Better Tiling and Array Contraction for Compiling Scientific Programs *

Geoff Pike and Paul N. Hilfinger
Computer Science Division
University of California at Berkeley
{pike,hilfinger}@cs.berkeley.edu

Abstract

Scientific programs often include multiple loops over the same data; interleaving parts of different loops may greatly improve performance. We exploit this in a compiler for Titanium, a dialect of Java. Our compiler combines reordering optimizations such as loop fusion and tiling with storage optimizations such as array contraction (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. Our strategy is to optimize aggressively but to expose the compiler's decisions to external control. We double or triple the performance of Gauss-Seidel relaxation and multigrid (versus an optimizing compiler without tiling and array contraction), and we argue that ours is the best compiler for that kind of program.

1 Introduction

Many scientific kernels, such as multigrid, FFT, and matrix multiplication, are easy to write in a page or two of code. To get state-of-the-art performance, however, two conditions must be met: appropriate optimizing transformations must be applied, and parameters such as tile sizes must be chosen well. Both conditions are met automatically, for some specific kernels, by special-purpose packages (e.g., PHiPAC [1], ATLAS [29], and FFTW [7]). Our work is a step towards meeting those two conditions automatically—for any program—in a general-purpose compiler.

In particular, our compiler is better than previous ones at optimizing Gauss-Seidel relaxation and multigrid, and we offer an optional simulated-annealing search over tile sizes and other parameters. Thus, we allow programmers to code straightforwardly but still achieve great performance.

State-of-the-art transformations used in the hand-optimization of multigrid (e.g., Douglas et al. [6] and Sellappa and Chatterjee [24]) inspired the novel compiler optimizations that we describe below. The resulting system is innovative for its ability to tile and fuse loops in a natural and non-restrictive way (section 3). When multiple loops are tiled and fused together, efficient data reuse and array contraction can be crucial to performance. Our variant of array contraction (section 4) is similar to that of Strout et al. [26], but it can realize much greater savings.

This paper summarizes highlights of the first author's dissertation [22]. Much more detail, including extensions for the parallel case, may be found there. Here, we focus on sequential compiler optimizations and assessments of their effectiveness.

2 Titanium

Our compiler optimizations are applicable to any language commonly used for scientific programming. As a concrete prototype, we are using Titanium, a dialect of Java developed at Berkeley. Details of the language are available in a reference manual [8]. Fig-

