Parameterized Tiling for Imperfectly Nested Loops

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

February 27, 2009

Technical Report CS-09-101

Parameterized Tiling for Imperfectly Nested Loops

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

February 27, 2009

Abstract

Parameterized tiled loops—where tile sizes are run-time parameters rather than constants fixed at code generation time—are quite useful for several cases. Such cases include empirical search for optimal tile sizes in iterative compilers and highly optimized library generators like ATLAS, and parallelizing compilers that enable the number of processors to be a run-time parameter. However, automatically generating such code is a luxury, only available for perfectly nested loops (single iteration space) where all the statements are surrounded by the same set of loops. We present a framework for generating parameterized tiled loops from arbitrarily nested affine control loops. In this framework, tiling is applied dimension-by-dimension, disproving the decades-long belief that the tiled loop generation problem has exponential complexity. For perfectly nested loops, our algorithm has $O(m \times (d+p))$ where m is the number of bounds, d the maximum nesting depth of loops, and p the number of program parameters. Based on this, we are able to avoid the exponential complexity even for imperfectly nest loops. In the currently accepted view, imperfectly nested loops are handled by first embedding all iteration spaces into a common higher dimensional space and then applying the techniques for perfectly nested loops. This requires two expensive steps: the embedding itself and index set splitting to obtain efficient code. We take a completely different view by exploiting the textual structure of the original loop nest and dimension-by-dimension tiling. Also, we formulate a legality condition for imperfectly nested loop tiling. Our code generation efficiency is better than fixed size tiling with embedding. The efficiency of generated code is comparable to those techniques. Our technique does not rely on the expensive polyhedral operations and works on loops. Its scalability and simplicity make our techniques attractive for production compilers.

1 Introduction

Partitioning computation is a program transformation that is becoming more important with the advent of multi-core chips and the growing gap between the memory/network performance and computational power. There are many reasons to believe that this gap will continue to widen. The number of cores on a chip is expected to increase at an exponential rate leading to a few hundred cores in the near future while continuing to fulfill promise on machine performance implicitly made in Moore's law. Large computation and high performance cannot be achieved without partitioning computation for exploiting parallelism and efficient use of memory hierarchy. Due to the lack of tools, this program restructuring must often be done by hand, but is not easy even for specialized programmers on a particular architecture. Moreover, a highly tuned programs for a particular architecture are not portable. In the next generation, tools to aid programmers to write high performance implementations will be a key component in software development.

In many compute- and data-intensive programs a significant portion of the execution time is consumed by loops operating on arrays. Optimizing these loops involves improving data locality and parallelizing the computation performed by loops. Tiling [15, 19, 30, 34] has been used for improving data locality and exposing/exploiting the parallelism. Its effectiveness has been proven through almost three decades of research and high performance implementations of linear algebra and stencil computations, such as ATLAS and PHiPAC. With the advent of the multi/many core era, tiling becomes even more important.

The transformation partitions a program into a set of smaller pieces (called *tiles*) so that each of those pieces either fit to a resource such as registers, cache or physical memory, or reduce communication cost between processors and memory hierarchy. Various aspects of tiling have been extensively studied: how to pre-process loops to make tiling legal and enhance data locality [19, 34] (e.g. loop skewing, loop permutation and other unimodular transformations); tile shape selection [8, 14, 28] and tile size selection for memory hierarchy as well as parallelism [5, 10]; the generation of tiled loop [12, 15, 34]. Tiled code generation is often an ignored step in the process. This is partly because techniques for loop generation from a union of polyhedra is well studied [17, 25, 33, 27, 6] and tiled iteration space is a polyhedron [15]. Recently, a decomposition approach, where tile-loops and point-loops are generated separately, was proposed to reduce the complexity of the generation process [12].

Parameterized tiled loops are tiled loop nests where tile sizes are not constants but given as run-time parameters. There are many situations where such tiled loops are preferable: any empirical search for optimal tile sizes such as iterative compilers and auto-tuners [26, 32]; run-time tile size adaptation for varying resource due to resource sharing [23, 24]; and parallelizing compilers [3] that allow the number of processors to be a run-time parameter. Recently, Lakshminarayanan et al. [29] presented a theory and tools for parameterized tiled loop generation.

Unfortunately, most of the above techniques are restricted to perfectly nested loops. No technique for parameterized tiled loop generation from imperfectly nest loops has been proposed, to date. Additionally, all the previous solutions for fixed size tiling suffer from an expensive generation process and complex generated code. When tiling is used for parallelism, especially on distributed memory machines, the resulting code has an considerable impact on subsequent processes like generating statements for communications.

In this paper, we present an algorithm and tool for generating parameterized tiled loops for arbitrary loop nests. It is novel in the sense that it does not rely on embedding—the conventional approach to extend techniques for perfectly nested loop tiling to those for imperfectly nested loops. To the best of our knowledge, all previous approaches for imperfectly nested loops are based on embedding. The contributions of this paper are:

- a formulation for generating tiled loops dimension by dimension that enables us to generate parameterized/fixed tiled loops without exponential complexity; this is directly applicable to perfectly nested loops as well.
- a legality condition for tiling imperfectly nested loops that does not rely on making them perfectly nested;
- An algorithm for generating parameterized tiled loops from arbitrary nested affine control loops without complex polyhedral operations such as projections and index set splitting.
- In addition, since the algorithm works on only a loop AST, it is ideal for production compilers.
- An evaluation of the efficiency of both the generation itself and the generated code on benchmarks such as stencil and matrix factorization code. It shows that our algorithm is efficient and the resulting code is comparable to those of fixed size tiled loops with embedding functions given at compile time.

- Our parameterized tiled loop generators will be made available as an open source toolkit.

The key insight to the dimension-by-dimension tiling is that the tile space for the projection of the iteration space is large enough to contain the projection of all the non-empty tile origins and the projection of the iteration space is already present in the loop bounds. So, all the bounds of the tile loops can be directly obtained from the bounds of the original loops without expensive procedures like Fourier-Motzkin elimination.

The main intuition behind our parameterized tiled loop generation from imperfectly nested loops is that (*i*) a legality condition for tiling can be formulated without embedding all the iteration spaces into a common space—equivalently making the loops into perfectly nested loops and (*ii*) the structure of tile loops can be derived from that of the original loops.

The rest of the paper is organized in the following way. Section 2 provides a background and notation for the paper. Section 3 explains a brief but powerful formulation for the tile space leading to an efficient tiled loop generation algorithm. Section 4 gives our main intuition on how to generate parameterized tiled loops using an example, provides the generation algorithm, and formulates a legality condition for our tiling approach. Section 5 provides an experimental result on comparison with fixed embedding tiled loop generation method.In Section 6, we conclude our discussion.

