Computer Science
Technical Report

On Parameterized Tiled Loop Generation and Its Parallelization

DaeGon Kim and Sanjay V. Rajopadhye
`[kim|svr]@cs.colostate.edu`

January 22, 2010

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792   Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

On Parameterized Tiled Loop Generation and Its Parallelization

DaeGon Kim and Sanjay V. Rajopadhye
[kim|svr]@cs.colostate.edu

January 22, 2010

Abstract

Tiling is a loop transformation that decomposes computations into a set of smaller computation blocks. The transformation has proved to be useful for many high-level program optimizations, such as data locality optimization and exploiting coarse-grained parallelism, and crucial for architecture with limited resources, such as embedded systems, GPUs, and the Cell. Data locality and parallelism will continue to serve as major vehicles for achieving high performance on modern architectures. Parameterized tiling is tiling where the size of blocks is not fixed at compile time but remains a symbolic constant that can be selected/changed even at runtime. Parameterized tiled loops facilitate iterative and runtime optimizations, such as iterative compilation, auto-tuning and dynamic program adaption. Existing solutions to parameterized tiled loop generation are either restricted to perfectly nested loops or difficult to parallelize on distributed memory systems and even on shared memory systems when a program does not have synchronization free parallelism.

We present an approach for parameterized tiled loop generation for imperfectly nested loops. We empoly a direct extension of the tiled code generation technique for perfectly nested loops and three simple optimizations on the resulting parameterized tiled loops. The generation as well as the optimizations are achieved purely syntactic processing, hence loop generation time remains negligible. Our code generation technique provides comparable efficiency of generated code to the existing code generation techniques while our generated code remains simple and suitable for parallelization.

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

1 Introduction

Achieving high performance on modern architecture is becoming an demanding challenge. As the trend of multiple cores in a single chip continues, and the number of cores increases to fulfill the performance improvement implied by Moore's law, generating high performance code on modern architecture becomes even more complex. It requires exposing and exploiting parallelism, while enhancing data locality.

Many compute- and data-intensive applications spend most time on executing loops. An important class of such kernels is affine control loops. Tiling is a very useful loop transformation in high level optimization. It decomposes computations into sets of smaller blocks. Over decades of research and practical use in high performance implementations, tiling has proved to be effective for improving data locality and exposing coarse grained parallelism.

The blocks of computation, called tiles, are mapped to system resources: memory hierarchy, registers, functional units or cores/processors. The size of blocks makes data and/or computation fit into a certain resource, but also provides a fine control on the ratio between computation and communication. Hence, the tile size has a great impact on performance. Much research has addressed the problem of finding good and optimal tile sizes [13, 7, 10, 6].

In parameterized tiling, tile sizes are not fixed at compile time, and remain symbolic constants. They may be chosen and changed at runtime and even during a single run. Parameterization of tile sizes is quite useful for any empirical tuning/search in auto-tuners (ATLAS [24], PHiPAC [4] and SPIRAL [21]) and iterative compilers [13, 14], runtime feedback system and dynamic program adaption [18, 19].

In this paper we present a simple and efficient technique for generating parameterized tiled loops from imperfectly nested loops with hyper rectangular tile shape. Our approach consists of a direct extension of tiled loop generation technique for perfectly nested loops, and three simple optimizations, applied by a simple syntactic processing without

```
for (t = 0; t < Tmax; t++)
  for (i = t+1; i <= t+N-2; i++)
    x(i-t) = (x(i-t-1)+x(i-t)+x(i-t+1))/3;
```

```
for (t1 = shift_up(1-s1,s1); t1 < Tmax; t1+=s1)
  for (t2 = shift_up(t1-s1+3-s2,s2); t2 <= t1+N-2; t2+=s2)
    // point-loops
    for (t = max(0,t1); t < min(Tmax,t1+s1); t++)
      for (i = max(t+1,t2); i <= min(t+N-2,t2+s2-1); i++)
        x(i-t) = (x(i-t-1)+x(i-t)+x(i-t+1))/3;
```

Figure 1: Simple doubly nested loops for Gauss-Seidel style stencil computation; the loops have been transformed to make tiling legal

sophisticated analysis or heavy machinery like Fourier-Motzkin Elimination. Experimental results show that these optimizations have significant impact on the generated code, and provide comparable performance to existing solutions. We preserve tile loops to be perfectly nested, so it is suitable for subsequent parallelization on shared/distributed memory machines.

We also provide a technique for statically restructuring parameterized tiled loops for wave-front scheduling of tiles. Because of parameterization, transformations on tiles cannot be formulated as integer/rational matrices, and hence, the techniques in the polyhedral framework cannot be used to transform the parameterized tiled loops. This limited restructuring transformation has been an open problem. We develop our approach based on two key insights: one on the schedule, and the other on the loop generation. Restructuring parameterized loops to employ the wavefront scheduling is also achieved by syntactic processing without any heavy machinery.

2 Background

We first briefly recall parameterized tiled loop generation for perfectly nested loops [12]. The input is a perfectly loop nest of depth d for which tiling is legal, and it produces a loop nest of (at most) depth $2d$. The first d loops are called *tile-loops* and the remaining are *point-loops*. Tile-loops enumerate tiles, more precisely, tile origins—the lexicographically earliest points in tiles— and point-loops enumerate all iteration points within a tile.