*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 grant ACI-9619020, 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.
0-7695-1524-X/02 $17.00 (c) 2002 IEEE

```
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 1: Example of Titanium code amenable to fusion, tiling, and array contraction.

ure 1 is a sample snippet of Titanium code.

For our purposes here, the salient language characteristics are:

- Titanium is an explicitly parallel dialect of Java.
- 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. A `foreach` executes its body, `S`, $|D|$ times with p bound to each element of D in an unspecified order.

3 Tiling and Fusing Loops

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—usually corresponding to the flow of data. If 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 can produce 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 if (cheap) runtime tests are necessary to ensure safety: we require no static information about loop bounds, array bounds, or array aliasing. (An always-safe version of optimistically transformed code is also generated, in case the runtime test fails.) Parameterization allows 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 programmer-specified fitness criterion that is unknown to `tc`, the Titanium compiler. "Fitness" 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.

3.1 Formalism

For an N-dimensional `foreach`, we define the *tile space* T with *tile size* K to be the cross product $\mathbb{Z}^N \times \{0, \ldots, K-1\}$. The k^{th} *step* of the *tile at* $\vec{x}$ is the point $\langle \vec{x}, k \rangle \in T$. The *tile at 0* means the tile at $[0, \ldots, 0]$. We specify a loop reordering by a total order on T, $\prec$, and a bijection, $C{:}\, T \leftrightarrow \mathbb{Z}^N$. That is, the codomain of C serves as the loop's *iteration space,* the domain of the loop control variable. For q and q' in tile space and $C(q)$ and $C(q')$ in the loop's iteration space, define $C(q) \prec C(q')$ iff $q \prec q'$. Our implementation always uses lexicographic order on T. A *tile* is a set of points $\langle \vec{x}, 0 \rangle, \ldots, \langle \vec{x}, K-1 \rangle$ in T or the corresponding set in the loop's iteration space, $C(\langle \vec{x}, 0 \rangle), \ldots, C(\langle \vec{x}, K-1 \rangle)$.

Figure 2 illustrates the above formalism. It extends straightforwardly to the multi-loop case.