2 Background: Tiling Perfectly Nested Loops

The tiling transformation takes a d-depth (perfectly) nested loop and produces a loop nest of depth (at most) $2d$. The main idea is the decomposition of the iteration space into a collection of smaller sets, called tiles. Each tile becomes an atomic block of computation, in other words, the order of computation is changed.

When loop bounds are affine functions of program parameters and outer loop indices, the iteration space can be expressed as a polyhedron. Consider the loop in Figure 1. Its iteration space can be written as

$$\{i, j \mid i \leq N; 1 \leq j \leq i\}$$

Note that there are only three inequalities that define the iteration space because $1 \leq i$ is redundant. A geometric representation of this iteration space is shown in Figure 2.

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

Figure 1: Triangular iteration space: the body of the loop is represented with the macro S1 for brevity

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

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

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

Whenever there is no ambiguity we use z in the place of $\vec{z}$. We denote the k-th component of z as z_k. We also denote tile size vector as s.

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

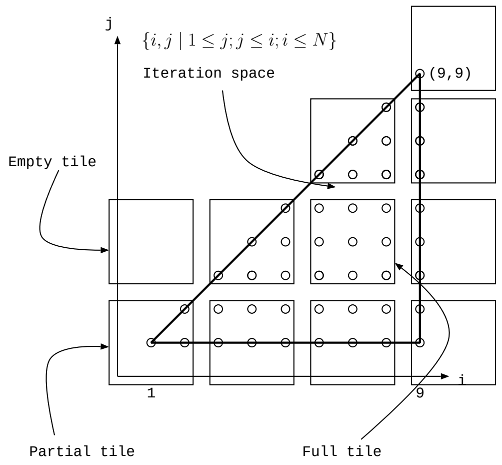

Figure 2: An iteration space when $N = 9$ and its 3×3 tiling

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

The essence of tiled loop generation is constructing two sets of loops—*tile-loops* that enumerate all the tiles (all the tile origins) and *point-loops* that enumerate all the points in a tile. When tile-loops and point-loops are separately generated, point-loop generation is trivial, just adding appropriate tile-bounds at the original loop bounds. Generation of tile-loops is not trivial because they must enumerate not only all the full tiles but also all partial tiles. Lakshminarayanan et al. [29] defined a set an *outset* if it contains all the partial/full tile origins. An outset is precise if it does not contain any empty tile origins.

They also proposed an outset that is not precise but a polyhedron. Their construction method is based on shifting constraints. Formally, their outset is written as

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

where s' is $s - 1$ and

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

The term $-Q^+ s'$ can be interpreted as a shift of a hyperplane. The outset of the iteration space for the example in Figure 1 is shown in Figure 3. When a hyperplane is a lower bound of the k-th dimension, it will be shifted by s_k. So, $1 \le j$ becomes $1 - (s_j - 1) \le t_j$.

Then, an existing tool such as CLOOG is used for generating all the points in the outset, and then the generated loops are further processed—lower bounds are adjusted and strides are set to tile sizes—to visit only tile origins. Figure 4 shows the generated code using their open source tool *HiTLOG* [18]. The macro $up(t, s)$ in lower bound gives an integer p such that $p = \lceil t/s \rceil \times s$. It adjusts the lower bounds so that the loops visit tile origins correctly.

Later, we will use their shifting operation to obtain loop bounds and compare this code with our generated code when input programs are perfectly nested.

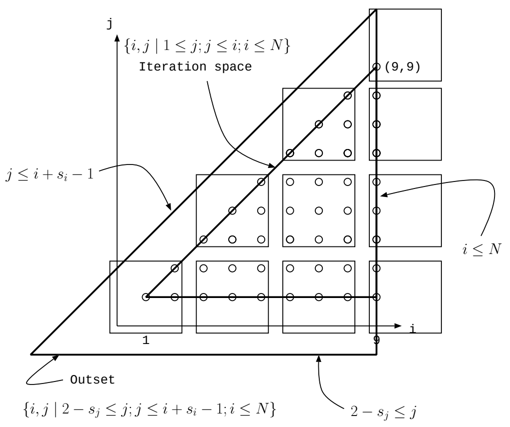
j
$\{i,j \mid 1 \le j; j \le i; i \le N\}$
Iteration space
(9,9)
$j \le i + s_i - 1$
$i \le N$
i
1
9
Outset
$\{i,j \mid 2 - s_j \le j; j \le i + s_i - 1; i \le N\}$
$2 - s_j \le j$

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

```
// tile-loops
for (ti = up(-si-sj+3,si); ti <= N; ti+=si)
  for (tj = up(-sj+2,sj); tj <= ti+si-1; tj+=sj)
    // point-loops
    for (i = MAX(1,ti); i <= MIN(N,ti+si-1); i++)
      for (j = MAX(1,tj); i <= MIN(i,tj+sj-1); j++)
         S1(i,j);
```

Figure 4: Tiled loops from the example in Figure 1 from HiTLOG

3 Tiling Dimension-by-Dimension for Perfectly Nested Loops

This section shows how to apply tiling dimension by dimension, i.e., from the outermost loop to innermost loop. To obtain the tile loops from the original loop nest, we apply three simple rules to change the loop bounds:

- For each lower bound at depth k, we subtract $s_k - 1$ from it.
- If an outer iterator i appears in an upper bound expression and its coefficient c_i is positive, we add $c_i \times (s_i - 1)$ to the upper bound
- If an outer iterator i appears in a lower bound and its coefficient c_i is negative, we subtract $c_i \times (s_i - 1)$ from the lower bound

If a bound has multiple affine expressions, we treat each one individually with the above rules. The complexity of generating tile-loops with post-processing is $O(m \times (d + p))$ when there are m affine loop bounds, d is the depth of the loops and p is the number of program parameters. We call this approach to generating tile-loops as dimension-by-dimension *D-tiling*.

3.1 Example

Consider the doubly nested loop in Figure 1. The bounds on i are $1 \le i$ and $i \le N$. We apply three rules and get $2 - s_i \le i$ and $i \le N$. Note that N is a program parameter and is not considered as an outer iterator. Similarly, we obtain $2 - s_j \le j$ and $j \le i + s_i - 1$. The final tile loops are shown in Figure 5. The main difference between this tile-loop and the one in Figure 4 is that there is no s_j in the lower bound on i. To obtain s_j in the lower bound of i, one needs an expensive operation, either Fourier-Motzkin elimination or projection. Our lower bound is tighter, but it still visits all the necessary tile origins.

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

Figure 5: Tile loops obtained from the example in Figure 1 using dimension-by-dimension tiling

Another difference is that our tile-loops visit fewer tile origins. In other words, the outset we implicitly construct is smaller than the outset proposed by Lakshminarayanan et al. In this example, we do not have *any* empty tile origins. The tile-loops in Figure 4 scan an empty tile origin. However, in general, our tile-loops may also visit empty tiles.

