Parameterized Tiled Loops for Free

Lakshminarayanan Renganarayanan DaeGon Kim Sanjay Rajopadhye Michelle Mills Strout

Computer Science Department
Colorado State University
{ln,kim}@cs.colostate.edu Sanjay.Rajopadhye@colostate.edu mstrout@cs.colostate.edu

Abstract

Parameterized tiled loops—where the tile sizes are not fixed at compile time, but remain symbolic parameters until later—are quite useful for iterative compilers and "auto-tuners" that produce highly optimized libraries and codes. Tile size parameterization could also enable optimizations such as register tiling to become dynamic optimizations. Although it is easy to generate such loops for (hyper) rectangular iteration spaces tiled with (hyper) rectangular tiles, many important computations do not fall into this restricted domain. Parameterized tile code generation for the general case of convex iteration spaces being tiled by (hyper) rectangular tiles has in the past been solved with bounding box approaches or symbolic Fourier Motzkin approaches. However, both approaches have less than ideal code generation efficiency and resulting code quality. We present the theoretical foundations, implementation, and experimental validation of a simple, unified technique for generating parameterized tiled code. Our code generation efficiency is comparable to all existing code generation techniques including those for fixed tile sizes, and the resulting code is as efficient as, if not more than, all previous techniques. Thus the technique provides parameterized tiled loops for free! Our "one-size-fits-all" solution, which is available as open source software can be adapted for use in production compilers.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Compilers, Optimization

General Terms Algorithms, Experimentation, Performance

Keywords parameterized tiling, bounding box, Fourier-Motzkin elimination, code generation

1. Introduction

Tiling [12, 27, 18, 31] is a loop transformation that matches program characteristics (locality, parallelism, etc.) to those of the execution environment (memory hierarchy, registers, number of processors, etc.) Many problems relating to tiling have been extensively studied: how to pre-process a loop to make tiling legal (e.g. loop-skewing and other unimodular transformations) [31, 18]; tile shape optimization [7, 26, 11]; and tile size selection to optimize for memory hierarchy as well as interprocessor communication [8, 4]. However, as noted by Goumas et al. [9], the code generation problem after tiling has not received as much attention. Until their paper, most compilers and automatic parallelizers did not generate tiled-code for arbitrary parallelepiped-shaped tiles, and arbitrary polyhedral iteration spaces, even though an algorithm was described in the early work of Irigoin and Triolet [12]. The techniques of Goumas et al. are the current state of the art when the tile sizes are fixed at compile time. In this paper we address the problem when tile sizes are not compile-time constants, but remain symbolic parameters in the code.

There are many reasons why the *parameterized* tiled code generation problem is important. First, iterative compilers [16, 17] and "autotuners" or application-specific code generators such as ATLAS [29] and SPIRAL [24], optimally tune parameters including tile sizes, through exploration of a design-space of parameter values. A recent study of tiling for stencil computations [14] found that selecting the tile size that results in the "best" performance is difficult. With a fixed tiled code generator, the code needs to be repeatedly generated and recompiled for each tile size, whereas, with a parameterized tiled code generator, the code is generated only once and used for all the tile sizes.

Second, parameterized tiled code enables run-time feedback and dynamic program adaptation. For example, run-time tile size adaptation has been successfully used improve execution on shared cache processors [22] and also for adapting parallel programs to varying workloads [21]. Finally, parallelizing compilers should generate code that enables the number of processors to be set at run time [2]. For polyhedral iteration spaces, this problem is similar to the general problem of generating parameterized tiled code; therefore, any solution for generating parameterized tiled code can be directly adapted to enable setting the number of processors at runtime.

There is an easy solution to the parameterized tiled loop generation problem: simply produce a parameterized tiled loop for the *bounding box* of the iteration space, and introduce guards to test whether the point being executed belongs to the original iteration space. When the iteration space is itself (hyper) rectangular, as in matrix multiplication, this method is obviously efficient. However, many important computations, such as LU decomposition, triangular matrix product, symmetric rank updates, do not fall within this category. Moreover, even if the original iteration space is (hyper) rectangular, the compiler may choose to perform skewing transformations to exploit temporal locality (e.g. stencil computations) thus rendering it parallelepiped shaped. Parallelepiped-shaped iteration spaces also occur when skewing is performed to make (hyper) rectangular tiling legal. For such programs, the bounding box strategy results in poor code quality, because a number of so called "empty tiles" are visited and tested for emptiness. Another drawback for the bounding box strategy is that calculating the bounding box of arbitrary iteration spaces may be time-consuming. The worst-case time complexity of computing a bounding box is exponential [5].

PLDI'07 June 11–13, 2007, San Diego, California, USA.

The main difficulty with generating parameterized tiled loop code has been the fact that the Fourier-Motzkin elimination technique that is used for scanning polyhedra [3] does not naturally handle symbolic tile sizes, and leads to a nonlinear formulation. Amarasinghe proposed a symbolic extension of the standard Fourier-Motzkin elimination technique [2, 1] and implemented it in the SUIF system [30]. It is well known that Fourier-Motzkin elimination has doubly exponential worst case complexity. The symbolic extension inherits this worst case complexity, adds to the number of variables in the problem, and reduces the possibilities for redundancy elimination.

In this paper, we present a simple and efficient approach for generating parameterized tiled code that handles any polyhedral iteration space and parameterized (hyper) rectangular tilings. We show that the problem can be formulated into the subproblems of generating loops that iterate over tile origins, and loops that iterate over the points within tiles. These subproblems can be formulated as a set of linear constraints where the tile sizes are parameters, similar to problem size parameters. This allows us to reuse existing code generators for polyhedra, such as CLooG [6], and implement our code generator through simple pre- and post-processing of the CLooG input and outputs. The key insight is expressing the bounds for the tile loops as a superset of the original iteration space and then post processing the generated loops by adding a stride and modifying the computation of the lower bounds. In addition, we develop and prove the correctness of two loop overhead optimization techniques that avoid visiting empty tiles and avoid unnecessary guards for full tiles. This paper makes the following contributions:

- We present an algorithm that generates tiled loops from any parameterized polyhedral iteration space, while keeping the tile sizes symbolic variables. The fact that our algorithm can be directly applied to the case when the tile sizes are fixed, makes our method a one-size-fits-all solution, ideal for inclusion in production compilers.
- An empirical evaluation on benchmarks such as LUD and triangular matrix product show that our algorithm is both efficient and delivers good code quality. Our experiments present the first quantitative analysis of the cost of parameterization in tiled loops.
- We also present an algorithm that separates the loops into those that iterate over partial tiles and those that iterate over full tiles. Such a separation has the added benefit that it enables transformations like loop unrolling or software pipelining, (which are often applied only to rectangular loops) to be applied to the (rectangular) loops that iterate over the full tiles.
- Our implementation is available as open source software [28].

