storage quite as much. This effect is mitigated by data reuse within a tile stack.

9.3 Results

Parallel results are encouraging. The running time for a parallelized loop run on one processor can be within 1% of the same loop without parallelization. Thus, we know that the overhead imposed by memory fences is minor. In the embarrassingly parallel case, we acheived speedups up to 3.9 on 4 processors (data not shown). Data for six pipeline-parallel GSRB codes are shown in table 9.1. We did not run the parameter search script. We simply present results from six different tile shapes. Two programs are the same as from the previous chapter, but with the `parallel` directive inserted. A third is the same as `rb9`, but with only a 5-point stencil. And the last three programs (`rbrb9h`, `rb9h`, and `rb5h`) are the same as

Program	Tile shape	time on 1 proc (s)	time on 2 procs (s)	time on 4 procs (s)
`rb5`	1×20	1.678	2.029	1.547
`rb5`	2×10	1.756	1.501	1.026
`rb5`	4×5	1.75	1.506	1.011
`rb5`	5×4	1.782	1.461	.963
`rb5`	10×2	1.767	1.454	.936
`rb5`	20×1	1.822	1.337	.901
`rb9`	1×20	3.658	3.202	3.18
`rb9`	2×10	2.971	3.122	2.403
`rb9`	4×5	2.675	2.306	1.7
`rb9`	5×4	2.631	2.125	1.576
`rb9`	10×2	2.554	1.793	1.285
`rb9`	20×1	2.384	1.592	1.031
`rbrb9`	1×20	13.581	9.819	8.043
`rbrb9`	2×10	9.889	6.717	4.655
`rbrb9`	4×5	7.385	4.828	3.069
`rbrb9`	5×4	6.904	4.508	2.783
`rbrb9`	10×2	5.955	3.823	2.249
`rbrb9`	20×1	5.822	3.833	2.142
`rb5h`	1×20	7.721	5.157	2.789
`rb5h`	2×10	7.801	4.829	2.555
`rb5h`	4×5	7.856	4.627	2.416
`rb5h`	5×4	7.797	4.825	2.553
`rb5h`	10×2	8	4.782	2.491
`rb5h`	20×1	7.948	4.404	2.273
`rb9h`	1×20	9.991	7.041	4.54
`rb9h`	2×10	9.418	5.552	2.949
`rb9h`	4×5	9.48	5.77	3.174
`rb9h`	5×4	9.42	5.553	2.963
`rb9h`	10×2	9.415	5.534	2.958
`rb9h`	20×1	9.255	5.223	2.715
`rbrb9h`	1×20	25.916	16.105	10.361
`rbrb9h`	2×10	22.052	13.028	7.373
`rbrb9h`	4×5	19.531	11.175	6.021
`rbrb9h`	5×4	18.97	11.051	5.888
`rbrb9h`	10×2	17.978	9.818	5.153
`rbrb9h`	20×1	17.741	10.217	5.291

Table 9.1: Parallel results for a 4-way 700MHz Pentium III SMP. Results are wall-clock time, to the millisecond. These are the best times from five runs. The wide tiles, such as $20x1$, require less overhead and less communication.

the first three, but with additional arithmetic operations at each stencil point. In some cases the speedups are below three on four processors, but that just indicates a high communication-to-computation ratio, not anything wrong with our implementation. In the best case the speedups are close to optimal, particularly when arithmetic per memory access is high. We confirmed that synchronization was not the problem by manually removing synchronization primitives: performance changed only a few percent (data not shown). Adding more arithmetic per memory operation, on the other hand, improved the speedup quite noticeably.

Chapter 10

Conclusion

10.1 Future Work

Adding more optimizing transformations is the single most important improvement to make. In particular, we would like to allow multiple special-case tiles as mentioned in §5.4. It might also be important to generate special cases for common array layouts (e.g., unit-strided row-major order) to offset the generality of Titanium's strided, multidimensional arrays.

Improving the speed and ease-of-use of parameter search is also important. In a few years we hope that a combination of improved software and Moore's Law will make multi-day searches be the exception rather than the rule.

10.2 Highlights

We have presented a system that combines new and old techniques for compilation and parameter search. This dissertation is a step towards the goal of automatically creating highly-tuned scientific programs directly from source code written in a high-level programming language.

For now, the programmers we have in mind are willing to spend some time tuning their code and their compiler parameters. Given that, and the difficulty in statically selecting parameters such as tile sizes, it makes sense to provide automatic parameter searching alongside the compiler. Furthermore, including automatic parameter searching logically leads one to include more aggressive and speculative optimizing transformations in the compiler. Our philosophy is to optimize aggressively but to expose the compiler's decisions to external control. Since we expect to generate numerous executables during the tuning process, optimizations that *may* pay off are relatively more important.

One consequence of our philosophy is that deciding what optimizations to use should be partially deferred to runtime. If an optimization may be safe or may be beneficial, in some cases we generate multiple versions of a section of source code and decide which version to use at runtime. By deferring decisions to runtime we can optimize more programs because we are not as limited by the quality or quantity of static compiler analyses.