3.2 Why It Works

The main idea behind our tile-loop construction is that the tile space for a projection of a polyhedron is big enough to include all the full/partial tile origins. This makes it unnecessary to use polyhedral operations of exponential complexity that have been used in the all the existing techniques. In fact, our loop bound modification rules themselves are the constraint shifting operation. The main difference is the set of input constraints and the way to construct tile-loops.

As we said earlier, our outset is not precise. We still need to prove that the set we construct is an outset, i.e., it contains all the full/partial tile origins. This follows from the fact that the outset we construct is a particular instance of outsets presented by Lakshminarayanan et. al. [29]. The difference is that we may have some more redundant constraints. Note that there are many distinct outsets from a single set depending on its constraint form. The redundant constraints in the original iteration space may become non-redundant constraints in its outsets.

3.3 Discussion

Tile-loops that are generated by D-tiling scan no more number of tile origins than the previous method. However, they may visit more iterations at outer loops. Take the example in Figure 1 and modify the lower bound of i to $-N \leq i$. HiTLOG will generate the same tile-loops as it does from the original example because the bound is redundant and changing the lower bound of i does not affect the shape of the iteration space. However, the tile-loops generated by D-tiling will visit all the multiples of s_i between $-N - s_i + 1$ and N. Since the input loops are typically well engineered, the tile-loop overhead may be insignificant. Moreover, this aspect of D-tiling is quite useful when we apply tiling to imperfectly nested loops.

4 Tiled Loop Generation

So far, we have discussed only perfectly nested loops. This section provides an algorithm for generating tiled loops from imperfectly nested loops. First, we illustrate a simple scheme for tiling and discuss the legality issue. Then, we provide a detailed algorithm for the simple scheme. In subsection 4.5, we formulate a precise condition when the simple scheme works and when it does not. Finally, we explain how to modify the algorithm so that it generates tile-loops for more general case.

4.1 Running Example

Consider the loop nest in Figure 6 which is not perfectly nested like our previous example.

```
for (i = 1; i <= N;i++)
  for (j = 1; j <= i;j++)
     S1(i,j);
  S2(i);
```

Figure 6: An imperfectly nested loop where the iteration space of $S1$ is a triangular and that of $S2$ is a line segment: the body of the loop is represented with the macro S1 and S2.

There are two statements in the loops. The actual statements are not given to avoid the legality issue for the moment. We assume that it is legal to tile.

Now, we apply D-tiling to the doubly nested loop, and get a loop nest that is similar to the one in Figure 5 but has two statements. We peel out the last iteration of t_j tile-loop and annotate it with two statements $S1$ and $S2$. The modified loops are shown in Figure 7.

```
// tile-loops
for (ti = up(2-si,si); ti <= N; ti+=si)
  for (tj = up(2-sj,sj); tj <= ti+si-sj-1; tj+=sj)
    // point-loops (S1)
  tj = down(ti+si-1,sj);
  // point-loops (S1,S2)
```

Figure 7: Tile loops obtained from the example in Figure 6 using D-tiling and peeling out the last iteration of t_j loop

To generate point-loops, we take the original code and simply add tile bounds to the lower and upper bounds. For instance, the lower bound of i becomes 1 and t_i, i.e., $\max(1, t_i)$. The resulting point-loops will have all the statements in the original loops. We obtain the point-loops that have only $S1$ by simply not printing $S2$ from point-loops with all the statements. The final tiled code is shown in Figure 8.

```
// tile-loops
for (ti = up(2-si,si); ti <= N; ti+=si)
  for (tj = up(2-sj,sj); tj <= ti+si-sj-1; tj+=sj)
    // point-loops (S1)
    for (i=max(1,ti);i<=min(N,ti+si-1);i++)
      for (j=max(1,tj);j<=min(i,tj+sj-1);j++)
        S1(i,j);
  tj = down(ti+si-1,sj);
  // point-loops (S1,S2)
  for (i=max(1,ti);i<=min(N,ti+si-1);i++)
    for (j=max(1,tj);j<=min(i,tj+sj-1);j++)
       S1(i,j);
    S2(i);
```

Figure 8: Final tiled loops from the example in Figure 6

One may ask whether it is legal to tile in such a way. If $S1(i, j)$ is `y(i)+=L(i,j)*x(j)` and $S2(i)$ is `y(i)+=x(i)` as in the product of lower triangular matrix L with unit diagonal and a vector x, this tiling is legal and is what an expert programmer typically writes by hand. However, if this computation is a triangular linear system solver, this tiling is not always legal.

In the remainder of this section, we present the answer to the following three questions: (*i*) precisely what is such tiling, (*ii*) how to know whether such tiling is legal without human knowledge about the computation, and (*iii*) how to handle the case where such tiling is not legal.

4.2 Tile-Loop Generation

As we saw in the example, tile loop generation consists of two steps: (*i*) generating tile loop bounds using D-tiling and (*ii*) restructuring the tile-loops such that statements only appear in the innermost loops. Later, these statements in each innermost loop will be replaced by point-loops that contains only those statements.

A detailed algorithm is given in Figure 9. The algorithm is for an imperfectly nested loop where there is one common outermost loop. When there are multiple outermost loops, we can apply the algorithm to each one individually. One may apply this algorithm to the original loops in Figure 6 and easily obtain the tile-loop in Figure 7.

One aspect of this tiling is that each statement is tiled in exactly same way as each statement is tiled separately except the peeling of either first (or last) iteration or both. This is because we do not embed all the statements into a common space. Statements do not have more context. For example, statement $S2$ in the example is associated to neither t_j nor j.

In the algorithm, statements are placed into the immediately following loop if such a loop exists, but different loops are not combined. One may want to choose the preceding loop of the statements, if exists. This choice can be made in either way, if this tiling is legal. We will address the legality issue in subsection 4.5.

4.3 Point-Loop Generation

The point-loop generation is straightforward. For each bound in the original loop, we add an appropriate tile bound to it. For instance, we add t_i at the lower bound lb_i of i loop so that the new bound becomes $max(t_i, lb_i)$. Similarly, we add $t_i + s_i - 1$ at the upper bound ub_i of i loop so that the bound becomes $min(t_i + s_i - 1, ub_i)$. Note that the resulting tile-loops has statements only in the innermost loops and the set of statements will be replaced by point-loops that contains those statements. Depending on the set of statements, there may be a loop that contains no statements in the set. In such case, we do not print the loop itself.