Constructing point-loops is straightforward. Take the original loop nest, and replace a lower bound lb_i at depth i by $max(lb_i, t_i)$ where t_i is simply an iterator name of the corresponding loop in tile-loops. Similarly, replace an upper bound ub_i by $min(ub_i, t_i + s_i - 1)$. To generate tile-loops, we use D-tiling proposed by Kim et al. [12]. First, take an original loop nest, and replace iterator names by corresponding tile-loop iterator names and change a step size at level i by corresponding tile size parameter s_i. Now, if tile iterator t_k either (i) appears in a lower bound and its coefficient is positive, or (ii) appears in an upper bound and its coefficient is negative, replace t_k with $t_k - (s_k - 1)$. Finally, replace the resulting lower bound tlb_i by $\lceil (tlb_i - (s_i - 1))/s_i \rceil \times s_i$ at level i. The use of this approach is motivated by the fact that it does not require any polyhedral operations and all the tile origins are on a single lattice.

Consider simple doubly nested loops in Figure 1. We will derive parameterized tiled loops from these loops. First, we derive the outermost tile loop. Since there are no iterators in the bounds, the upper bound remains same, and the lower bound becomes $\lceil (1 - s_1)/s_1 \rceil \times s_1$. Note that $Tmax$ is not a iterator but a program size parameter. For the innermost tile-loop, we first replace $t + 1$ by $t_1 - s_1 + 2$, i.e., iterator name changes and t_1 is replaced by $t_1 - (s_1 - 1)$. Finally, a new lower bound is $\lceil (t_1 - s_1 + 3 - s_2)/s_2 \rceil \times s_2$. However, t in the upper bound will become just t_1 because it appears in an upper bound and its coefficient is positive. The final parameterized tiled code is shown in Figure 1. For brevity, we use two functions: *shift_up(a,b)* for $\lceil a/b \rceil \times b$ and *shift_down(a,b)* for $\lfloor a/b \rfloor \times b$.

3 Parameterized Tiled Loop Generation from Imperfectly Nested Loops

We first describe our input program specification and then provide algorithms for generating parameterized tiled loops for *embedded* imperfectly nested loops. Throughout this section, we will use a solver for lower triangular linear systems in Figure 2 as a running example.

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

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

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

Figure 2: A solver for lower triangular linear systems written in these styles: a simple imperfectly nested loop, a perfectly nested loop with affine-guards, and an *embedded* imperfectly nested loop

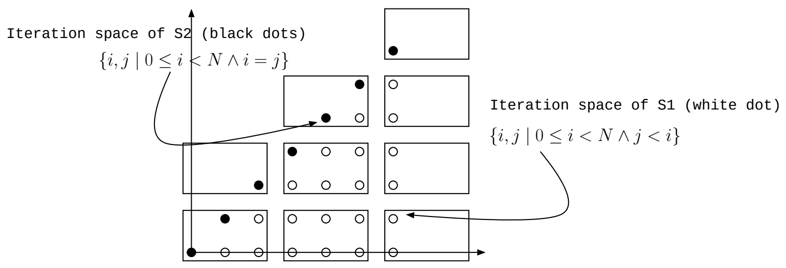

Iteration space of S2 (black dots)
$\{i, j \mid 0 \le i < N \wedge i = j\}$
Iteration space of S1 (white dot)
$\{i, j \mid 0 \le i < N \wedge j < i\}$

Figure 3: Iteration space of the solver for lower triangular linear systems

3.1 Input Programs

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

Such specifications can be obtained by embedding all the iteration spaces into a common space. Figure 3 shows the iteration spaces of our example. Notice that the two embedded loops in Figure 2 have the same iteration space. Embedded imperfectly nested loops can be obtained by generating loops from a union of polyhedra in a common space. In such loop nests, loops at the same depth have distinct loop counter ranges. For example, the ranges of the two loops in Figure 2, $\{j \mid 0 \le j < i\}$ and $\{j \mid j = i\}$, are disjoint. Also, the values taken by loop counters at the same depth is in the increasing order. In our context we expect to apply the tiling to the code automatically generated by a tool like CLooG [3] or Omega [20]. Hence this requirement, *embedded* imperfectly nested loops, is not too restrictive. The starting point for textual processing in PrimeTile [9] is exactly *embedded* imperfectly nested loops.

3.2 Tile-loop Generation

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

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

Figure 4: Algorithm to derive perfectly nested loops from embedded imperfectly nested loops

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

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

Figure 5: Two point-loops for the solver for lower triangular linear system: from embedded imperfectly nested loops and from perfectly nested with affine guards

For embedded input programs, we take the lower bound of the first loop and the upper bound of the last loop for each loop depth. Figure 4 shows a simple algorithm to derive perfectly nested loops from embedded imperfectly nested loops for the tile loop generation. Note that loop counters at the same depth are in increasing order. The iteration space of the derived perfectly nested loops contains the iteration space of each statement. Then we apply D-tiling to the perfectly nested loops.

Now, we show how to apply this algorithm to our running example. Consider the last loop nest in Figure 2. There is one loop at depth 1, and its lower and upper bounds are 0 and $N-1$. Since there is only one loop, taking the lower bound of the first loop and the upper bound of the last loop is the same as taking the loop itself. There are two loops at depth 2. By taking the lower bound of the first loop and the upper bound of the last loops, we obtain a loop whose lower and upper bounds are 0 and i respectively. This gives us doubly nested loops, to which we apply D-tiling and obtain tile-loops.