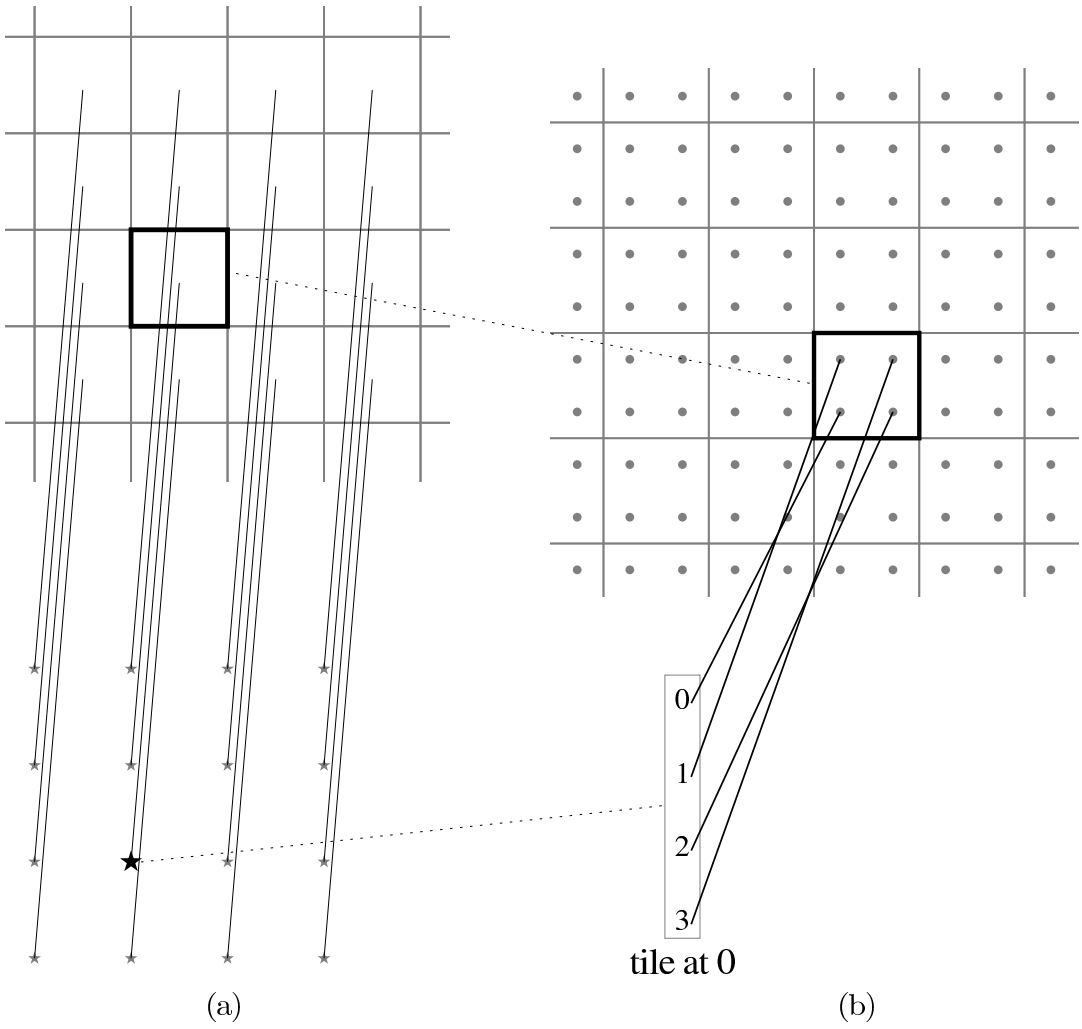

Figure 2: Two views of a tiling with a 2x2 square tile. The tile at 0 is highlighted throughout. (a) Top: square tiles in the iteration space. (Individual iterations not shown.) Bottom: $T = \mathbb{Z}^2 \times \{0, 1, 2, 3\}$ is shown with a star indicating each tile. The correspondence between the two spaces is roughly indicated. (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.

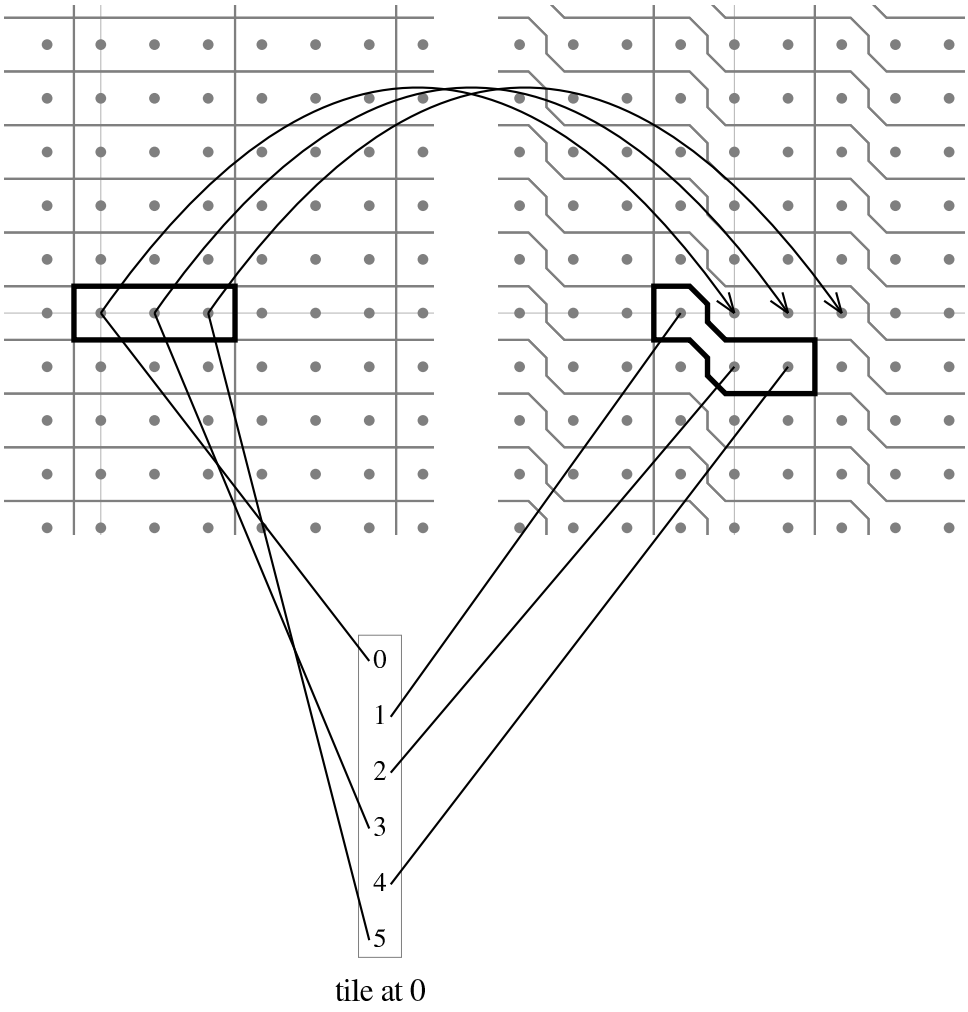

Figure 3: One way that the loops in Figure 1 can be optimized. The top shows a portion of the loops' iteration spaces with tiles outlined. (The first loop's iteration space is on the left.) The tile at 0 is highlighted. Below is tile space, $\mathbb{Z}^2 \times \{0, 1, 2, 3, 4, 5\}$, with only the tile 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 (section 4). 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. It can be replaced with a write to a 1-dimensional compiler-generated temporary array.

3.2 Inducing a Tiling

In this section we outline the method we use for *inducing* a tiling for $n+1$ consecutive `foreach` loops, L_0 through L_n. We assume the first n loops are already tiled and fused together, with a tile space $T = \mathbb{Z}^N \times \{0, \ldots, K-1\}$. (Tiling one loop is well understood. Repeated application of this method can tile and fuse together any number of loops.) If successful, we output an integer $K' > K$ and a correspondence between the $n+1$ loops' iteration spaces and $T' = \mathbb{Z}^N \times \{0, \ldots, K'-1\}$. Figure 3 shows sample output with $K = 3$ and $K' = 6$.

The basic idea is to select a point $\vec{z} \in \mathbb{Z}^N$ and consider the situation before and after each step of the tile at $\vec{z}$. Let $P(\vec{z}, k)$ be the subset of T that precedes step k of the tile at $\vec{z}$. For $S \subset T$, let *Ready*(S) be the nodes in L_n's iteration space that may legally execute if all elements of S have executed. Our guess for the tile at $\vec{z}$ in the induced tiling will include all K steps of the input's tile at $\vec{z}$, with nodes of L_n interleaved based on when they become legal: the input step $\langle \vec{z}, k \rangle$ will be followed by

$$\mathit{Ready}(P(\vec{z}, k) \cup \{\langle \vec{z}, k \rangle\}) \setminus \mathit{Ready}(P(\vec{z}, k)) \quad .$$