4.4 Legality of Tiled-Loop Generation

We use a sequence of operations on a loop AST. Here we explain what the algorithm achieves and provide an informal argument for one aspect of legality of tiling: the tiled loops must execute the exactly same set of instances of each statement in the original loops. When tile-loops are constructed, statements may have been replicated because of peeling tile-loop iteration. Also, the number of surrounding loops of statements is changed in the peeled tile-loop iteration. If there is no peeling of surrounding loops of a statement at depth k, the number of its surrounding loops in the tiled loops is $2k$. However, if we reverse the loop peeling operations, each statement have exactly same tile-loops as those obtained by applying perfectly nested loop tiling to that statement alone. Also, The point-loops of each statement is exactly same as that of perfectly nested tiling to the statement. Consequently, this whole transformation is a bijection from an instance of a statement in the original loops to an instance of a statement in the final tiled loops. In other words, tiling is legal in terms of operations that the program performs. The legality issue of the order of operations will be discussed in the following subsection.

```
Input: $AST$ - imperfectly nested loops
1: Apply D-tiling to $AST$ /* To obtain tile loop bounds */
2:
3: for $L$ : each loop of $AST$ from depth 1 to maximum depth do
4:     $hasStatement \leftarrow false$
5:     $StmtList \leftarrow \emptyset$
6:     for $c$ : each child of $L$ do
7:         if $c$ is statement then
8:             add $c$ to $StmtList$
9:             $hasStatement \leftarrow true$
10:        else if $c$ is loop then
11:            if $hasStatement$ = true then
12:                peel the first iteration of $c$ and add $StmtList$ before the children of $c$
13:                $StmtList \leftarrow \emptyset$
14:                $hasStatement \leftarrow false$
15:                $LastLoop \leftarrow c$
16:            end if
17:        end if
18:    end for
19:    if $hasStatement$ = true then
20:        peel the last iteration of $LastLoop$ and add $StmtList$ after the children of
           $LastLoop$ /* If the first iteration of $LastLoop$ is peeled out, then add a guard
           statement to ensure that a tile is executed only once. */
21:        $hasStatement \leftarrow false$
22:    end if
23: end for
```

Figure 9: An algorithm for generating tile-loops

4.5 Legality Condition

Now we provide a legality condition similar to that of full permutability of perfectly nested loops. Unlike the legality condition for perfectly nested loops, the order of tiles cannot be described just in the terms of lexicographic order of tiles, but requires the information about textual order of tiles (point-loops) as well.

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

A legality condition on tiling is a certain restriction on dependences among tiles, not within points in a tile. For example, full permutability of loops, a well known legality condition for perfectly nested loops, imposes that dependences among tiles are non-negative in all components. So, the tile-loops themselves are fully permutable. It is based on the fact that the relative order among points within a tile remains same. This property holds also for our imperfectly nested loop tiling scheme.

Before deriving a condition, let us examine the legality issue of our running example. When

$S1(i,j)$ is `y(i)+=L(i,j)*x(j)` and $S2(i)$ is `y(i)+=x(i)`, there are two flow dependences:

$$\begin{aligned}(S1,(i,j)) &\rightarrow (S1,(i,j-1))\\ (S2,(i)) &\rightarrow (S1,(i,i-1))\end{aligned}$$

First, consider the dependence from (i,j) of $S1$ to $(i,j-1)$ of $S1$. Let (t_i^1,t_j^1) and (t_i^2,t_j^2) be the tile origins of the tiles containing (i,j) and $(i,j-1)$ respectively. Then, $t_i^1=t_i^2$ and t_j^2 is either t_j^1 or t_j^1-1. After tiling, this dependence will be preserved. This is also clear from the fact that the loop without $S2$ is fully permutable.

Now, consider the second dependence, $(S2,(i))\rightarrow(S1,(i,i-1))$. Let (tt_i^1,tt_j^1) and (tt_i^2,tt_j^2) be the tile origins of (i) and $(i,i-1)$. Then, $tt_i^1=tt_i^2$ because the i loop is common. We have only two possible cases: either the point-loop that contains the statement $S1$ textually precedes that of containing statement $S2$, or two statements are in the same point-loop. So, the dependence will be satisfied either by the textual order of point-loops or within a point-loop (tile).

Theorem 4.1. *An imperfectly nested loop tiling in the algorithm in Figure 9 is legal if for each data dependence d of (S,z) on (T,z'), the first n_{ST} components of $z-z'$ are non-negative and T textually precedes S.*

Proof of Theorem 4.1 Let $z=(z_1,\ldots z_{n_{ST}},\ldots,z_{n_S})$ and $z'=(z'_1,\ldots z'_{n_{ST}},\ldots,z'_{n_T})$. Let $t=(t_1,\ldots t_{n_{ST}},\ldots,t_{n_S})$ and $t'=(t'_1,\ldots t'_{n_{ST}},\ldots,t'_{n_T})$ be the tile origin of z and z', respectively. We are given that $z'_k\le z_k$ for $1\le k\le n_{ST}$. Then, $t'_k\le t_k$ for all $1\le k\le n_{ST}$. By the scheme of loop peeling on tile-loops or the order of loops, either of the following is true: any set of point-loops containing T textually precedes any set of point-loops containing S, or both S and T appear in the same point-loops. The main idea is that the textual order of statements are preserved after tiling.

This restriction on the direction of dependence and textual order is quite strong. Imagine that we just tile the first i dimension of the running example. For a given tile, any computation of $S1$ can be done without any computation of $S2$. Another view of this theorem is that tiling is applied to the common loops and the remainder of the loops are tiled independently.

However, some dependences in real programs do not satisfy this condition. Common examples are triangular linear system solver and matrix factorization such as LU and Cholesky decomposition. Let us further investigate these computations more carefully.

Now, we assume that our running example is a triangular linear system solver. Then, $S1(i,j)$ becomes `y(i)+=L(i,j)*x(j)` and $S2(i)$ is `y(i)/=L(i,i)`. There are three dependences:

$$\begin{aligned}(S1,(i,j)) &\rightarrow (S1,(i,j-1))\\ (S2,(i)) &\rightarrow (S1,(i,i-1))\\ (S1,(i,j)) &\rightarrow (S2,(j))\end{aligned}$$

From the previous analysis, we know that the first two dependences satisfy the condition in theorem 4.1. We only consider the last dependence. Since $j<i$, the first component of dependence is non-negative. However, the dependence direction from $S1$ to $S2$ is the same as the textual order of two statements. Let (t_i,t_j) be a tile origin of $(S1,(i,j))$. We want to find the iterations that produces the values being read by tile (t_i,t_j). The image of the tile by dependence function is the precise set of the iterations that produces such values. So, we get $\{i\in\mathcal{D}_{S2}\mid t_j\le i\le t_j+s_j-1\}$. On the other hand, the iterations that are executed in the tile t_i of $S2$ are $\{i\in\mathcal{D}_{S2}\mid t_i\le i\le t_i+s_i-1\}$. Figure 10 shows geometric view of this dependence relation. The value of t_j determines which iterations of $S2$ are being used. If these two sets are disjoint like two bottom tiles with origin at $(6,0)$ and

$(6,3)$, the dependences are satisfied. However, they are not disjoint for some tiles. Specially, if we decrease the size of s_j to 1, the tiled code generated by our method executes no iterations in $(S2,6)$, $(S2,7)$ and $(S2,8)$ until all the tiles whose $t_i = 6$. The intersection is not empty, so this is not legal execution order. So we cannot tile the region where these two sets are not disjoint.