3.3 Point-loop Generation

The basic idea for generating point-loops remains the same. In the original loop program, replace every lower bound lb_i that occurs at level i by $max(lb_i, T_i)$ and every upper bound ub_i by $min(ub_i, T_i + S_i - 1)$. The loops visits the intersection of the original iteration points and points within a given tile. For example, consider the statement $S2$ in our running example. Its iteration points are $\{i, j \mid 0 \leq i < N \wedge j = i\}$. The corresponding loop is the last loop nest in Figure 2, ignoring the first j loop. So, we obtain loops where the outer loop counter i iterates from $max(T_i, 0)$ to $min(T_i + S_i - 1, N)$, and the inner loop counter j from $max(T_j, i)$ to $min(T_j + S_j - 1, i)$. Similarly, we can obtain the point-loops for the first statement. Since the common loop of these two statements are, by definition, identical, so are the common loops of the derived point-loops. Therefore, we can merge these two loops textually and finally obtain the loop for both statements shown in Figure 5 (above).

For input programs with guards, we can treat the whole sequence of guarded blocks as a single statement. In other words, the point-loops are derived from the perfectly nested loop rather than the iteration space of each block. By this process, we will obtain the point-loops in Figure 5 (below). One may convert the condition for each guard into loop bounds to derive the other kind of point-loops. For example, we can convert $j = i$ into $j \geq i$ and $j \leq i$. We can obtain the same point-loops as that in Figure 5 (above).

3.4 Parameterized Tiled Loop Generation

The final parameterized tiled loops is a simple insertion of the point-loops as the body of the tile-loops. The approach that we take in Section 3.2 and 3.3 is most direct extension of tiled loop generation for perfectly nested loops to imperfectly nested loops.

Tiled loop nests produced by this approach have been believed to be inefficient. However, there are a several reasons for choosing this as a starting point. First, with parameterized tile sizes and multiple iteration spaces, there is a risk of code size explosion. When tile sizes are fixed, generated code inefficiency is resolved by splitting tile-loops into many different regions based on the sets of statements being executed, either through a standard loop generation algorithm or a technique for eliminating guards. Where to split is statically known. For parametric tile sizes, this separation is much more complex than that for fixed tile sizes. In this situation it is possible for all the statements to be executed in a single tile for some values of tile sizes, while in extreme cases a tile may have only a statement. Furthermore, all the combinations of statements may be executed in a tile. Splitting all the cases might cause exponential growth in terms of the generation time and code size. Second, imperfectly nested tile loops are complicated to transform. Even for perfectly nested parameterized tile loops, a simple transformation like skewing for parallel execution of tiles has not been proposed. Third, all the computations are tiled unlike in the existing solution [9]. This makes the approach suitable to parallelization for machines where computing units have a limited resource, like Cell architecture.

3.5 Correctness

The correctness of this direct approach is straightforward. It follows directly from the correctness of the algorithm for perfectly nested loops presented by Kim et al. [12] The correctness of each optimization technique to be presented later comes from the properties of the sets being used, and those properties have been formally proved [23, 12].

3.6 Optimizations

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

3.6.1 Removing Statements from Point-Loops

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

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

3.6.2 Splitting Full Tiles

Like the first optimization, this optimization separates a set of tiles from the other tiles. An inset of an iteration space is a set containing all the full tile origins [23, 12]. A tile is full if all the points with a tile are valid iteration points. We may further specialize point-loops by identifying the inset of a statement. Note that iteration spaces of statements are disjoint each other. An tile that belongs to an inset of a statement does not contain any other statements. Therefore, the other statements need not to be checked.

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

Figure 6: Final structure of parameterized tiled loops from triangular solver; after optimization. For the general point-loops for both statements, see Figure 5

3.6.3 Selecting Iteration Space

The previous techniques provides a mechanism to distinguish particular tiles from others to specialize point-loops. However, there are $2^l - 1$ combinations of statements where l is the number of statements, and the optimizations themselves do not provide information for what set of statements leads to performance improvement. This optimization is not a technique, but a guideline for selecting a set of statements as well as splitting full tiles for statements. The statements that have *equalities* in their surrounding loops will be excluded, and the other statements are candidates to be selected. Tiles for theses statements are unlikely to be executed many times and be full tiles. For example, we will generate point-loops for only the first statement, but that for the second statement only.

3.7 Structure of Parameterized Tiled Loops

The structure of a generated tiled loop nest consists of (i) perfectly nested tile-loops, and (ii) a sequence of if/else if/else statements where the else clause has the most general point-loops from Section 3.3. The other conditional clauses are specialized point-loops guarded by appropriate conditions. The final structure of parameterized tiled loops for our running example is presented in Figure 6.

3.8 Experimental Results

We evaluated the efficiency of our approach on four common benchmarks. We compared our technique with that of PrimeTile on generated code efficiency.

3.8.1 Experimental Setup

We performed efficiency evaluation on four widely used benchmarks in Figure 7. We compiled all the code with gcc 4.4 and Intel icc 11.1 and ran all experiments for generated code efficiency evaluation on an Intel Core 2 Duo running 2.2 GHz with 2MB L2 Cache and 1GB memory. The execution time of non-tiled version for the benchmarks is also presented with the size of program parameters.

