Multi-level Tiling: M for the Price of One

DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout
(kim,ln,drostron,svr,mstrout)@cs.colostate.edu
Computer Science Department
Colorado State University
Fort Collins, Colorado, USA

ABSTRACT
Tiling is a widely used loop transformation for exposing-/exploiting parallelism and data locality. High-performance implementations use multiple levels of tiling to exploit the hierarchy of parallelism and cache/register locality. Efficient generation of multi-level tiled code is essential for effective use of multi-level tiling. Parameterized tiled code, where tile sizes are not fixed but left as symbolic parameters can enable several dynamic and run-time optimizations. Previous solutions to multi-level tiled loop generation are limited to the case where tile sizes are fixed at compile time. We present an algorithm that can generate multi-level parameterized tiled loops at the same cost as generating single-level tiled loops. The efficiency of our method is demonstrated on several benchmarks. We also present a method—useful in register tiling—for separating partial and full tiles at any arbitrary level of tiling. The code generator we have implemented is available as an open source tool.

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

General Terms
Algorithms, Experimentation, Performance

Keywords
multi-level tiling, parameterized code generation, parallelism, data locality

1. INTRODUCTION
Achieving good performance on modern systems—such as multi-core clusters that have deep memory hierarchies and many parallel processing units—requires explicit structuring of the application to exploit parallelism and data locality. Models and tools that can aid programmers with this structuring will be at the heart of next generation performance/productivity tools.

Tiling [13, 30, 19, 33] is a loop transformation that matches program characteristics (locality, parallelism, etc.) to those of the execution environment (memory hierarchy, registers, number of processors, etc.). As noted by Carter et al. [7], it can be, and often is, used at multiple-levels to structure a high-performance implementation. For example, the highly tuned matrix multiplication implementation generated by ATLAS or PHiPAC [31, 6] uses two levels of tiling: one for caches and another for registers and instruction level parallelism (ILP). High performance implementations of stencil computations use (at least) two levels of tiling: one for parallelism and another for caches [27]. Furthermore, with the advent of multi-core processors, additional levels of tiling could provide the higher-granularity parallelism. Multi-level tiling has almost become a design pattern for high performance implementations [5, 8].

Currently, due to the lack of tool support, multi-level tiling is only used by experts who have good knowledge of loop transformations and optimizations. In particular, deriving multi-level tiled code by hand is very tedious and error prone and currently there is no tool support for it. The tiled code consists of the tiled loops and the transformed loop body. In fact, just deriving the bounds of multi-level tiled loops is a very difficult task involving error prone calculations. For the special case of rectangular loop nests there are some standard techniques available for deriving the multi-level tiled loops. However, several important applications such as numerical linear algebra routines (e.g., LU decomposition, triangular matrix product, symmetric rank updates from BLAS) and stencil computations (after applying skewing to enable tiling) have non-rectangular iteration spaces. Multi-level tiling has been shown to be very useful for them [14, 15, 27].

Effective use of tiling involves the selection and adaptation of the tile sizes. Parameterized tiled code refers to tiled code where the tile sizes are not (fixed) compile time constants but are left as symbolic parameters. It can enable run-time feedback and dynamic program adaptation. For example, run-time tile size adaptation has been successfully used to improve execution on shared cache processors [23] and also for adapting parallel programs to varying workloads [22]. Compilers need to generate code in which the number of processors can be set at run time [2]. Another important use of tile size adaptation is in the context of utility computing, where programs are expected to be mobile—migrate and adapt to a new set of resources [9]. Such adaptations
with respect to the number of processors and memory characteristics can be directly mapped to tile size adaptations. Iterative compilers [17, 18] and “autotuners” such as ATLAS [31] and SPIRAL [25], can use parameterized tiled codes to avoid the generation of codes for each tile size that is explored.

We are aware of only one solution that can generate arbitrary levels of multi-level tiled code for general polyhedral iteration spaces [15]. Their technique is limited to the case when tile sizes are fixed at compile (tiled loop generation) time, which is a severe limitation in the situations discussed above. We propose a technique for generating multi-level, parameterized tiled loops. Our technique is general enough to generate parameterized or fixed or even mixed (tile sizes fixed at some levels and left as parameters at other levels) multi-level tiled loops. The following are the contributions of our paper.