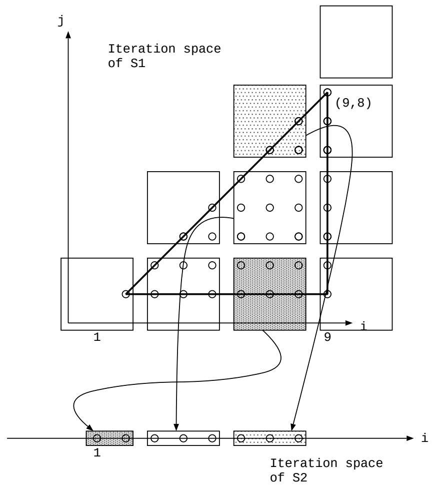

Figure 10: Tiles of $S1$ and the iterations of $S2$ on which they depend in the example of Figure 6

In the analysis of this example, we took a tile of a producer statement and applied the dependence function to it. We derived a precise condition when our tiling method becomes illegal. The following theorem gives a precise reasoning for the general cases.

Claim 4.2. For a dependence d of (S,z) on (T,z') where the components up to n_{ST} are non-negative and S textually precedes T, tiling is not always legal if $Proj(image(t,f),n_{ST}) \cap Proj(t,n_{ST}) \neq \emptyset$ where $z' = f(z)$ and t is a tile.

Note that we can compare the sets up to the common dimensions. This condition just tells us when tiling is not legal, but it can also be used to guide tiled loop generation.

We elaborate what this theorem tells us with the help of another example. Consider the loop for Cholesky decomposition in Figure 11. There are three statements and six (value-based) dependences.

$$\begin{array}{rcl}
(S1,(k)) & \rightarrow & (S3,(k-1,k,k)) \\
(S2,(k,i) & \rightarrow & (S3,(k-1,i,k)) \\
(S2,(k,i)) & \rightarrow & (S1,(k)) \\
(S3,(k,i,j)) & \rightarrow & (S3,(k-1,i,j)) \\
(S3,(k,i,j)) & \rightarrow & (S2,(k,i)) \\
(S3,(k,i,j)) & \rightarrow & (S2,(k,j))
\end{array}$$

The last four dependences satisfy the condition in Theorem 4.1, but the first two dependences do not satisfy that condition. Now, let t be a tile of $S1$. The set of points in the tile can be described as $\{k \mid t_k \leq k \leq t_k+s_k-1\}$. Now, take an image of t_k by the dependence function $(k \rightarrow k-1,k,k)$. The points of $S3$ in the image of tile $\{k,i,j \mid t_k-1 \leq k \leq t_k+s_k-2; t_k \leq i \leq t_k+s_k-1; t_k \leq j \leq t_k+s_k-1\}$ are the value that t is depends on. Similarly, we do the same analysis for the second dependence. The iterations of $S3$ on which a tile $t' = \{k,i \mid \{t_k \leq k \leq t_k+s_k-1; t_i \leq i \leq t_i+s_i-1\}$ of $S3$ depend are $\{k,i,j \mid t_k-1 \leq k \leq t_k+s_k-2; t_k \leq i \leq t_k+s_k-1; t_k \leq j \leq t_k+s_k-1\}$. To make tiling legal, we can impose that the intersection of the projection of those iteration with a tile is empty. For instance, to make tiling legal on the first dependence, the projection $\{k \mid t_k - 1 \leq k \leq t_k + s_k - 2\}$ of the image is disjoint with the original tile $\{k \mid t_k \leq k \leq t_k + s_k - 1\}$. Then, the condition becomes $s_k = 1$. If this condition is imposed, the second dependence has the same property that the intersection of projections of—a tile and its image by dependence function—on the common dimension is empty. We know that tiling of only k loops are legal. Another way to make tiling legal is to map all the points in the image into a single point-point, i.e., not to tile.

```
for (k = 1; i < N;i++)
  A(k,k)=sqrt(A(k,k));          -- S1
  for (i = k+1; j < N;j++)
     A(i,k)/=A(k,k);            -- S2
     for (j = k+1; j <= i;j++)
       A(i,j)-=A(i,k)*A(j,k); -- S3
```

Figure 11: Cholesky decomposition

4.6 Tiled Loop Generation

Our strategy is that we do not tile when tiling is not legal at certain area or impose an condition on the relation between tile sizes so that the generated code using the algorithm in Figure 9. For example, we produce the tiled loops shown in Figure 12 for a triangular linear system solver. We do not tile as soon as tiling become not legal. Note that the point-loops for $S1$ and $S2$ do not have tile upper bound and after the point loops t_j is assigned to its upper bound. Using our simple tiled loop generation algorithm with this simple modification, we can the code that appears in the literature that address tiling imperfectly nested loops.

With the knowledge of legality issue, one may be able to choose a different approach for tiled loop generation. Another way to efficiently use our technique is developing an enabling transformation so that the parameterized tiled generation is legal and simple, and the resulting code is efficiency. A detailed study for enabling transformation is beyond the scope of this paper.

Our algorithm provide the separation of perfectly nested point-loops and imperfectly nested point-loops. One may want to separate full tiles from empty/partial tiles. We can directly use the inset [29], which contains only full tile origins, by constructing it dimension by dimension. The inset of a point-loop is just the inset of its surrounding loops.

```
// tile-loops
for (t_i = up(2-s_i,s_i); t_i <= N; t_i+=s_i)
  for (t_j = up(2-s_j,s_j); t_j <= t_i+s_i-1; t_j+=s_j)
    // tiling is legal
    if ( t_i+s_i-1<t_j || t_j+s_j-1<t_i )
      // point-loops (S1)
      for (i=max(1,t_i);i<=min(N,t_i+s_i-1);i++)
        for (j=max(1,t_j);j<=min(i,t_i+s_i-1);j++)
          S1(i,j);
    // tiling is not legal
    else
      // point-loops (S1,S2)
      for (i=max(1,t_i);i<=min(N,t_i+s_i-1);i++)
        for (j=max(1,t_j);j<=i);j++)
          S1(i,j);
        S2(i);
      t_j = t_i+s_i-1; // last iteration
```

Figure 12: Tiled loops for a triangular linear system solver.