In the next section, we present the important issues involved in tiled code generation. Next, Section 3 resolves the first problem, namely scanning the tile origins, and Section 4 describes how to scan the individual tiles. In Section 5, we present the experimental validation of our method. Section 6 then describes the optimization that enables splitting the full and partial tiles, together with the proof of the technique. Section 7 discusses related work, and we conclude in Section 8.

2. Anatomy of Tiled Loop Nests

Tiling is an iteration reordering transformation that transforms a d-depth loop nest into one of depth up to $2d$. In this section we study the structure of tiled loops and develop an intuition for the concepts involved in generating them. In later sections, these concepts are formalized and used in deriving a simple and efficient algorithm for the generation of tiled loops.

```
for (k = 1; k <= N_k;k++)
  for (i = k+1; i <= k+N_i;i++)
          S1(k,i);
```

Figure 1. 2D iteration space found commonly in stencil computations. The body of the loop is represented with the macro S1 for brevity.

Consider the iteration space of a 2D parallelogram such as the one shown in Figure 1, which is commonly found in stencil computations [18]. Figure 2 shows a geometric view of the iteration space superimposed with a 2×2 rectangular tiling. Observe that there are three types of tiles: *full*—which are completely contained in the iteration space, *partial*—which have a partial, non-empty intersection with the iteration space, and *empty*—which do not intersect the iteration space. The lexicographically earliest point in a tile is called its *origin.* The goal is to generate a set of loops that *scans* (i.e., visits) each integer point in the original iteration space based on the tiling transformation, where the tiles are visited lexicographically and then the points within each tile are visited lexicographically. We can view the four loops that scan the tiled iteration space as two sets of two loops each, where the first set of two loops enumerate the tile origins and the next set of two loops visit every point within a tile. We call the loops that enumerate the tile origins the *tile-loops* and those that enumerate the points within a tile the *point-loops.*

2.1 Bounding Box Method

One solution for generating the tile-loops is to have them enumerate every tile origin in the bounding box of the iteration space and push the responsibility of checking whether a tile contains any valid iteration to the point-loops. The tiled loop nest generated with this bounding box scheme is shown in Figure 3. The first two loops (`kT` and `iT`) enumerate all the tile origins in a bounding box of size $N_k \times (N_i + N_k)$ and the two inner loops (`k` and `i`) scan the points within a tile. A closer look at the point-loop bounds reveals its simple structure. One set of bounds are from what we refer to as the *tile box bounds,* which restrict the loop variable to points within a tile. The other set of bounds restricts the loop variable to points within the iteration space. Combining these two sets of bounds we get the point loops that scan points within the iteration space and tiles. Geometrically, the point loop bounds correspond to the intersection of the tile box (or rectangle) and the iteration space, here the parallelogram in Figure 2.

The bounding box scheme provides a couple of important insights into the tiled loop generation problem. First, the problem can be decomposed into the generation of tile-loops and the generation of point-loops. Such a decomposition leads to efficient loop generation, since the time and space complexity of loop generation techniques is a doubly exponential function of the number of bounds. The second insight is the scheme of combining the tile box bounds and iteration space bounds to generate point-loops. Another important feature of the bounding box scheme is that tile sizes need not be fixed at loop generation time, but can be left as symbolic parameters. This feature enables generation of *parameterized tiled loops,* which has many applications as discussed in the previous section. However, the bounding box scheme can suffer from inefficiency in the generated loops in that the tile-loops can enumerate many empty tiles.

2.2 When Tile Sizes Are Fixed

When the tile sizes can be fixed at the loop generation time an *exact* tiled-loop nest can be generated. Tile-loops that only enumerate origins of tiles that have a non-empty rational intersection with

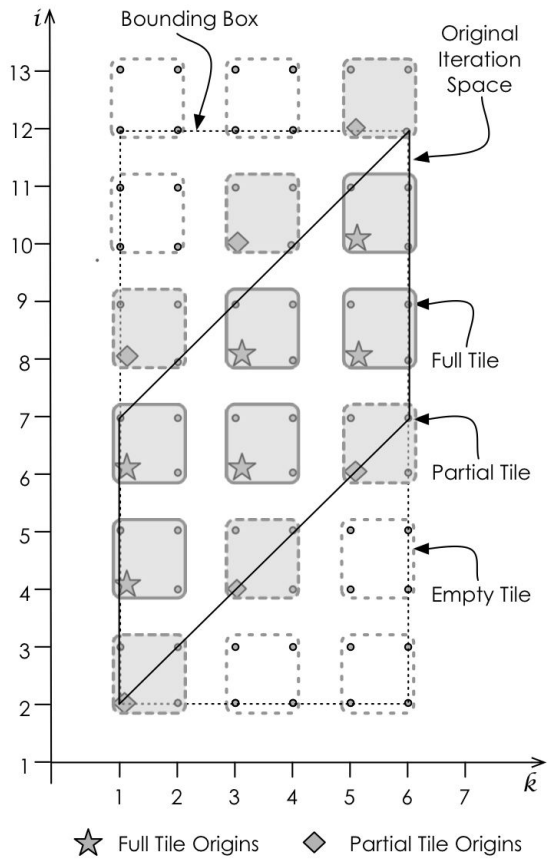

Figure 2. A 2×2 rectangular tiling of the 2D stencil iteration space with $N_i = N_k = 6$ is shown. The bounding box of the iteration space together with full, partial, and empty tiles and their origins are also shown.

```
for (kT = 1; kT <= Nk; kT += Sk)
 for (iT = 2; iT <= Ni+Nk; iT += Si)
  for (k= max(kT,1);k<=min(kT+Sk-1,Nk);k++)
   for (i= max(iT,k+1);i<=min(iT+Si-1,k+Ni);i++)
       S1(k,i);
```

Figure 3. Tiled loops generated using the bounding box scheme.

the iteration space are exact. Ancourt and Irigoin [3] proposed the first and now classic solution for generating the exact tiled loops when the tile sizes are fixed. When the tile sizes are fixed the tiled iteration space can be described as a set of linear constraints and the loops that scan this set can be generated using Fourier-Motzkin elimination [3, 31]. The exact tiled loop nest for the 2D stencil example is shown in Figure 4. Note that the efficiency due to the exactness of the tile-loops has come at the cost of fixing the tile sizes at generation time. Such loops are called *fixed tiled loops*.