We compare *Ready*$(P(\vec{z}, 0))$ to *Ready*$(P(\vec{z} + e, 0))$, with e bound to each positive unit vector, to extend our guess to other nodes of L_n.

In practice this works fabulously on stencil codes, among others. However, the guess we generate is only guaranteed to be legal if:

- The correspondence between L_n's iteration space and the new $K' - K$ steps of T' is a bijection, and
- The reordering violates no ordering constraints (e.g., dependences).

If either condition fails then we do not fuse L_n with the previous loops. The conditions are tested by constructing a Presburger formula and testing its satisfiability by invoking the Omega Library [13].

4 Array Contraction

Of our storage-related optimizations, we'll describe array contraction, which is the most important. Arrays used as scratch space can sometimes be eliminated or dramatically reduced in size. The tiled version of the program in figure 1 is about 1.4 times faster 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.

The essential idea of our array contraction method is illustrated by Figure 3. To contract an array we require that for each write to it, the value is dead within some fixed distance in tile space—e.g., the write at $\langle \vec{x}, j \rangle$ is always dead by $\langle \vec{x} + \vec{v}, k \rangle$. The distance (e.g., $\langle \vec{v}, k - j \rangle$) determines how many values from that particular write can be live simultaneously, and therefore whether to hold those values in scalar(s), 1-dimensional temporary array(s), ..., or $(N-1)$-dimensional temporary array(s).

Strout et al. [26] analyze storage requirements for an array B when there is some v such that $B[p - v]$ must be dead by the time $B[p]$ is written, for all p. The identical analysis applies to storage requirements for 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 arises when some integer $t > 1$ divides all components of v. Figure 4 presents a variation on the previous example. A basic tiling of that program, Figure 5, illustrates the complication. 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 in this case. 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 shown in Figure 6.

In general, a particular textual write that is contracted writes to a scalar, a fixed-size circular buffer of scalars, a compiler-generated temporary array, or a