We generated six different versions of parameterized tiled loops. First, we generated using the direct extension of parameterized tiled loop generation without any optimizations. There are two kinds of point-loops: (i) perfectly nested loops with a sequence of guards and (ii) an embedded imperfectly loop nest. The first case is presented as *naive* and the second *naive-e*. Then we construct specialized point-loops for the statements that do not have equality

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

Figure 7: Benchmarks for evaluation of parameterized tiled loop generation technique for imperfectly nested loops. ∗There are four distinct statements in FDTD benchmark. Their iteration space can be divided into six disjoint regions. In our generation, the number of those regions is treated as the number of statements. The last column shows the execution time of original (i.e., non-tiled) program with gcc and icc.

constraints. When these optimizations are applied to *naive*, it becomes *opt*. Similarly, we have *opt-e* version. Then we further optimize this optimized code by splitting full tiles from the other tiles. They are represented by *split* and *split-e*. Finally, we have six versions: *naive*, *naive-e*, *opt*, *opt-e*, *split*, and *split-e*. In order to provide a comparison with a existing solution, we also generated parameterized tiled loops using PrimeTile [9].

3.8.2 Results

There are four charts that show the total execution time for each benchmark. The x-axis always represents tile sizes. For each tile size, there are 14 data points: 7 points using gcc and 7 points using icc. Among each seven data points, the first three pairs show the impact of using two different point-loops. For example, the first two data points shows the execution time of naive extension with two different point-loops. The first odd/even data points (like 1st, 3rd, and 5th points) show the impact of optimizations. In the tiled loops from PrimeTile, full tiles are split from the other computation together with optimized point-loops. So, *split-e* is the most similar to *prime* in characteristics although the computation other than that captured by full tiles in *prime* may not be tiled at all.

Figures 8, 9 and 10 show the total execution time for cache tiling on three matrix operation kernels. For very small tile sizes (2 to 4), *prime* performs better than our proposed method. For bigger tiles (64 to 512), our optimized code performs better. For intermediate tile sizes, the difference is almost negligible. This is the general trend for the first three kernels which have few statements.

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

Figure 11 shows the total execution time of FDTD kernel for various tile sizes. Unlike the other kernels, non-cubic tile sizes are used. The data from this kernel is a quite different from the others. The main difference in program characteristics is that there are many statements. Although it has four statements, four iteration spaces must be decomposed into six disjoint regions for generating parameterized tiled loops. For the comparison on tiled loop generation, it has six statements. Our optimized code outperforms that from PrimeTile even for small tile sizes.

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

4 Parallel Execution of Parameterized Tiles

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

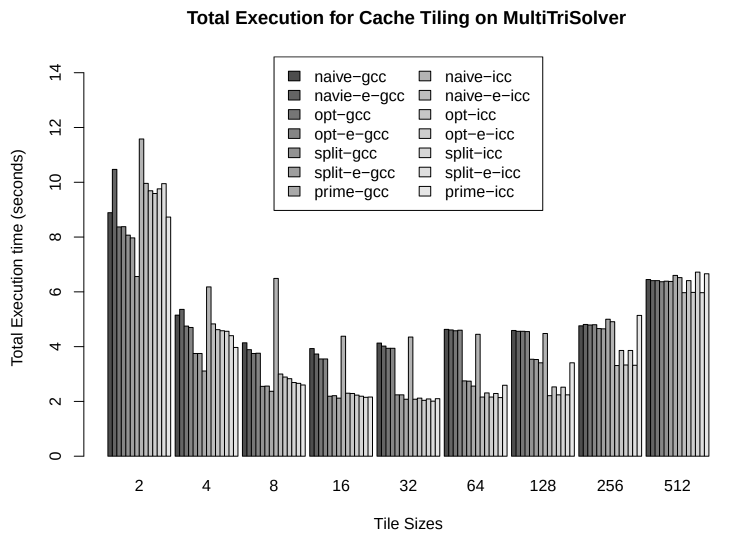


Figure 8: Total execution time of cache tiling on MultiTriSolver with cubic tile size