Some code bloat is common to all tiling transformations, and our aggressive stance does nothing to discourage it. We value speed too highly to be deterred by a few kilobytes of additional machine code. Our generated code displays all the typical benefits of tiling, including decreases in memory usage, cache usage, cache misses, and instructions executed. Running time is sometimes two or more times faster than with a basic optimizing compiler. Sequential running time for multigrid should also be at least 10% faster than competing approaches due to our superior array contraction and our willingness to spend a few CPU days searching for a good set of parameters.

Bibliography

[1] A. Aiken and D. Gay. Barrier Inference. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 342–354, San Diego, CA, 1998.

[2] Christopher R. Anderson. *An Implementation of the Fast Multipole Method Without Multipoles.* UCLA Report CAM 90-14, Dept. of Mathematics, UCLA, Los Angeles, CA, July 1990.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Tools, and Techniques.* Addison-Wesley, 1986.

[4] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4):345–420, December 1994.

[5] U. Banerjee. Unimodular transformations of double loops. In *Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 192–219, Irvine, CA, 1990.

[6] BeBOP. `http://www.cs.berkeley.edu/~richie/bebop/`.

[7] J. Bilmes et al. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proc. ICS'97*, pages 340–347, 1997.

[8] W. L. Briggs. *A Multigrid Tutorial.* SIAM, 1987.

[9] Doug Burger, James R. Goodman, and Alain Kägi. Quantifying memory bandwidth limitations of current and future microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, 1996.

[10] Larry Carter, Jeanne Ferrante, Susan Flynn Hummel, Bowen Alpern, Kang-Su Gatlin. *Hierarchical Tiling: A Methodology for High Performance.* UCSD Technical Report CS96-508, November 1996.

[11] C. C. Douglas et al. Maximizing Cache Memory Usage for Multigrid Algorithms. In Z. Chen, R. E. Ewing and Z.-C. Shi, editors, *Multiphase Flows and Transport in Porous Media: State of the Art*, Springer-Verlag, Lecture Notes in Physics, Berlin, 2000.

[12] FFTW. `http://www.fftw.org/`.

[13] D. Gay and A. Aiken. Memory Management with Explicit Regions. *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998.

[14] P. N. Hilfinger et al. *Titanium Language Reference Manual*. Technical Report CSD-01-1163, Computer Science Division, University of California, Berkeley, 2001.

[15] Karin Högstedt, Larry Carter, and Jeanne Ferrante. Determining the Idle Time of a Tiling. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, January 1997.

[16] S. Flynn Hummel, I. Banicescu, C. Wang, and J. Wein. Load Balancing and Data Locality via Fractiling: An Experimental Study. In Boleslaw K. Szymanski and Balaram Sinharoy, editors, *Proc. Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 85–89. Kluwer Academic Publishers, Boston, MA, 1995.

[17] Eun-Jin Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. Ph.D. thesis, University of California, Berkeley, 2000.

[18] Eun-Jin Im and Katherine Yelick. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. *International Conference on Computational Science*, 2001.

[19] Wayne Anthony Kelly. *Optimization Within a Unified Transformation Framework*. Ph.D. thesis, University of Maryland, 1996.

[20] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Library interface guide*. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, March 1995.

[21] T. Kisuki, P. M. W. Knijnenburg, K. Gallivan, and M. F. P. O'Boyle. The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling. In *Proc. FDDO-3*, pages 31-40, 2000.

[22] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. *Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation*. Technical Report 2000-07, LIACS, Leiden University, 2000.

[23] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric Multi-level Blocking. In *SIGPLAN 1997 conference on Programming Language Design and Implementation*, June 1997.

[24] Arvind Krishnamurthy. *Compiler Analyses and System Support for Optimizationing Shared Address Space Programs*. Ph.D. thesis, University of California, Berkeley, 1998.

[25] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[26] B. Liblit and A. Aiken. Type systems for distributed data structures. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 199–213, Boston, MA, 2000.

[27] B. Liblit, A. Aiken, and K. Yelick. Data Sharing Analysis for Titanium. Technical Report CSD-01-1165, Computer Science Division, University of California, Berkeley, 2001.

[28] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24:445–475, 1998.

[29] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[30] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[31] M. F. P. O'Boyle, P. M. W. Knijnenburg, and G. G. Fursin. *Feedback Assisted Iterative Compilation*. Preprint, 2000.

[32] G. Pike, L. Semenzato, P. Colella, P. Hilfinger. Parallel 3D Adaptive Mesh Refinement in Titanium. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.

[33] Sriram Sellappa and Siddhartha Chatterjee. Cache-Efficient Multigrid Algorithms. In *Proceedings of the 2001 International Conference on Computational Science (ICCS 2001)*, San Francisco, CA, May 2001.

[34] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1998.

[35] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A Unified Framework for Schedule and Storage Optimization. In *Proceedings of the 2001 SIGPLAN Conference on Programming Language Design and Implementation.*

[36] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT CS-97-366, LAPACK Working Note No. 131, University of Tennessee, 1997.

[37] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[38] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the 3rd SIAM Conference on Parallel Processing*, 1987.

[39] Michael Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, pages 655–665, 1989.

[40] David G. Wonnacott. *Constraint-Based Array Data Dependence Analysis.* Ph.D. thesis, Dept. of Computer Science, University of Maryland, August 1995.

[41] K. Yelick et al. Titanium: A High-Performance Java Dialect. In *Proceedings of the ACM 1998 Workshop on Java for High Performance Network Computing*, Stanford, CA, February 1998.