```
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;
```

Figure 4: A slightly different example amenable to array contraction—a variant of Figure 1.

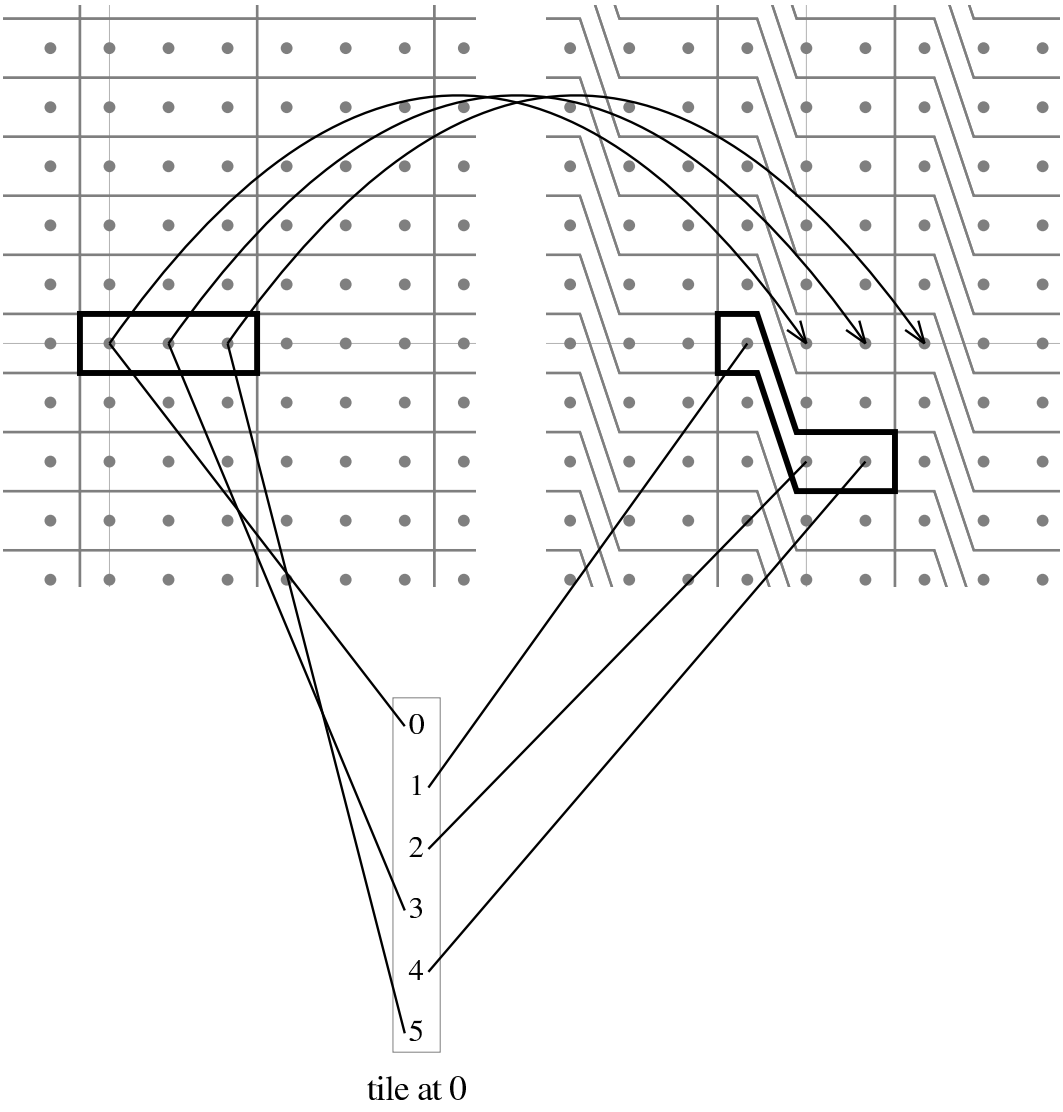

Figure 5: Illustration, in the same style as Figure 3, of Stoptifu's default tiling of the code from Figure 4.

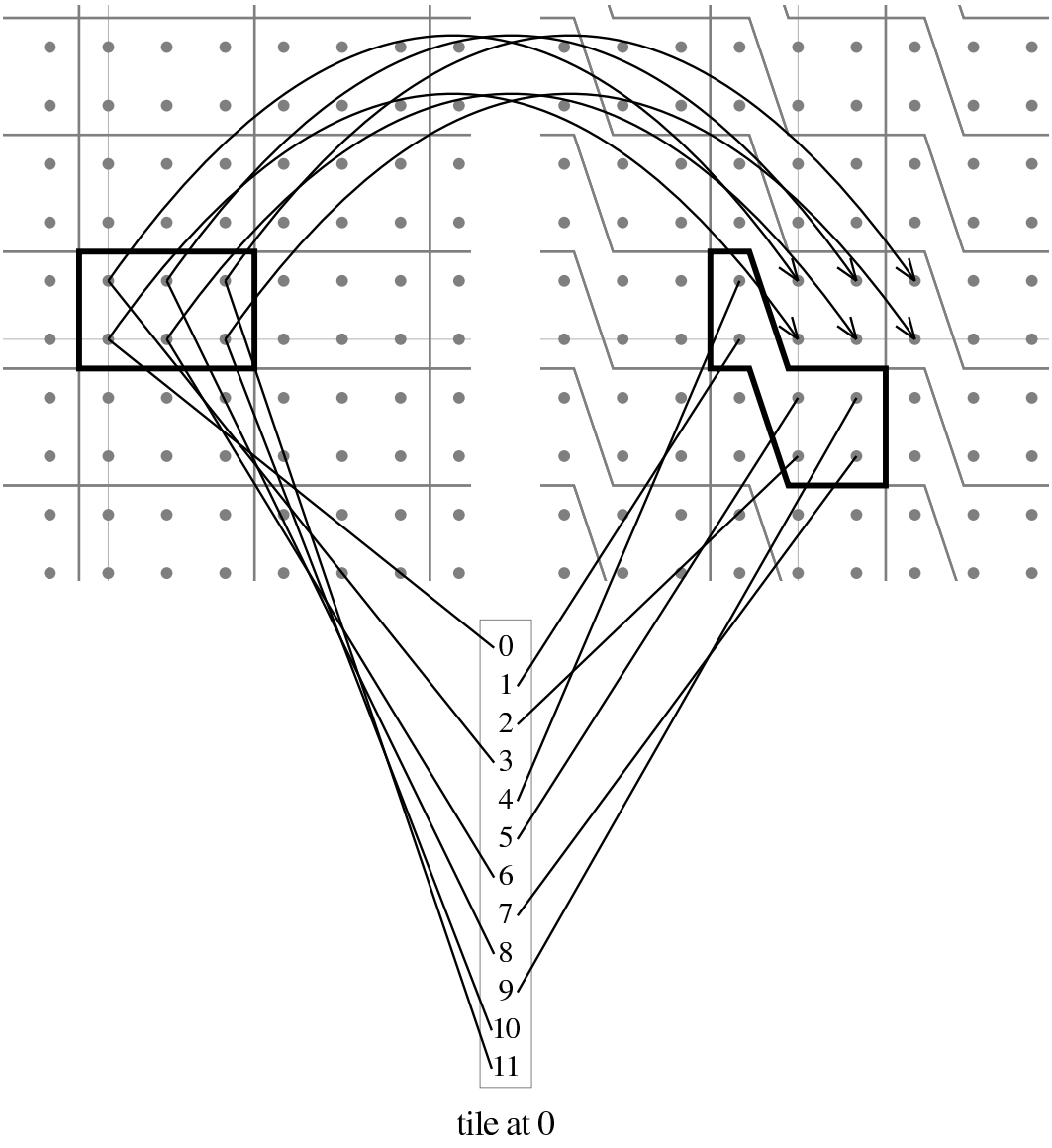

Figure 6: Using a bigger tile, temporary data consumed within a given tile stack always come from the immediately previous tile.

fixed-size circular buffer of compiler-generated temporary arrays. We have illustrated the first three cases by example. We apply the fourth case when one compiler-generated $(N - \xi)$-dimensional temporary array is insufficient but we find some $t > 1$ such that t of them is sufficient.

5 Results

We present results for three Titanium programs: a 1-dimensional cellular automaton simulation (`ca`), Gauss-Seidel relaxation (`rbrb9`), and multigrid (`mg`). Figure 7 presents the kernel of the first program. The second program is based on a C++/FORTRAN implementation of Anderson's Method of Local Corrections (MLC) by Phil Colella and Paul Hilfinger. Their implementation performs two red-black passes in a row in several places, and `rbrb9` is just that. (According to Sellappa and Chatterjee [24], 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. A V-cycle of 3D multigrid, based on Titanium code for Adaptive Mesh Refinement ([23]) by Luigi Semenzato, is our largest benchmark, `mg`.

Most testing was done on two PCs running GNU Linux. One is a 866MHz Intel Pentium III Coppermine, and one is a 1.4GHz AMD Athlon Thunderbird. The latter uses both gcc and icc (Intel's C compiler) while the former uses gcc only. We also present one result on a Sun Solaris UltraSPARC using gcc. (`tc` generates C code.) All problem sizes were chosen to fit comfortably in main memory but not in cache.

Without tiling and storage optimizations, but with all other optimizations, the baseline times for `ca` are 7.66s for the 167MHz UltraSPARC and 2.09s for the 866MHz Pentium III (table 1). A two hour search on the Pentium doubled the baseline performance. Improvement on the UltraSPARC is noticeable but less spectacular.

The benefits of decreasing memory traffic are more pronounced on the Pentium because its processor speed to memory speed ratio is higher. It appears that ratio will continue to increase in the future. The Sun also searches parameter space relatively slowly because equivalent compilations take longer.

Table 2 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 are 3.73 seconds (gcc) and 3.70 seconds (icc). With `tc`'s default parameters, that improves to 2.48 seconds (gcc) and 2.36 seconds (icc). The bottom line is a three-fold speedup.

There are numerous variations on multigrid (e.g., Briggs [2]), 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 [24] show a multigrid program that spends 80% or more of its running time doing GSRB. Using our results on `rbrb9`, 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 in the V-cycle. As expected, our method for inducing the fusion of loops 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 the other results, both array contraction and parameter search were necessary to do 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 (last line of table 3). The latter search used the best result from the 120 hour search as its initial position in parameter space.