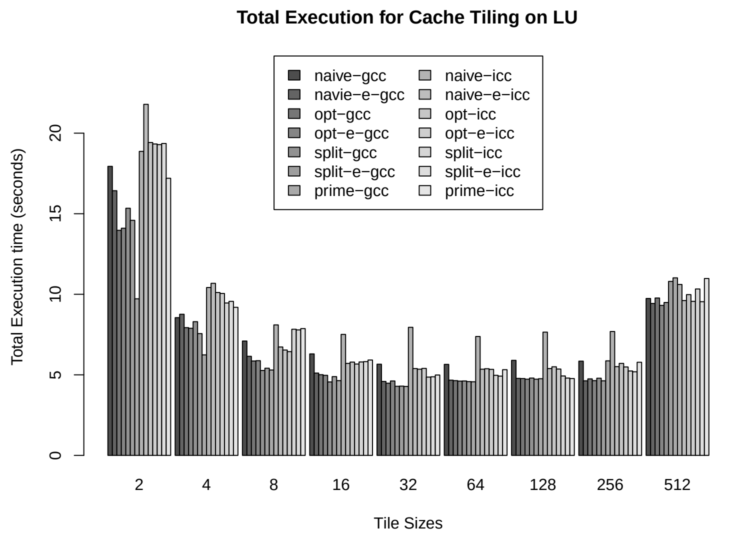


Figure 9: Total execution time of cache tiling on LU with cubic tile size

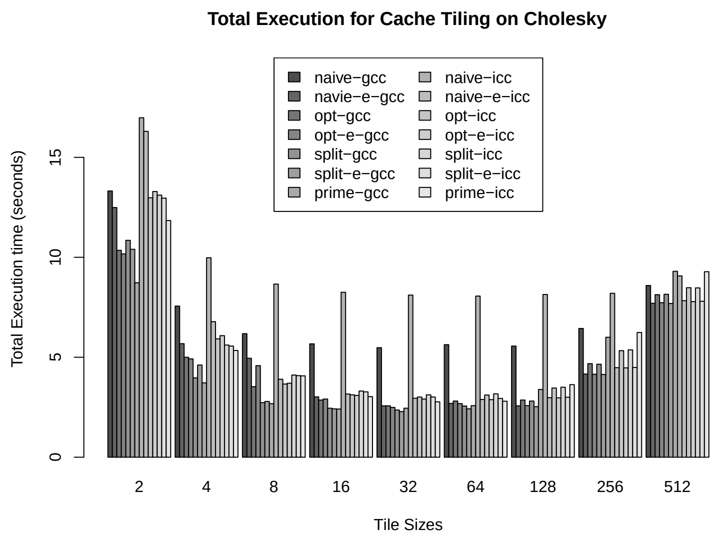


Figure 10: Total execution time of cache tiling on Cholesky with cubic tile size

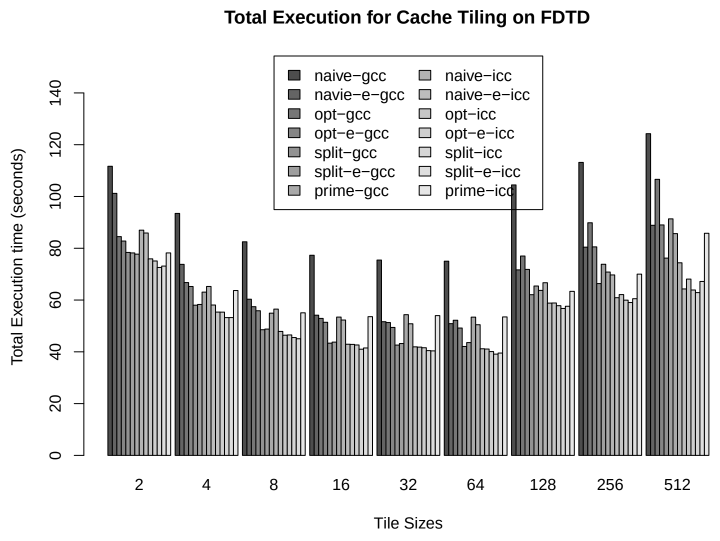


Figure 11: Total execution time of cache tiling on FDTD with tile size along time dimension fixed as 4

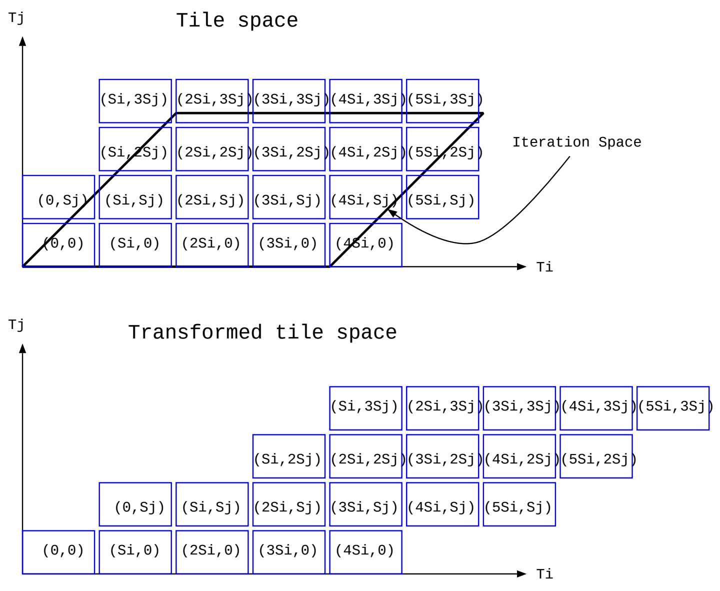


Figure 12: Tile space and its transformed space; each tile is denoted by its origin coordinate; the tiles in the transformed space is denoted by the tile origin coordinates in the original space

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

Consider the iteration space $\mathcal{I}$ in Figure 12:

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

Its tile space can be formulated as the canonical projection onto the first two dimensions of $\{t_i, t_j, i, j \mid s_i \times t_i \leq i \leq s_i \times t_i + s_i - 1 \land s_j \times t_j \leq i \leq s_j \times t_j + s_j - 1 \land (i, j) \in \mathcal{I}\}$. So, when tile sizes, s_i and s_j, are fixed (i.e., constant integers), the tile space remains polyhedra. Therefore, a skewing transformation such as $(t_i, t_j \rightarrow t_i + t_j, t_j)$ can be applied. The scanning loop for this skewed space allows the tiles to be executed in a wavefront fashion. Figure 12 also conceptually shows how these loops enumerate the tiles. However, this approach works only when tile sizes are fixed and hence the tile space is formulated as polyhedra.

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

Now, consider a tile space where tile sizes are parameterized and the space is viewed as an intersection of polyhedra and a parameterized integer lattice. Figure 12 shows an example of parameterized tiling and skewing transformation. Note that figures are not a precise representation for parameterized tiling. The figure gives an impression that the width of tiles is larger than their height even though such assumption is not made. In this case, transformation for skewing tiles is formulated as $(t_i, t_j \rightarrow t_i + (s_i/s_j) \times t_j, t_j)$. To the best of our knowledge, polyhedral framework does not support this transformation. Hence, neither transformation nor code generation technique in the polyhedral model can be applied.