5 Implementation and Experimental Result

We implemented our tiled loop generation algorithm as simple visitors written in Java on a loop AST(Abstract Syntax Tree) generated by the SableCC compiler generator [11]. Our code generator generates parameterized tiled loops as well as fixed tiled loops. Also, tile-loops and point-loops can be generated independently. The generator will be available as an open source toolkit.

5.1 Experimental Setup

To evaluate the efficiency of our code generator, we used four common benchmarks in Table 1. We compared our method with a fixed size tiling using CLOOG [6]. We used gmp-enabled CLOOG 0.14.1. All the statements are embedded into a common high dimensional polyhedron and we use Xue's formulation [34] to generate fixed tiled loops. We tiled all the loops.

We ran all experiments for generated code efficiency evaluation on Intel Core 2 Duo running 2.2 GHz with 2MB L2 Cache and 1GB memory. We compiled all the code using gcc 4.1.2 with the optimization level -O3. The timings were measured using gettimeofday(). For generation efficiency evaluation, we use only default option of CLOOG and reports the generation time given by CLOOG. All the I/O times are ignored. To measure our code generation time we use currentTimeMillis() method in java.lang.System class.

	Description	loop depth/s-tate-ments
MatMul	Matrix multiplication	3/2
MultiTriSolver	Multiple triangular linear systems solver	3/2
LUD	LU decomposition of a matrix without pivoting	3/2
Cholesky	Cholesky decomposition	3/3

Table 1: Benchmarks used for generation efficiency and generated code quality evaluation

5.2 Results

We first profile the code generation time of our code generator and that of fixed embedding approach using CLOOG. For fixed size tiling, we generated all the tile sizes that is a power of 2 and between 2 and 512. We use data when tile sizes are 64. When tile sizes are small like 2 and 4, the generation is fast and the size of code is small. After that, the generation time and code sizes are almost same. The generated data is shown in Table 2. Even though all the benchmarks have a relatively small number of statements and the maximum depth, fixed tile size tiling took 8.78 seconds for LUD. In terms of generation efficiency, our code generator is significantly faster. We also measured the number of lines. The CLOOG generated code is highly optimized based on the range of program parameters. For a specific range, the number of lines is a lot smaller than the reported number. However, it is still larger than the code generated by our method.

Benchmarks	Parameterized	Fixed + Embedding
MatMul	0.009 sec/ 24 lines	0.33 sec/ 41 lines
MultiTriSolver*	0.025 sec/ 25 lines	0.67 sec/ 224 lines
LUD	0.002 sec/ 24 lines	8.78 sec/ 5583 lines
Cholesky	0.004 sec/ 47 lines	2.38 sec/ 1287 lines

Table 2: Comparison of generation time and the number of lines—parameterized vs fixed tiling—an embedding is given for fixed size tiling (*: the generated code is modified by hand)

Figure 13-17 shows the execution time and loop overhead of the four benchmark. The loop overhead is measured by replacing the actual statements with a simple statement that increments a scalar variable.

Figure 13 shows the total execution time of MatMul. We only measured the execution time of matrix multiplication because the structures of loops from two different approach are very similar. The program is written in a trivial triply nested loops, and the innermost loop accumulated the values. There is an initialization statement just before the innermost loop. The embedding function for the initialization statement is $(i, j \rightarrow i, j, 0)$, i.e., it is aligned at $k = 0$. The efficiency of our generated code is as good as that of fixed size tiling with embedding.

Figure 14 and 15 shows the execution time and the loop overhead of MultiTriSolver. The outermost loop i iterates over problem instances of a single triangular system solver. We first generated the code using our generator and then modified it so that it does not tile when tiling is not legal. So, its tile-loops are perfectly nested and the tile-loops has a guard of testing the legality. The embedding function is the same as the one given in [1]. The division statement has an embedding function $(i, j, k \rightarrow i, j, j)$. When tile sizes are very small, the execution time of our code is larger than that of fixed size tiling with embedding. When tiles size are more than 8, they become similar. In terms of loop overhead, our code is better than the other.

For Cholesky and LUDom, we generated parameterized tiled loops using our code generator, and then restricted the tile sizes (rather than introducing a guard) so that the generated code is correct. In Cholesky, the loop overhead is comparable for most of tile sizes, but the total execution time is 40 percent higher than that of fixed size tiling with embedding. In LUDom, the loop overhead of fixed size tiling with embedding approach is high, and the execution time has a similar pattern with MultiTriSolver.

In summary, in terms of the efficiency of tiled loop generation our method is better than fixed size tiling with embedding. The generated code from our techniques is more compact and readable. Our generated code has more loop overhead when tile sizes are small and similar or less when tile sizes are big. In terms of total execution time of benchmark, the efficiency of our generated code is comparable when tile size is not small. Although our code is not as efficient as that of fixed embedding tiling, we expect that the imperfectly nested tiling is used with bigger tile sizes.

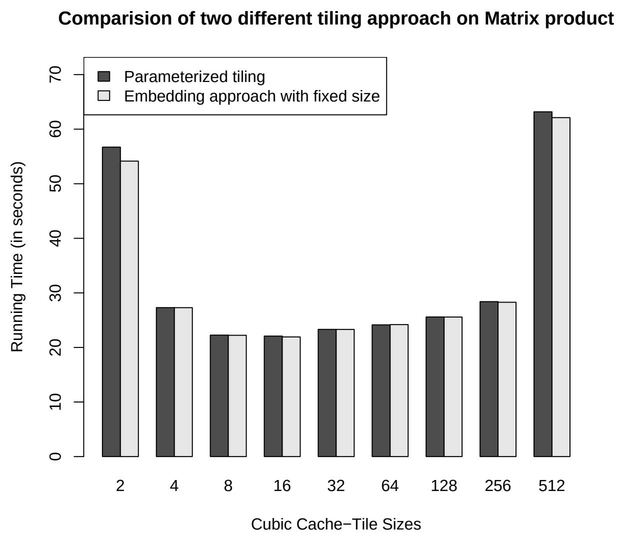

Figure 13: Total execution time for MatMul on two 2000×2000 matrices

6 Related Work

Ancourt and Irigoin [4] proposed a technique for generating loops scanning a single polyhedron using Fourier-Motzkin elimination over inequality constraints. Le Verge et al. [20, 21] proposed a technique using the dual representation of polyhedra. The loop generation from a polyhedron is often required after uni-modular transformations such as loop permutation and skewing.

Irigoin and Triolet [15] show that the tiled iteration space after fixed size tiling can be formulated as a polyhedron with higher dimension by adding d dimensions. For example, the 3×3 tiled iteration space of our example is

$$\{t_i, t_j, i, j \mid \quad \begin{aligned} &3t_i \le i \le 3t_i + 2; 3t_j \le j \le 3t_j + 2 \\ &; i \le N; 1 \le j \le i \end{aligned} \quad \}$$

Either of the above tools may be used (in fact, most of them can generate such tiled code). However, it is well known that since the worst case complexity of Fourier-Motzkin elimination is doubly exponential in the number of dimensions, this may be inefficient especially when multi-level tiling is needed. Due to this problem, Goumas et al. [12] proposed a method where the tiled loop generation problem is decomposed into two sub-problems, one to scan the tile origins, and the other to scan points within a tile, thus obtaining significant reduction of the worst case complexity.