```
/* 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 7: Pseudocode for `ca`.

	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 1: Results for `ca` on 866MHz Pentium III and on 167MHz UltraSPARC. The 0h line shows running times after compiling with `tc`'s default parameters. Longer searches yielded no further improvement. The problem size was five times larger on the Pentium.

	Array con-	Runtime after search		
Effort	traction	with gcc	with icc	MFLOPS
baseline	no	3.73	3.70	90.8
0h	yes	2.48	2.36	142
120h	no	1.49	1.44	233
120h	yes	1.42	1.34	251
120h	variable	1.30	1.22	275

Table 2: Results for `rbrb9` on 1.4GHz Athlon. For all three 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. Array contraction slows compilation, so *varying it* (last line) leads to a wider search through parameter space than *forcing it to be enabled* (second to last line).

Effort	runtime (s)	MFLOPS
baseline	2.72	100
0h	2.62	104
120h	2.17	125
...+8h	2.01	135

Table 3: Results for `mg` on 1.4GHz Athlon Thunderbird.

6 Related Work

6.1 Optimization by Hand

Studies of manually rewritten source programs have demonstrated optimizations that are quite similar to what we have done. Manual optimization of sequential multigrid and related codes is primarily focused on moving each datum from or to memory as infrequently as possible (Carter et al. [4]; Douglas et al. [6]; Sellappa and Chatterjee [24]). 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.

6.2 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 [30] and *Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning* by Lim et al. [20]. 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 [19]. Second, we do more array contraction. SUIF does array contraction only in the case where the array can be contracted to a scalar variable [20]. Both their own work and the aforementioned manual optimization work indicate that a more general form of array contraction 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.

A version of the D Compiler System aggressively fuses loops and reorders array storage (Ding and Kennedy [5]). Their technique is even more aggressive than ours in some respects. However, their system has two limitations that hinder its ability to transform multigrid codes: it can only fuse loops in a one-to-one ratio, and it does no array contraction.