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

4.1 Basic Ideas Illustrated

One of two main ideas behind our approach is that we know all the tiles that need to be visited for a given time stamp. Consider that we find $(S_i, 2S_j)$ at the original tiled space in Figure 12. Without transforming the tiled loops, we know that $(2S_i, S_j)$ and $(3S_i, 0)$ tiles need to be executed with $(S_i, 2S_j)$. All valid tiles that can be written as $(S_i, 2S_j) + k \times (S_i, -S_j)$ for some k need to be visited with $(S_i, 2S_j)$. In other words, $(S_i, 2S_j)$ will be executed at $time = 3$ computing from $(S_i)/S_i + (2S_j)/S_j$. Any tiles (t_i, t_j) where $t_i/S_i + t_k/S_j = 3$ will be executed with $(S_i, 2S_j)$ at $time = 3$. The wave-front schedule of parameterized tiled loops for this example can be expressed as $t_i/s_i + t_j/s_j$.

In general, the wave-front schedule is

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

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

Our approach uses the original scanning order, and computes the last tile index using the schedule. For example, we use the outer loop of the original tile-loops, and compute t_j using $t_j = (time - t_i/S_i) * S_j$ for our running example. The bounds of t_j are used for checking the validity of a tile. Now, the only necessary information is its time stamp. We use the time stamp of the first tile as starting time and that of the last tile as ending time stamp. We iterate over time and the original tile-loops with appropriate processing will visit all the tiles that need to be executed for a given time.

4.2 Generation Algorithm

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

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

Similarly, we compute the last tile

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

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

Now, we construct a loop that scans all the time stamps between the first and last time stamps. We denote this loop as $(time, start, end, 1)$. We directly use input loop nest except the innermost loop. The innermost loop will be replaced by the following assignment statement

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

and a guard whose condition is

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

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

The algorithm is presented in Figure 13. One may want to change the scheduling equation with different one.

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

```
Input: $AST$ - perfectly nested loops up to depth d, a loop $L_k$ at depth k consists of iterator $t_k$, lower bound $lb_k$, upper
    bound $ub_k$ and step size $s_k$
 1: for k= 1 to d do
 2:     $t_k \leftarrow lb_k(t_1, \ldots, t_{k-1})$
 3: end for
 4: $start \leftarrow t_1/s_1 + \cdots + t_d/s_d$
 5: for k= 1 to d do
 6:     $t_k \leftarrow ub_k(t_1, \ldots, t_{k-1})$
 7: end for
 8: $end \leftarrow t_1/s_1 + \cdots + t_d/s_d$
 9: $Tloops \leftarrow loop(time, start, end, 1)$
10: for k=1 to d-1 do
11:     add $L_k$ to $Tloops$ as its innermost loop's body
12: end for
13: create a statement S as $t_d$ = $(time - (t_1/s_1 + \cdots + t_{d-1}/s_{d-1})) \times s_d$
14: create a guard G with $(lb_d \leq t_d) \wedge (t_d \leq ub_d)$ as its condition and the body of $L_d$ as its body
15: add S and G as the body of $Tloops$'s innermost loop
```

Figure 13: Generation algorithm for wave-front scheduling of parameterized tiled loops

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

Figure 14: A simplified parameterized tiled loop for the example in Figure 12; the actual loop body is replaced by a macro for brevity; the loop body is irrelevant to the generation algorithm

4.3 Algorithm Walk-through

The simplified tiled loops for the example in Figure 12 is given in Figure 14. Now, we apply our algorithm to these loops. Note that we are interested in only tile loops, i.e., up to depth 2 for this loop nest.

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

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

Note that the upper bounds may not be a multiple of tile size for a given depth. Similarly, we create a statement for assigning end by the time stamp of this last tile.

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

The final code produced by our algorithm is given in Figure 15.

4.4 Experimental Results

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

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

```
$t_i = 0; t_j = max(0, shift\_up(t_i - M - s_j + 1, s_j));$
$start = t_i/s_i + t_j/s_j;$
$t_i = shift\_down(M + N, s_i); t_j = shift\_down(min(t_i + s_i - 1, N), s_j);$
$end = t_i/s_i + t_j/s_j;$
for ($time = start$ ; $time \le end$ ; $time$++)
  for ($t_i = 0$; $t_i \le M + N$; $t_i += s_i$)
    $t_j = (time - t_i/s_i) \times s_j$
    if ($max(0, shift\_up(t_i - M - s_j + 1, s_j)) \le t_j \wedge t_j \le min(t_i + s_i - 1, N)$)
      for ($i = max(0, t_i)$; $i \le min(M + N, t_i + s_i - 1)$; $i$++)
        for ($j = max(0, i - M, t_j)$; $j \le min(i, N, t_j + s_j - 1)$; $j$++)
          S1(i,j);
```

Figure 15: Transformed loop nest for wave-front scheduling from the parameterized tiled loops in Figure 14

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

Figure 16: Four benchmarks for the evaluation of the restructured parameterized tiled loop nest for parallel execution on shared memory systems

tiled programs with constant tile sizes.

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

Figure 17 shows the loop overhead of three kernels between before and after loop restructuring for wave-front scheduling. The overhead due to transformations is not significant.