The classic scheme, in addition to requiring fixed tile sizes, also suffers from loop generation inefficiency. It takes as input all the constraints that describe the bounds of the $2d$ loops of the tiled iteration space, where d is the depth of the original loop nest. Since the method is doubly exponential on the number of constraints, this increased number of constraints might lead to situations where the loop generation time may become prohibitively expensive [9].

```
for(kT=0;kT<=⌊(Nk/2)⌋;kT++)
 for(iT=max(1,kT);
   iT<=min(⌊(2*kT+Ni+1)/2⌋,⌊Nk+Ni/2⌋);iT++)
  for(k=max(max(1,2*kT),2*iT-Ni);
      k<=min(min(2*kT+1,2*iT),Nk);k++)
   for(i=max(2*iT,k+1);
       i<=min(2*iT+1,k+Ni);i++)
          S1(k,i) ;
```

Figure 4. Tiled loops generated for fixed tile sizes using the classic scheme.

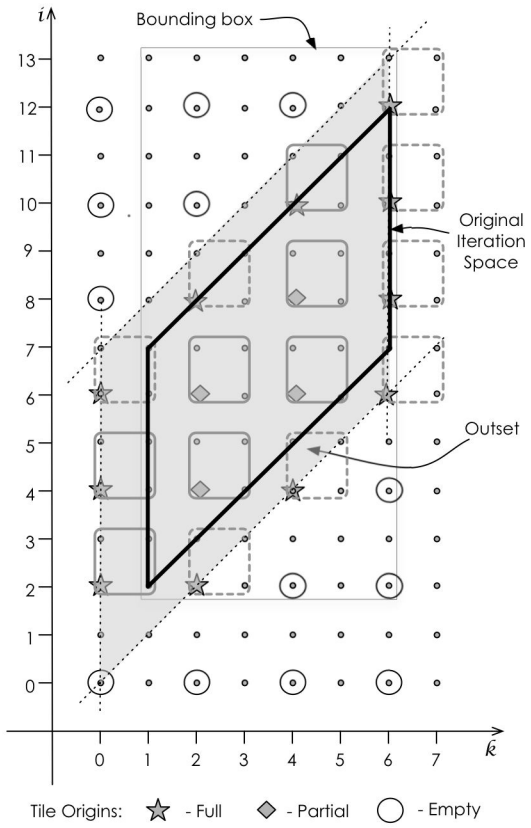

Figure 5. A 2×2 rectangular tiling of the 2D stencil iteration space with $N_i = N_j = 6$. The outset and bounding box are also shown. Compare the number of empty tile origins contained in each of them.

Goumas et al. [9] improve on the classic scheme by dividing the loop generation problem into two subproblems, similar to the approach taken with bounding box, but their generated code visits fewer empty tiles than bounding box. However, their solution is still only applicable to fixed tile sizes.

2.3 Best Of Both

We propose a tiled code generation method that achieves the best of both worlds: the simple decomposed loop structure used by the bounding box method and the Goumas et al. technique, the code quality provided by the fixed tile size methods, and the benefits of parameterized tile sizes provided by the bounding box method. We develop formal theory and use it to derive a method which provides efficient generation of efficient parameterized tiled loops.

```
kTLB = -Sk+2; kTLB = ⌈kTLB/Sk⌉*Sk;
for(kT = kTLB; kT <= Nk; kT += Sk)
 iTLB = kT-Si+2; iTLB = ⌈iTLB/Si⌉*Si;
 for(iT = iTLB; iT <= kT+Ni+Sk-1; iT += Si)
  for(k= max(kT,1);k<=min(kT+Sk-1,Nk);k++)
   for(i=max(iT,k+1);i<=min(iT+Si-1,k+Ni);i++)
          S1(k,i);
```

Figure 6. Parameterized tiled loops generated using outset. The variables kTLB and iTLB are used to shift the first iteration of the loop so that it is a tile origin, and explained later (Section 3.2.2).

The key insight is the construction of a set called the *outset*, which contains all possible tile origins for non-empty tiles. The *outset* is similar to the Tile Origin Space (TOS) constructed by Goumas et al. [9], but there are two important differences. First, the outset we construct includes the tiles sizes as parameters, whereas the tile sizes are fixed for the TOS. Second, we feed the outset to any code generator capable of scanning polyhedra, and then post-process the resulting code to add a step size and shift the lower bounds of the tile loops. Goumas et al. generate tile loops that iterate over the image of the TOS after applying tiling.

The outset has all the benefits of a bounding box, but enumerates very few empty tiles. In general, it is parameterized by the tile size, but for illustration purposes Figure 5 shows the outset instantiated for the 2D stencil example and 2×2 tiles. In this example, the outset includes only one empty tile origin at $(0, 0)$, far fewer than the number of empty tiles that the bounding box includes.

Geometrically, the outset construction can be viewed as shifting of the hyper-planes that define the lower bounds of the loops. For our 2D example, we shift the left vertical line and the two 45 degree lines, where the left vertical line and the top 45 degree line constitute the lower bound of k, and the bottom 45 degree line forms the lower bound for i. These lines are shifted out so that they will contain the origin of any tile which has a non-empty intersection with the iteration space, i.e., any tile that would contain a valid iteration point. Loops that scan the outset are post-processed and then used as the tile-loops. The tiled loops generated by scanning the outset is shown in Figure 6.

The outset has several important properties. It can be constructed without fixing the tile sizes, hence can be used for generating parameterized tiled loops. Second, it can be constructed very efficiently—in time and space linear in the number of loop bounds. In comparison, automatic construction of the bounding box is more expensive—we are not aware of any linear time algorithm that constructs a bounding box given the constraints that define an iteration space. Third, the outset can be used to decompose tiled loop generation into separate tile-loop and point-loop generation. Fourth, it can be used efficiently in cases when the tile sizes are fixed, parameterized or mixed, i.e., some are fixed and some are left as parameters. These properties lead to a single simple efficient algorithm for both parameterized as well as fixed tiled loop generation. The following sections discusses these properties in more detail.

3. Generating the Tile-Loops with Outset

In this section, we describe our method for generating the tile-loops. We first formally define the set that contains all the non-empty tile origins and then motivate a relaxation of this set which can be computed efficiently. We then reduce the problem of generating tile-loops to one of generating loops that scans the intersection of the outset polyhedron and a parameterized lattice. We describe a single method that can be used to generate tile-loops for both fixed as well as parameterized tile sizes.