Strout et al. introduced *universal occupancy vectors* as a way to express equivalence classes among an array's indices [26]. An array can be contracted if the compiler determines that at any time only one array element per equivalence class can be live. However, only one dimension of an array can be contracted, because two array indices are in the same equivalence class if and only if they differ by some multiple of the universal occupancy vector. Our new variant of array contraction can contract an array to any combination of lower-dimensional arrays and scalars. Even in the case where contracting one dimension of an array is *prima facie* optimal, our system can often contract *most* of the array to a fixed number of scalars. This is explained in section 4, and illustrated in Figure 3.

Song et al. describe a compiler that combines loop shifting, loop fusion, and array contraction [25]. Tiling is not done: a given loop's iterations are performed in the same order as specified in the FORTRAN source code. Furthermore, loop shifting is a blunt tool for exposing loop fusion opportunities. The advantage they gain by limiting the set of possible transformations is that selecting a transformation at compile time can be reduced to solving a network flow problem. Our compiler provides a richer set of transformations, but probably runs slower. We accept that trade-off because some programs we care about, including multigrid, require the richer set of transformations to run fast.

Thies et al. [27] 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.

Kodukula et al. describe a system that fuses loops aggressively and performs hierarchical tiling [17]. Their system works by grouping all loop iterations

that touch a given parallelogram of a given array, which can work well for dense linear algebra. However, it relies on the 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.

For simplicity, we have only investigated tilings that tile all space with a single tile. More complicated reordering techniques include Flynn Hummel et al.'s "fractiling," a method that uses tilings with many different tile sizes [9]; and Jin et al.'s "recursive prismatic time skewing," which thus far has only been demonstrated in a hand-optimization study [12]. We are considering incorporating such techniques into our system.

6.3 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 [1], ATLAS [29], Sparsity [10] [11], FFTW [7], and OCEANS [14] [15] [21] [16].

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. 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, OCEANS demonstrates 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.

Combining the brute force of parameter searching with modeling techniques is a sensible extension to our current search method (e.g., Vuduc et al. [28]). OCEANS also combines parameter space modeling with search.

7 Conclusion

The compiler optimizations we do yield the best available approximation to state-of-the-art hand-optimization techniques for multigrid. We apply them aggressively even in some cases where runtime tests are necessary to ensure safety. In combination with a parameter search via simulated annealing, simple codes yield great performance. We also view this work as a step towards "PHiPAC for any source program."

Acknowledgement. We'd like to thank the six anonymous reviewers for their helpful comments on the original submission.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] Chen Ding and Ken Kennedy. Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse. In *Proc. IPDPS 2001*, San Francisco, CA, 2001.

[6] 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.

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

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

[9] 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.

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

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

[12] G. Jin, J. Mellor-Crummey, and R. Fowler. Increasing Temporal Locality with Skewing and Recursive Blocking. In *Proc. SC2001*, Denver, Colorado, November 2001.

[13] 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.

[14] 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.

[15] 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.

[16] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Iterative Compilation. In *Embedded Processor Design Challenges—System Architecture, Modeling and Simulation (SAMOS)*, Springer Lecture Notes in Computer Science vol. 2268, pages 171–187, 2002.

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

[18] 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.

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

[20] 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.

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

[22] Geoffrey Pike. *Reordering and Storage Optimizations for Scientific Programs*. Ph.D. dissertation, University of California, Berkeley, January 2002.

[23] 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.

[24] 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.

[25] Y. Song, R. Xu, C. Wang, and Z. Li. Data Locality Enhancement by Memory Reduction. *15th ACM International Conference on Supercomputing*, June 2001.

[26] 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.

[27] 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.*

[28] R. Vuduc, J. Demmel, and J. Bilmes. Statistical Models for Automatic Performance Tuning. In *Proceedings of the 2001 International Conference on Computational Science (ICCS 2001)*, San Francisco, CA, May 2001.

[29] 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.

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