Figure 18 shows the total execution time of MultiTriSolver and Gauss-Seidel stencil computation. MultiTriSolver consists of independent triangular system solvers, so it has synchronization-free parallelism. So this kernel does not require the wave front scheduling, and therefore the original parameterized tiled loops are used without restructuring. We generates three reference code using Pluto [5], CLooG, Omega. We generate code from Pluto with option -tile and -parallel and tile sizes are fixed to $16 \times 16 \times 16$. We directly use the generated code without any modification. Note that Pluto does not always choose wave-front scheduling for parallelization. In order to obtain code with wave-front scheduling, we generate code using CLooG and Omega with same tile sizes. We use $(t_1, t_2, t_3 \to t_1 + t_2 + t_3, t_1, t_2)$ as a transformation on tiles, and this transformation provides the most similar code to our restructured parameterized tiled code. We generate two versions of code using our technique: one without full-tile splitting (ParWave) and the other with splitting (Parwave(Split)). The restructuring process is same, but the starting parameterized tiled loops are different. Our approach provide a comparable performance to the fixed size tiling with skewing transformation, but the tile sizes remains as runtime parameters.

Figure 19 shows the total execution time of LU and Cholesky. We generate code from Omega using option 2 for LU. Omega does not generate a correct code with option 3 from our specification of LU.

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

5 Related Work

The problem of generating tiled loops has been closely related to that of generating scanning loops from polyhedra. Ancourt and Irigoin [2] proposed a technique for scanning a single polyhedron, based on Fourier-Motzkin elimination over inequality constraints. Le Verge et al. [16, 15] proposed an algorithm that exploits the dual representation of

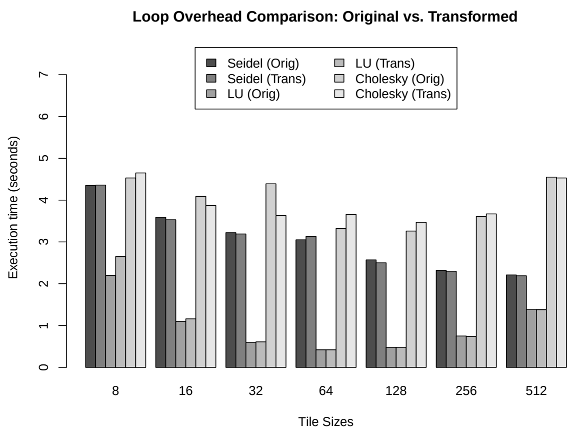
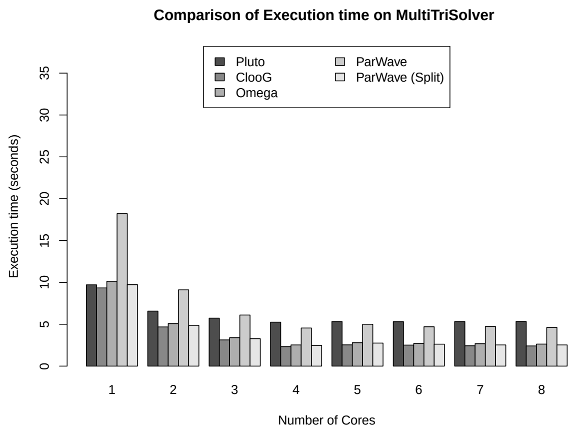


Figure 17: Loop overhead comparison between before/after loop restructuring



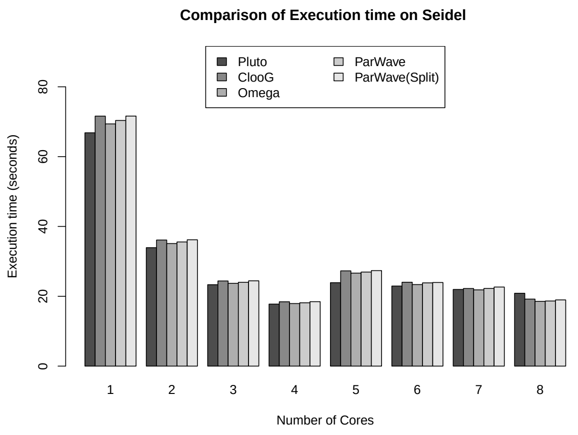
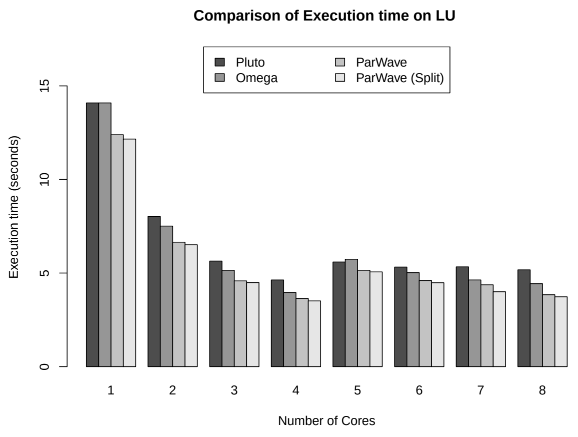


Figure 18: Total execution time of Multiple triangular solver and Gauss-Seidel stencil computation. All the loops in fixed tile size code are tiled with $16 \times 16 \times 16$. ParWave is parameterized tiled code without splitting full tiles, and ParWave(Split) with splitting full tiles