Our input model is perfectly nested loops. Our techniques are applicable to cases where rectangular tiling is valid or can be made valid by any loop transformation. Many important applications contain loops of this kind. We assume that the input loop nest is appropriately transformed so that rectangular tiling is (now) valid.

3.1 The Outset and its Approximation

For correctness, tiled code should visit all the tiles that contain points in the original iteration space. To generate the tile loops separately from the point loops, we visit all of the tile origins within a polyhedron we call the outset. The *outset* includes all possible tile origins where the tile for that tile origin includes at least one point from the original iteration space.

The original loop is represented as a set of inequalities

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

where z is the iteration vector of size d, Q is a $m \times d$ matrix, $\vec{q}$ is a constant vector of size m, $\vec{p}$ is a vector of size n containing symbolic parameters for the iteration space, and B is a $m \times n$ matrix. The tiling is represented with the vector $\vec{s}$, where s_i indicates the size of the tile in dimension i, for $i = 1 \ldots d$.

We define the true outset polyhedron as the set of points in the original iteration space that, if they were tile origins, would define a tile that includes at least one point in the original iteration space. Formally, let $tile(\vec{x})$ specify the set of points that belong to the tile whose origin is $\vec{x}$,

$$tile(\vec{x}) = \{\vec{z} \mid \vec{x} \leq \vec{z} \leq \vec{x} + \vec{s'}\},$$

where $\vec{s'} = \vec{s} - \vec{1}$ with $\vec{1}$ being a size d vector containing all ones. The true outset is

$$P_{out} = \{\vec{x} \mid tile(\vec{x}) \cap P_{iter} \neq \emptyset\}.$$

P_{out} as defined above is an union of all tiles whose intersection with P_{iter} is non-empty. Computing this set explicitly is very expensive. So, we derive a reasonably tight approximation of P_{out} that is a single polyhedron and can be directly computed from the constraints of P_{iter}. We denote this approximation by $\widehat{P_{out}}$. As a comparison, one could also view the bounding box as a very loose approximation of P_{out}. $\widehat{P_{out}}$ can be computed in time and space linear in the number of constraints in P_{iter}. On the other hand, to the best of our knowledge, we are not aware of any linear time algorithm for computing the bounding box from constraints in P_{iter}. Henceforth we call $\widehat{P_{out}}$ the *outset*. The outset discussed in previous sections also refers to $\widehat{P_{out}}$.

We compute the outset, $\widehat{P_{out}}$, by shifting all the lower bounds of the original iteration space along the normal that faces out of the iteration space. The outset is defined as

$$\widehat{P_{out}} = \{\vec{x} \mid Q\vec{x} \geq (\vec{q} + B\vec{p}) - Q^{+}\vec{s'}\},$$

where Q^{+} is a $m \times d$ matrix defined as follows:

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

Note that the $\widehat{P_{out}}$ is defined using the constraint matrix, Q of the iteration space polyhedron. We can compute Q^{+} with a single pass over the entries of Q and hence in time linear on the number of constraints of P_{iter}. We now formally prove that $\widehat{P_{out}}$ contains all the non-empty tile origins.

THEOREM 1. $P_{out} \subseteq \widehat{P_{out}}$.

Proof:

If a point $\vec{x}$ is in P_{out}, then there exists a point $\vec{z}$ such that $\vec{z}$ is in P_{iter}, $\vec{z}$ is in $tile(\vec{x})$, and $\vec{z} = \vec{x} + \vec{i}$, where $\vec{0} \leq \vec{i} \leq \vec{s'}$. Since $\vec{z}$

is in P_{iter}, the following is true:

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

By substituting $\vec{z}$ with $\vec{x} + \vec{i}$, we derive the following:

$$Q\vec{x} + Q\vec{i} \geq (\vec{q} + B\vec{p}), \text{ for } \vec{0} \leq \vec{i} \leq \vec{s'}.$$

Due to the constraints on $\vec{i}$ and the fact that $Q^+ \geq Q$, it follows that $Q^+\vec{s'} \geq Qi$, and so the point $\vec{x}$ is also in $\widehat{P_{out}}$:

$$Q\vec{x} + Q^+\vec{s'} \geq (\vec{q} + B\vec{p}).$$

Thus, each point that is in P_{out} is also in $\widehat{P_{out}}$. ∎

Notice that though the tile sizes are not fixed and are included as parameters, the outset is still a polyhedron. This key property enables us to generate parameterized tile-loops, for now we can use all the theory and tools developed for generating loops that scan parameterized polyhedra.

3.2 Generating tile-loops

The tile-loops enumerate the tile origins. Two choices are available: (i) enumerate the tile origins as tile numbers in the tile space or (ii) enumerate the tile origins in the coordinates of the original iteration space. When the former is chosen, we need additional computations to map the tile numbers from the tile space to tile origins in the iteration space coordinates. Our method avoids this computation and generates loops that directly enumerate the tile origins in the original iteration space coordinates.

We can view the set of tile origins as the points in a lattice whose period is the tile sizes. We define the *tile origin lattice,* $\mathcal{L}(\vec{s})$, as the lattice whose period is given by the symbolic tile size vector $\vec{s}$. Note, that we are not fixing the tile sizes. Hence, $\mathcal{L}(\vec{s})$ is actually a *parameterized tile origin lattice.* We also do not require that the tile origin lattice start at any particular coordinate.

The outset contains all the non-empty tile origins and also other points which are not tile origins. The key insight is to generate loops that scans the whole of outset and modify them so that they skip the iterations that are not tile origins. Formally, we want to visit the points in the intersection of the outset and the tile origin lattice, i.e., $\widehat{P_{out}} \cap \mathcal{L}(\vec{s})$.

3.2.1 Striding the loops

Figure 7 shows an outset and a tile origin lattice for a 2×3 tiling. Let us call the loops that scan all the integer points in the outset as *outset-loops.* For a moment assume that the first iteration of every loop is aligned with a tile origin. Then we can skip the non-tile origins by just adding a *stride* to the loop variable with the corresponding tile size parameter. This simple post-processing of the loops that scans the outset gives us the loops that scans the intersection of outset and tile origin lattice. Note that the stride can be a fixed constant or a symbolic parameter. This allows us to use the same method for generating tile loops for both fixed and parameterized tile sizes.