Another disadvantage of the above tiled iteration space formulation is that it is no longer a polyhedron when tile sizes are not constants. The constraints become bi-linear forms. For the case where tile sizes are symbolic parameters, a simple case called *orthogonal* tiling — either rectangular loops tiled with rectangular tiles, or loops that can be easily transformed to this — was first considered. For the more general case, the standard solution, as described in Xue's text [34] has been to simply *extend* the iteration space to a rectangular one (i.e., to consider its bounding box), apply the orthogonal technique with appropriate guards to avoid computations outside the original iteration space. Jiménez et al. [16] develop code generation techniques for register tiling of non-rectangular iteration spaces. They generate code that traverses the bounding box of the tile iteration space to enable parameterized tile sizes, but the focus of their paper is applying index-set splitting

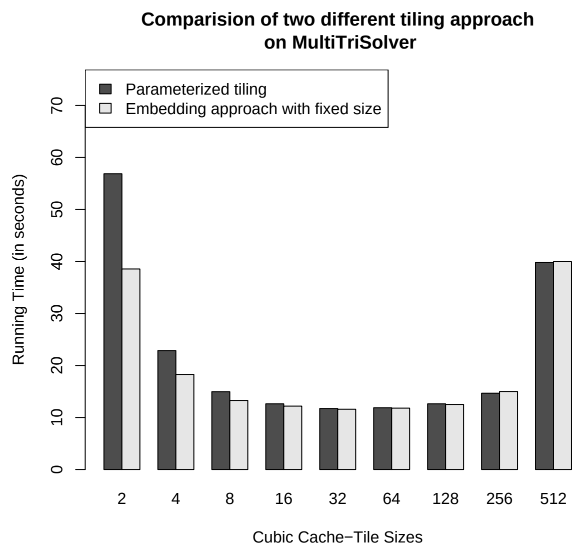

Figure 14: Total execution time for MultiTriSolver on 2000 variables and 3000 instances

to tiled code to traverse parts of the tile space that include only full tiles. Amarasinghe and Lam [2, 3] implemented, in the SUIF tool set, a version of FME that can deal with a limited class of symbolic coefficients (parameters and/or block sizes), but the full details have not been made available. Größlinger et al. [13] proposed an extension to the polyhedral model, in which they allow arbitrary rational polynomials as coefficients in the linear constraints that define the iteration space. Their genericity comes at the price of requiring computationally expensive machinery like quantifier elimination in polynomials over the real algebra, to simplify constraints that arise during loop generations. Due to this their method does not scale with the number of dimensions and the number of non-linear parameters. Lakshminarayanan et al. [29] proposed an approach where tile space is parameterized by tile sizes in addition to program parameters.

However, these techniques, decomposition and tile size parameterization, are restricted to perfectly nested loops. When tile sizes are constants, tiling has been extended to imperfectly nested loops. Although some authors do not consider arbitrary affine control loops, they proposed limited extension to imperfectly nested loops. Carr and Kennedy [9] proposed a technique that use an index set splitting to make tiling legal before it is applied. Song and Li [31] proposed a technique for stencil programs consisting of one outer time loop and a sequence of perfectly nested loops within the time loop.

Several general approaches have been proposed for tiling imperfectly nested loops. Roughly speaking, the existing techniques for arbitrary nested loops consist of the following steps: (i) converting them to perfectly nested loops (or embedding each iteration space into a common space), and (ii) applying all the knowledge developed for perfectly nested loops, such as legality conditions, finding linear transformations for enhancing data locality in the program, etc., and (iii) code generation. Unfortunately, the code generation step in these approaches is not as simple as that for perfectly nested loops, often requiring index set splitting to get efficient final code. This inherently comes from the first step, where guards or equalities are introduced to embed a statement surrounded by fewer loops into a higher dimensional loop nest. Also, finding an appropriate embedding function itself is not a trivial problem. Also, the previous work on imperfectly nested loop tiling focused on how to find embedding functions and tiled loop generation are left as a scanning

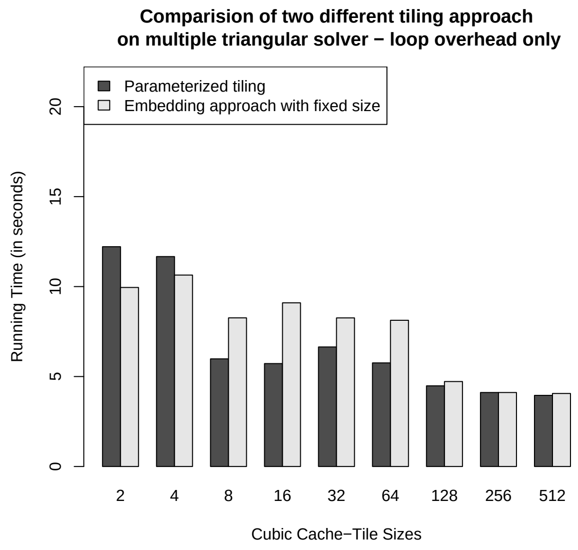

Figure 15: Loop overhead of MultiTriSolver on 2000 variables and 3000 instances

a union of polyhedron.

Ahmed et al. [1] proposed a technique for general imperfectly nested loops based on embedding into a common Cartesian product space. Lim et al. [22] 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. [7] 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. The transformed loops are fully permutable. Also, our focus is to generate parameterized tiled loops from imperfectly nested loops, rather than finding a good embedding.

7 Conclusion

Tiling imperfectly nested loops is an important loop transformation that enables us to parallelize programs and improve the overall performance of more general applications. Several approaches have been proposed for fixed imperfectly nested loops. Although parameterized tiled loops where tile sizes are a symbolic parameters are a quite useful, the existing methods only handle perfectly nested loops. We proposed D-tiling that reduces the complexity of the generation of fixed/parameterized/mixed tiled loops for perfectly nested loops to $O(m \times (d+p))$. We presented the technique for parameterized tiled loop generation problem based on the reasoning of legality conditions without making the original loop nest into a perfectly nested one. We compare the efficiency of our code generation and the generated code with fixed embedding approach. Experimental result shows that the efficiency of code generation is better than that of fixed embedding approach and the efficiency of the generated code is as good as that of fixed size tiling with embedding.

Our ongoing work includes a more detailed study on the relation between dependence presented in imperfectly nested loops and tiled loop generation. Our aim is to generate a more efficient code based on the constraints on tile sizes.