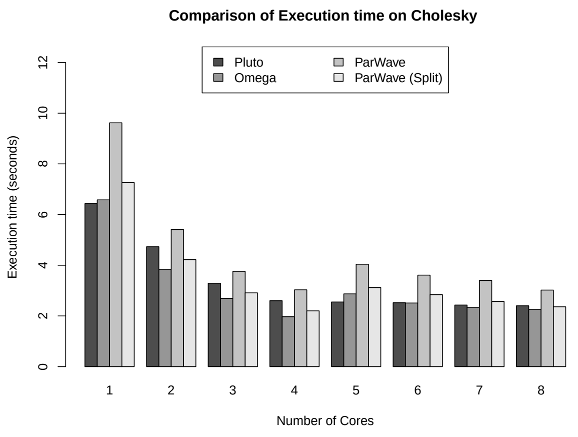


Figure 19: Total execution time of Cholesky and LU. All the loops in fixed tile size code are tiled with $48 \times 48 \times 48$. ParWave is parameterized tiled code without splitting full tiles, and ParWave(Split) with splitting full tiles

polyhedra with vertices and rays in addition to constraints. The general code generation problem for affine control loops requires scanning unions of polyhedra. Kelly et al. [11] solved this by extending the Ancourt-Irigoin technique, and together with a number of sophisticated optimizations, developed the widely distributed Omega library [20]. The SUIF [25] tool includes a similar algorithm. Quillere et al. proposed a dual representation algorithm [22] for scanning the union of polyhedra, and this algorithm is implemented in the CLooG code generator [3] and its derivative Wloog is used in the WRaP-IT project. Thanks to Irigoin and Triolet's proof that tiled iteration space is a polyhedron if tile sizes are integer constants, these techniques for generating loops that scan polyhedra can be directly used when tile sizes are fixed. Goumas et al. [8] proposed a method where the tiled loop generation problem is decomposed into two sub-problems: generation of tile-loops and generation of point-loops, thus obtaining significant reduction of the worst case complexity. Ahmed et al. [1] proposed a technique for general imperfectly nested loops based on embedding into a common Cartesian product space. Lim et al. [17] proposed a technique for independent threads that have no dependence between them, and also use their algorithm for finding largest outermost fully permutable loops for embedding. Bondhugula et al. [5] proposed a approach based on tiling hyperplanes that are linearly independent relaxed pipeline scheduling hyperplanes and the set of tiling hyperplanes provides an affine transformation.

When tile sizes are not constants, but symbolic parameters, the above techniques are not directly applicable. Recently, a several researchers have addressed to the problem of generating parameterized tiled loops for arbitrary polyhedral-shaped iteration space. Renganarayana et al. [23] proposed a technique for generating parameterized tiled loops from perfectly nested loops using an outset, a set containing all the necessary tile origins. Their outset remains a polyhedron, but parameterized by tile size parameters as well as program size parameters. Because the set was formulated as a polyhedron, they could use the techniques and tools for generating loops from polyhedra. They produced the final tiled loops by generating loops from their outset using CLooG and the syntactic post-processing of the loops. Hartono et al. [9] presented a technique for generating parameterized tiled loops from imperfectly nested loops. They do not rely on complex polyhedral operation but use syntactic manipulation after they obtained a kind of embedded original (non-tiled) loop nest. However, their resulting code is complex and difficult to parallelize except computations that have already synchronization-free parallelism, because their tiling model is slightly different from the conventional tiling where all tile origins lay on a single (either fixed constant or parameterized) lattice. They focus on extracting full tiles, and some part of iteration space may not be tiled. They achieved significant reduction of loop overhead even for very small tile sizes. Kim et al. [12] proposed an algorithm for generating parameterized tiled loops from perfectly nested loops with purely syntactic processing, and their generated code is similar to that of Renganarayana et al. [23].

6 Conclusions and Future Work

We present techniques for generating parameterized tiled loop for imperfectly nested loops and statically restructuring the loops for wave-front scheduling. The experimental results show (i) our technique for tiled loop generation provides a comparable code efficiency to the existing solutions, and (ii) statically restructured parameterized tiled loops are as good as and even better than the fixed size tiling combined with skewing transformation.

In the future we plan to extend our techniques to multi-level tiling for parallelism and data locality. Another direction of our interests is developing auto-tuning framework for optimal tile size selection.

References

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing '00*, page 31, Washington, DC, USA, 2000. IEEE Computer Society.

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

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

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

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

[6] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290. ACM Press, 1995.

[7] Basilio B. Fraguela, M. G. Carmueja, and Diego Andrade. Optimal tile size selection guided by analytical models. In *PARCO*, pages 565–572, 2005.

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

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

[10] Chung hsing Hsu and Ulrich Kremer. A quantitative analysis of tile size selection algorithms. *J. Supercomput.*, 27(3):279–294, 2004.

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

[12] DaeGon Kim and Sanjay Rajopadhye. Efficient tiled loop generation: D-tiling. In *LCPC 2009: 12th International Workshop on Languages and Compilers for Parallel Computing*. Springer Verlag, 2009.

[13] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

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

[15] H. Le Verge, V. Van Dongen, and D. Wilde. La synthèse de nids de boucles avec la bibliothèque polyédrique. In *RenPar'6*, Lyon, France, Juin 1994. English version "Loop Nest Synthesis Using the Polyhedral Library"in IRISA TR 830, May 1994.

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

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

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

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

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

[21] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, February 2005.

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

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

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

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