3.2.2 Shifting Lower Bounds

We now address the issue of aligning the first iteration of the outset-loops to a tile origin. Figure 7 shows two non-tile origins that correspond to first iterations of the i loop. We need to shift the lower bound to an iteration that corresponds to the next tile origin. Let LB_i be the lower bound of a loop variable i. Note that LB_i could be a function of the outer loop indices and parameters. The required shift can be thought of as the difference between the value of LB_i and the next tile origin. This shift can be computed as $\left\lceil \frac{LB_i}{s_i} \right\rceil \times s_i$. We would like to emphasize that this shift can be generated for fixed as well as parameterized tile sizes. This yields a single method for both fixed and parameterized tiled loop nest generation.

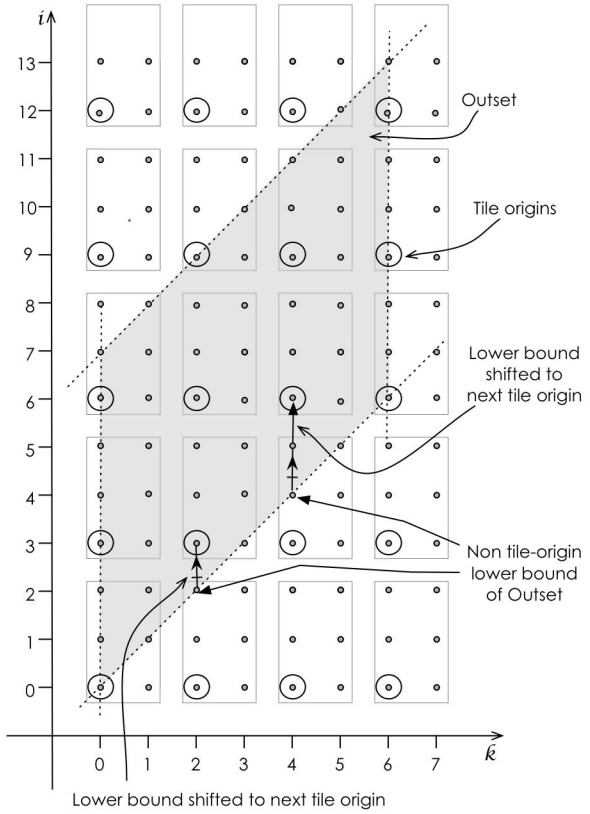

Figure 7. Intersection of a tile origin lattice for 2×3 tiles and the outset is shown. The original iteration space is omitted for ease of illustration. Note that the first iteration of the loops that scans the outset could be a non-tile origin. We need to shift this iteration to the next iteration that is tile origin.

The code previously presented in Figure 6 showing the parameterized tiled loops for the 2D stencil example (Figure 1) was generated using the scheme described above. Note how the skipping of the non-tile origins naturally translates into parametric strides of the loop variables. Also note how the lower bound shifts can be expressed as loop variable initializations.

3.2.3 Implementation

Our code generator takes as input the constraints that define P_{iter}. It constructs the outset ($\widehat{P_{out}}$), which is parameterized by the program and tile parameters. The outset-loops are generated using a standard loop generator for parameterized polyhedra. Thanks to our theory, all that is required to turn them into tile loops is a simple post-processing, actually pretty-printing, to add strides and lower bound shifts. These tile loops are then composed with the point-loops whose generation is described in the next section.

4. Generating the Point Loops

The point-loops make up a loop nest that enumerates all the points within a tile. To ensure that they scan points only in the original iteration space, their bounds are composed of tile bounds as well as iteration space bounds. When the point-loops are generated separately, the tile origin is not known.

Consider the triangular iteration space shown in Figure 8. Essentially, the intersection of a tile (without fixing the tile origin) and the iteration space is the set of points to be scanned by point-loops.

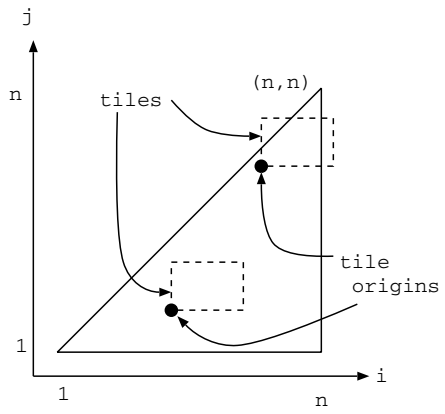

Figure 8. A triangular iteration space and tiles

To generate them, we can construct the intersection that is now parameterized by both program parameters and tile origin index. This approach does, however, increase the number of dimensions, which is a major factor at code generation time.

Since the tile bounds for rectangular tiling are simple, we can optimize the generation of the point loops. We first construct a loop nest that scans the original iteration space. Then, for each lower bound lb_i and upper bound ub_i, we add the tile lower bound, tlb_i, and upper bound, $tlb_i - s_i + 1$, (s_i is the tile size of i-th dimension) producing the lower bound $max(lb_i, tlb_i)$ and upper bound $min(ub_i, tlb_i - s_i + 1)$ of the point-loops. The point-loops for the example in Figure 8 are given below, with `iT` and `jT` representing the tile origin indices, and `Si` and `Sj` representing the sizes of the tiles along the i and j dimensions.

```
for i=max(1,iT) to min(N,iT-Si+1)
   for j=max(1,jT) to min(i,jT-Sj+1)
      body;
```

In addition, we can also generate simple point loops where iteration space bounds are not included. As shown in Figure 8, if a tile is a full tile, i.e., a subset of the iteration space, then the bounds for the original iteration space are not necessary. Such simple point loops are useful for the optimization described in Section 6.

5. Implementation and Experimental Results

We implement four different tiled loop generators: two for fixed tile sizes and two for parameterized tile sizes. The loop generator is available as open source software [28]. For fixed-size tiles, we implement the classic and decomposed methods. For the classic method, the constraints that represent the tiled iteration space are constructed from the original loop bounds and then fed to CLOOG to generate the tiled loops. For the decomposed method, we construct an outset with fixed tile sizes and use them to generate tile-loops and generate the point loops separately as discussed in the previous sections. For parameterized tiled code generation, we implement the parameterized decomposed method presented in this paper and the bounding box method. For the bounding box method, we assume that the bounding box is provided as an input. The bounding box is used in the place of outset to generate tile-loops and the parameterized point loops are generated as in the fixed methods except the tile sizes are now symbolic parameters for the point loops. For the parameterized decomposed method, we first generate the outset from the input loop bounds and use it to generate the tile-loops. We then generate the parameterized point loops and embed them in the tile-loops to get the final tiled loop nest.

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

Table 1. Benchmarks used for code quality evaluation.