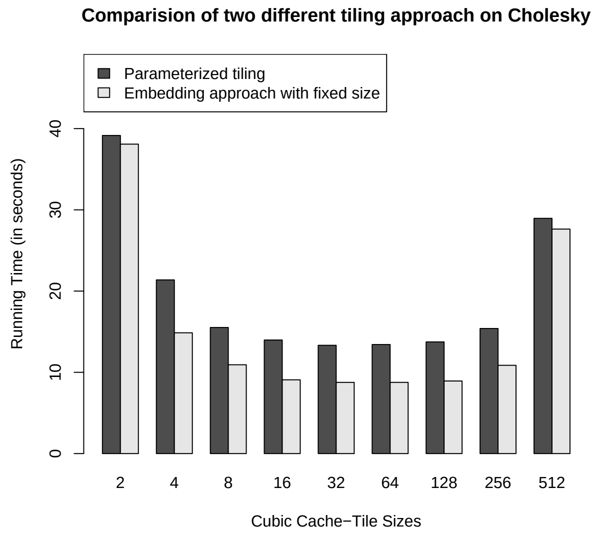

Figure 16: Total execution time for Cholesky on 3000×3000 matrix

References

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 31, Washington, DC, USA, 2000. IEEE Computer Society.

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

[3] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 126–138, New York, NY, USA, 1993. ACM Press.

[4] C. Ancourt and F. 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.

[5] R. Andonov and S. Rajopadhye. Optimal orthogonal tiling of 2-d iterations. *Journal of Parallel and Distributed Computing*, 45(2):159–165, September 1997.

[6] 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.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–113, New York, NY, USA, 2008. ACM.

[8] P. Boulet, A. Darte, T. Risset, and Y. Robert. (pen)-ultimate tiling? *INTEGRATION, the VLSI journal*, 17:33–51, August 1994.

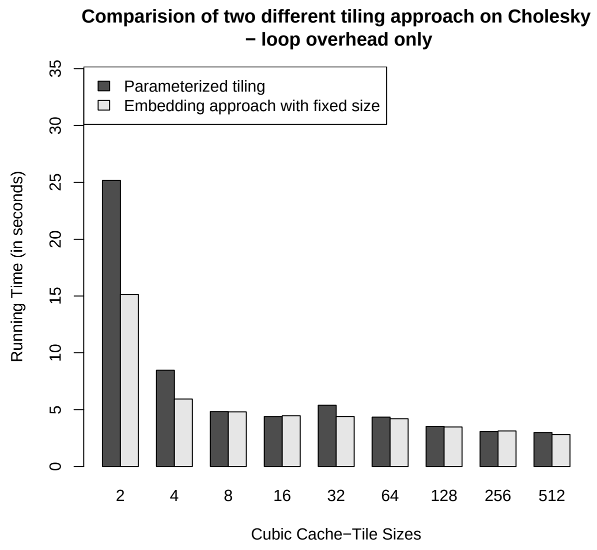

Figure 17: Loop overhead of Cholesky for the matrix of size 3000 × 3000 matrix

[9] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. In *In Proceedings of Supercomputing '92*, pages 114–124, 1992.

[10] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

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

[12] G. Goumas, M. Athanasaki, and N. Koziris. An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10), October 2003.

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

[14] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Principles of Programming Languages*, pages 160–173, Paris, France, January 1997. ACM.

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

[16] M. Jiménez, J. M. Llabería, and A. Fernández. Register tiling in nonrectangular iteration spaces. *ACM Trans. Program. Lang. Syst.*, 24(4):409–453, 2002.

[17] 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.

[18] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In *SC '07: Proceedings of*

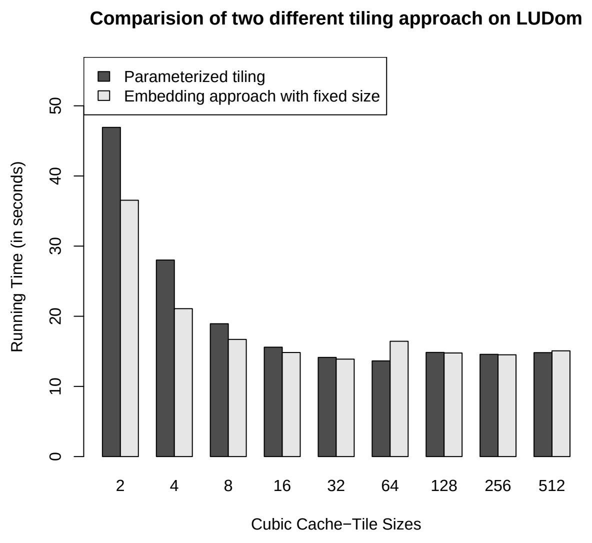

Figure 18: Total execution time of LUDom for the matrix of size 3000 × 3000 matrix

the 2007 ACM/IEEE conference on Supercomputing, pages 1–12, New York, NY, USA, 2007. ACM.

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

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

[21] 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.

[22] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and paractices of Parallel Programming*, pages 103–112, New York, USA, 2001. ACM Press.

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

[24] D. 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.

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

[26] 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.

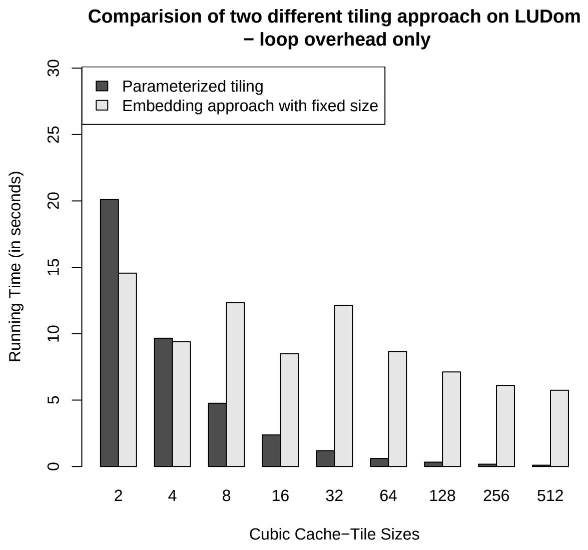

Figure 19: Loop overhead of LUDom for the matrix of size 3000 $\times$ 3000 matrix

[27] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal Parallel Programming*, 28(5):469–498, 2000.

[28] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[29] 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.

[30] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, RIACS, NASA Ames Research Center, August 1990.

[31] Yonghong Song and Zhiyuan Li. A compiler framework for tiling imperfectly-nested loops. In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 185–200, London, UK, 2000. Springer-Verlag.

[32] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[33] R. P. Wilson, R. S. French, C. 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.

[34] J. Xue. *Loop Tiling For Parallelism*. Kluwer Academic Publishers, 2000.