The experiments compare the various loop generating techniques in terms of the quality of the generated tile code and the efficiency of the tiled loop generation. Both of these measures depend heavily on the underlying code generator, because the techniques presented in this paper enable the implementation of parameterized decomposed tiling to use any loop generator capable of generating loops that scan a polyhedron as a black box. For generating the loops that scan a polyhedron we use the CLOOG loop generator, which has been shown to quickly generate high quality loops [6]. However, it is possible to replace the CLOOG generator with a different code generator such as the Omega code generator [23].

5.1 Experimental Setup

To evaluate the quality of the generated code, we use linear algebra computation kernels from BLAS3 and a stencil computation, as listed in Table 1. The stencil computation has a 3D iteration space, and operates on two dimensions of data. It is necessary to skew the stencil computation before applying tiling. Column 3 in Table 1 indicates the loop depth of the original loop, and the number of loops that are tiled.

We ran the experiments on an Intel Core2 Duo processor running at 1.86 GHz with an L2 cache of size 2MB. The system is running SMP Linux. For compiling our tiled loop nests we used `g++` version 4.1.1. with the highest optimization level (`-O3`). The timings use gettimeofday().

5.2 Results

For each combination of benchmark and implemented tiled code generation method, we time an approximation of loop overhead, the total run of the tiled benchmark, and the time required to generate the tiled code.

Figure 9 shows the loop overhead for SSYRK (symmetric rank k update) as a percentage of the total loop execution time. We time the execution time of the tiled loop bounds with only a counter as the body and divide the measured execution time by the execution time for the loop with the full body including the loop counter. The loop overhead is only approximate, because in the loop with the full loop body some of the loop bound instructions can be scheduled with instructions from the body, therefore this measure is an upper bound on the loop overhead. The approximate loop overhead on average can be as high as 40%. Figure 10 shows the total execution time for the SSYRK as the tile sizes vary. Notice that as the tile sizes become large enough to result in improved performance of the overall loop, the approximate percentage of time spent on loop overhead increases.

Figures 10-13 show the total execution time for the various benchmarks as the tile size varies. The cache effect that occurs as the tile size better uses cache can most clearly be seen for STRMM, 3D Stencil, and SSYRK. In general, the quality of the generated tiled code is comparable. The outliers occur at smaller tile sizes, where the parameterized tiled code generator based on bounding box significantly increases the running time for all benchmarks.

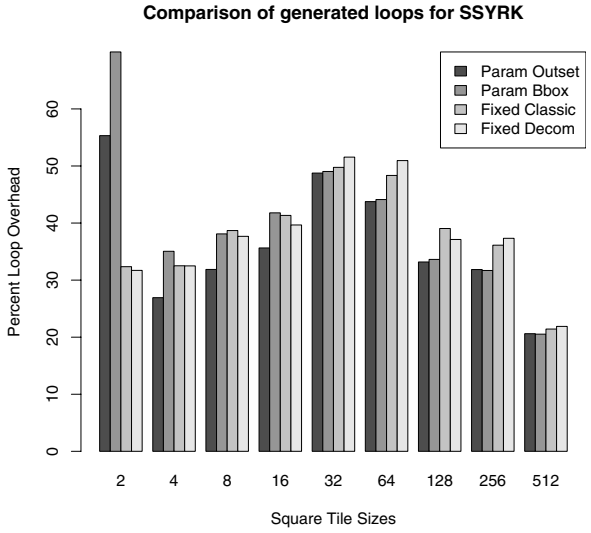

Figure 9. Percentage loop overhead =(counter / body and counter)$\times$100 of the SSYRK for matrices of size 3000×3000.

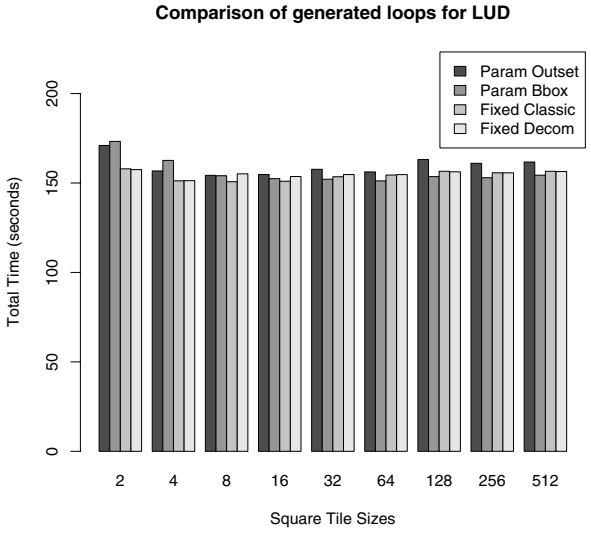

Figure 11. Total execution time for LUD on a matrix of size 3000×3000.

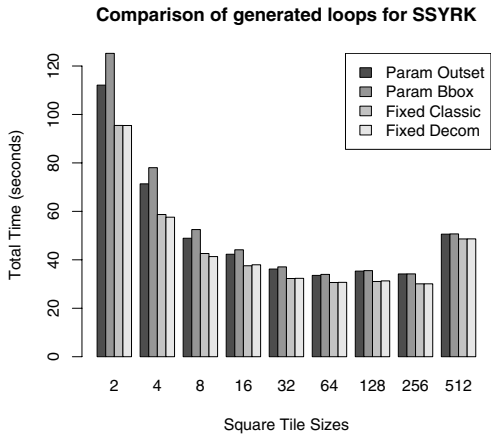

Figure 10. Total execution time for symmetric rank k update for matrices of size 3000×3000.

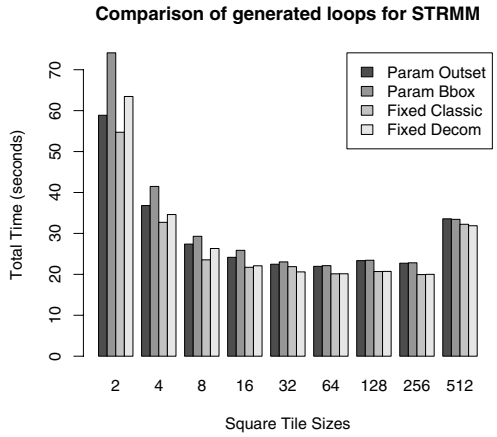

Figure 12. Total execution time for STRMM for matrices of size 3000×3000.

For cache tiling, the smaller tile sizes do not experience the best performance improvement; however, smaller tile sizes are critical for register tiling [13]. The parameterized decomposed method presented in this paper performs much better than bounding box at smaller tile sizes.

We also performed the same set of experiments on an AMD Opteron dual core processor running at 2.4 GHz with a cache of size 1MB, and obtained similar results as presented here.

The compilation time (in milliseconds) for the four tiled loop generation methods, viz., fixed classic, fixed decomposed, parameterized bounding box, and parameterized outset are shown in Table 2. The timings shown in the table are the average over five runs for each benchmark and each method. The timings include file IO. Further, the timings for the parameterized bounding box method do not include the time to generate the bounding box from the iteration space polyhedron. For the experiments it was given as user input. In a fully automated scenario, this additional time for generating the bounding box will add to the generation time of the bounding box method.

Overall, the cost of code generation for the three methods, viz., fixed decomposed, bounding box, and parameterized outset, falls within the range of 45 to 55 milliseconds. Hence they have very comparable generation efficiency (even when the time to generate the bounding box is not included). Though the fixed classic method seems to be significantly more efficient than the fixed decomposed method, as observed by Goumas et al. [9], it is expected to have scaling problems as the number of number of tiled loops increase.

In summary, the parameterized decomposed method presented in this paper generates code with performance comparable if not better than both fixed and parameterized tiled code generation methods. For parameterized tiled code generation, the parameterized decomposed method based on the outset is clearly better than the traditional bounding box method, especially for smaller

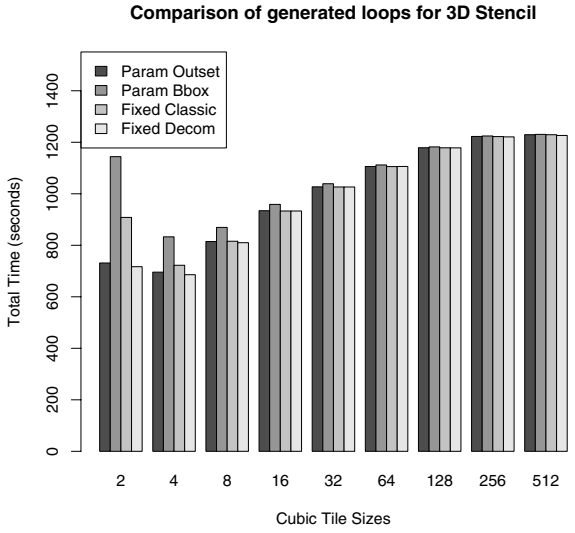

Figure 13. Total execution time for 3D Stencil on a 2D data grid of size 3000×3000 over 3000 time steps.

	LUD	SSYRK	STRMM	3D Stencil
fClassic	32.4	28.6	29.0	26.0
fDecom	55.2	51.0	50.4	45.0
pBbox	53.5	53.2	51.2	54.0
pOutset	52.0	53.8	52.1	54.1

Table 2. Tiled loop generation times (in milliseconds) of the four methods on the four benchmarks. The four methods fixed classic, fixed decomposed, parameterized bounding box, and parameterized outset are denoted by fClassic, fDecom, pBbox, and pOutset respectively.

tile sizes. The code generation time for all of the methods is comparable and quite small.

6. Finding Full Tiles Using the Inset

One possible source of loop overhead occurs within the loop bounds for each tile, which contain the bounds for the original iteration space as well as the tile so that no iterations outside of the original iteration space are executed. Ancourt and Irigoin [3] suggest that tiled code may be optimized by generating different code for full tiles versus partial tiles. Previous work [13] uses index set splitting to break the iteration space into full and partial tiles so that iteration bounds can be removed from the bounds for the full tiles. Other work [9] indicates that they differentiate between full and partial tiles, but details are not provided. Since distinguishing between full and partial tiles is important for register tiling and possibly hierarchical tiling, we present two possible approaches for doing just that. Both approaches are based on constructing the *inset* polyhedron such that any tile origins within the inset polyhedron P_{in} are tile origins for full tiles.

Distinguishing between full and partial tiles is applicable to all of the tiled code generation techniques discussed in Section 2. The inset can be computed as quickly as the outset, and it is possible to show that points are in the calculated inset if and only if they are possible tile origins for full tiles. Once the inset has been computed, it is possible to leverage existing code generators to generate the tile loops that traverse the inset executing only full tiles and the outset minus the inset executing partial tiles.

6.1 Algorithm for Computing Inset

As in Section 2, the original loop in question is represented as a set of inequalities

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

where z is the iteration vector of size d, Q is a $m \times d$ matrix, $\vec{q}$ is a constant vector of size m, $\vec{p}$ is a vector of size n containing symbolic parameters for the iteration space, and B is a $m \times n$ matrix. The vector $\vec{s}$ specifies the (hyper) rectangle tiling, with s_i indicating the tile size for the ith dimension of the iteration space.

We define the inset polyhedron P_{in} such that any tile origins that lie within the inset polyhedron are tile origins for full tiles. All the points in a tile satisfy an inequality constraint if and only if the extreme points for the tile satisfy the constraint. The extreme points of a (hyper) rectangle tile can be calculated as follows. Let $\vec{s'} = \vec{s} - \vec{1}$ and let $S' = diag(\vec{s} - \vec{1})$. Then S' times any binary vector of size d is an extreme point of the tile. Formally, the inset is

$$P_{in} = \{\vec{z} \mid \forall \vec{b} \in \{0,1\}^d, Q(\vec{z} + S'\vec{b}) \geq (\vec{q} + B\vec{p})\},$$

It is possible to compute the inset directly from the definition, but that would result in $m*2^d$ constraints, with many of them being redundant. Instead, we calculate a matrix Q^- from the Q matrix in the constraints for the original iteration space, such that

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

The algorithm for computing Q^- is $O(md)$ and results in m constraints for the inset,

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

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

THEOREM 2. $\widehat{P_{in}} = P_{in}$.

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

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

Note that the above inequality is true for all binary vectors $\vec{b}$. Each row represents 2^d constraints: one for each possible value of the binary vector $\vec{b}$. Since all of the entries in the S' matrix are non-negative, it is possible to select a particular binary vector for each row that results in the least possible value for each entry and therefore provides a tight bound for all the constraints represented by that row. Specifically that binary vector has entry b_j equal to one if and only if Q_{ij} is negative. Selecting the binary vector for each row, which results in the tightest bound is equivalent to calculating the matrix Q^-.

For all binary vectors $\vec{b}$, the following is true:

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

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

6.2 Code Generation Implementation

One property of an inset P_{in} is that $tile(z) \cap P_{iter} = tile(z)$ for all $z \in P_{in}$. In other words, constraints on the iteration space are

redundant for any tile whose origin is in the inset. By removing these unnecessary loop bounds in the point loops, we can possibly reduce the loop overhead further. One may perform this optimization by checking whether a tile origin belongs to the inset before executing point loops or by splitting the inset from the outset.

To use the check approach, code must be generated that determines if a particular iteration lies within the inset. The other approach is to split the inset from the outset. Consider the fact that $P_{in} \subseteq P_{out}$. We associate a statement X_1 with P_{in} and a statement X_2 with P_{out} and feed both polyhedra to a code generator. Now, if a loop nest scans both P_{out} and P_{in} without guards, then loops that scan the inset must include both statements. incorporating. and P_{in} without guards different statements, statements. Now, we know that iteration constraints are redundant whenever there are two statements in the loop since $P_{in} \subseteq P_{out}$. Therefore, we replace the loop bodies with statements X_1 and X_2 with the tile loops for full tiles, and we replace the loop bodies with statement X_2 only with tile loops for partial tiles. one statement,

This splitting scheme based on the union of inset and outset provides a way to enable a full versus partial tile optimization for parameterized tile code. Also, it is easy to incorporate this scheme using existing code generators. Note that many code generators have been designed and developed to remove guards by splitting the iteration space into disjoint regions associated to different sets of statements.

The tradeoff between splitting and inserting a check has not been fully explored. For register tiling, it would seem that checking each tile to determine if it is full clearly introduces too much overhead. However, splitting can result in significant blowup in code size, which can cause instruction cache problems. it reduces loop overhead without introducing another overhead checking is preferable in terms of code size.

7. Related Work

Ancourt and Irigoin proposed a technique [3] for scanning a single polyhedron, based on Fourier-Motzkin elimination over inequality constraints. Le Verge et al. [19, 20] proposed an algorithm that exploits the dual representation of polyhedra with vertices and rays in addition to constraints. The general code generation problem for affine control loops requires scanning *unions* of polyhedra. Kelly et al. [15] solved this by extending the Ancourt-Irigoin technique, and together with a number of sophisticated optimizations, developed the widely distributed Omega library [23]. The SUIF [30] tool includes a similar algorithm. Quillere et al. proposed a dual representation algorithm [25] for scanning the union of polyhedra, and this algorithm is implemented in the CLooG code generator [6] and its derivative Wloog is used in the WRaP-IT project.

Techniques for generating loops that scan polyhedra can also be used to generate code for fixed tile sizes, thanks to Irigoin and Triolet's' proof that the tiled iteration space is a polyhedron if the tile sizes are constants [12]. 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. Methods for generating code for non-unimodular transformations use techniques similar to ours, however they use fixed lattices and we use a parameterized lattice.

Our work is similar in scope to that of Goumas et al. [9], who decompose the generation into two subproblems, one to scan the tile origins, and the other to scan points within a tile, thus obtaining significant reduction of the worst case complexity. They proposed a technique to generate code for *fixed-sized, parallelogram* tiles. Their technique computes an approximation to the outset, similar to our $\widehat{P_{out}}$. Specifically, they compute the *image* of $\widehat{P_{out}}$ by the tiling transformation, H, and generate code to scan this image. Because of this, their code has ceiling and floor operations, and the loop body must compute an affine function of the loop indices to determine the tile origins. Their method can handle arbitrary parallelogram shaped tiles, and they also use a technique similar to our inset to optimize the code. Note however, that all their techniques are applicable only to fixed tile sizes.

Comparatively our algorithm handles parameterized tile sizes. The key insight is that we view the outset as a polyhedron with, other than the program parameters, n additional parameters, namely the tile sizes. This allows us to efficiently leverage most of the well developed tools, and our technique performs as well as, if not better than, all others, at no additional cost.

There are also a number of additional differences. Our algorithm generates tile loops whose indices always remain in the coordinate space of the original loop. This avoids floor and ceiling functions, and enables us to generate tile loops through a very simple post-processing: adjust the lower bounds, and introduce a stride corresponding to the tile size. Our method is restricted to transformations that can be expressed as a composition of a unimodular transformation, followed by a rectangular tiling (blocking).

There has been relatively little work for the case where tile sizes are symbolic parameters, except for the very simple case of *orthogonal* tiling: either rectangular loops tiled with rectangular tiles, or loops that can be easily transformed to this. For the more general case, the standard solution, as described in Xue's text [31] 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.

Amarasinghe and Lam [1, 2] 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. [10] 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.

Jiménez et al. [13] 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. The focus of their paper is applying index-set splitting to tiled code to traverse parts of the tile space that include only full tiles. Their approach involves less overhead in the loop nest that visits the full tiles; however, the resulting code experiences significant code expansion. We suggest two possible approaches for differentiating between full and partial tiles: either generate a check to determine if the tile being visited is a full tile, or associate two different loop bodies with the inset and outset and let any polyhedra scanning code generator generate the appropriate code. The trade-off between the overhead due to the check versus the cost due to code expansion that occurs using index-set splitting or loops that scan the union of polyhedra is unclear and an area for further study.

8. Conclusions

Iteration space tiling is an essential transformation for matching computation/communication characteristics of programs with loops to those of the resources (memory hierarchy as well as interprocessor communication). Parameterized tiled loop generation is the problem of producing tiled code where the tile sizes remain

symbolic parameters until link, or even run time. Previous solutions to this problem were based on either the bounding box, or on symbolic Fourier Motzkin elimination. The bounding box technique, although very simple, has poor code quality, especially for non-rectangular iteration spaces, where the overhead after tiling may be as high as 50%. The symbolic Fourier Motzkin elimination technique, has high complexity. We presented a single, simple technique for fixed and parameterized tiled loop generation that subsumes all previously known algorithms. Our experiments compared the generation efficiency and code quality of the parameterized technique with those for fixed tile sizes. They demonstrate that in terms of generation efficiency, the parameterized technique is as good as the state of the art for fixed tile sizes. The generated code is also as efficient as the tiled loops with fixed tile sizes. Thus the technique provides parameterized tiled loops for free!

Our ongoing work involves more detailed experimentation and incorporation of the generator in an autotuner for special application domains. We are currently working on using the concepts of inset and outset to generate multi-level parameterized tiled code. As a future work, we plan to extend the outset based technique to generate parameterized tiled loops for imperfect loop nests.

References

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

[2] 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.

[3] 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.

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

[5] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explict optimizations for stencil computations. In *Memory Systems Performance and Correctness*, 2006.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.

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

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

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

[28] TLOG: A parameterized tiled loop generator. Available at: `http://www.cs.colostate.edu/~ln/TLOG/`.

[29] 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.

[30] 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.

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