- We propose a technique for generating multi-level tiled loops where the tile sizes can be fixed (constants) or symbolic parameters or mixed. Our technique provides multiple-levels of tiling at the same cost of generating tiled loops for a single level of tiling.
- We propose a novel formalization of the classic tiling transformation [13, 33] to multiple levels.
- We propose a method for separating partial and full tiles at any arbitrary level, without fixing the tile sizes. We have implemented all the proposed code generation techniques and the tool is available open source [12].
- We perform extensive evaluation of both the generation efficiency and quality of the generated code on benchmark routines form BLAS, LUD, and stencil computations.

The rest of the paper is structured as follows. In the next section, we introduce the background concepts used in generating single-level tiled loops. In Section 3, we present two different formalizations of the multi-level tiling and in Section 4, we discuss techniques for separating partial and full tiles. In Section 5, we present the multi-level tiled loop generation algorithm. In Section 6, we describe the implementation of our code generator and present the experimental results. In Section 7, we discuss related work and in Section 8, we conclude with some pointers to future work.

2. BACKGROUND

We first describe the structure of single-level tiled loops and develop an intuition for the concepts involved in generating them. We then introduce the notation that we will use in the later sections.

Consider the stencil computation shown in Figure 1. Its iteration space is a 2D parallelogram as shown in Figure 3. Also shown in Figure 3 is a 2 × 2 tiling of the iteration space. Observe that there are three types of tiles: full—which are completely contained in the iteration space, partial—which are not completely contained but have a 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. Tiled loop generation involves the production of a set of loops that scans (i.e., visits in lexicographic order) each integer point in the original iteration space based on the tiling transformation. Each “visit” of a tile origin is itself the scan of all points in the tile. For the example in Figure 1, we can view the four loops that scan the tiled iteration space as two sets of two loops each: the first set enumerates the tile origins and the next set visits 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. A simple technique for tiled loop generation is to allow the tile-loops to scan the tile-origins in the bounding box of the iteration space. However, as shown in 3, the bounding box can contain many empty tile origins and make the tile-loops inefficient.

```
...)
S1(k,i);
```

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

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

Figure 2: Parameterized tiled loops generated using outset. The variables kTLB and S1LB are used to shift the first iteration of the loop so that it is a tile origin.

Two sets that are of interest are (i) the set that contains all the non-empty tile origins and (ii) the set that contains the origins of all full-tiles. The bounding box of the iteration space is an example of the former and is often used as a simple solution to generate the tile-loops. In our earlier work [28], we introduced two constructs, viz., outset and inset. The outset contains all the non-empty tile origins and the inset contains all the full-tile origins. The method proposed in this paper for multi-level tiled loop generation uses these sets. To make this paper self contained we have included a description of these sets in the following.

Our input model is perfectly nested loops. Our techniques are applicable to cases where rectangular tiling is valid or can be made valid by an appropriate preprocessing transformation (e.g., skewing). We assume that this has already been done. The input loop of depth d is represented as a set of m inequalities

\[ P_{ter} = \{ z \mid Qz \geq (q + Bp) \} \]

where \( z \) is the iteration vector of size \( d \), \( Q \) is a \( m \times d \) matrix, \( q \) is a constant vector of size \( m \), \( p \) is a vector of size \( n \) containing symbolic parameters for the iteration space, and \( B \) is a \( m \times n \) matrix. The tile sizes are represented by the vector \( s \); we use \( \tilde{s} \) to denote \( s - 1 \).

2.1 Generating tile-loops using outset

We use the outset to generate loops that iterate over the tile origins. The outset, \( P_{out} \), is constructed by shifting all the hyperplanes that define lower bounds of the original loop
Figure 3: A $2 \times 2$ rectangular tiling of the 2D Stencil iteration space from 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.

The outset contains all non-empty tile origins and also other points which are not tile origins. The idea is to generate loops that scan the whole outset, and post process them so that they skip the iterations that are not tile origins. This skipping is done by (i) a shift in the first iteration of a loop so that the iterations start from a tile origin and (ii) adding the appropriate tile size as a stride to the loop. The loops thus generated for the 2D Stencil example are shown in Figure 2—the statements immediately preceding the loop headers compute the shifts each tile-loop.

### 2.2 Splitting partial/full tiles using inset

Point-loops are constructed with bounds from the tiles and the iteration space. Full-tiles correspond to (hyper-) rectangles that are fully contained within the iteration space. Hence, for these tiles the iteration space bounds are redundant. The ability to identify full tiles enables the generation of two sets of loops: one with just the tile bounds for full tiles and another with both tile and iteration space bounds for the partial tiles leading to better code quality. Further, the separation of partial and full tiles is necessary for register tiling [14].

All previous methods [3, 10, 14] for separating partial and full tiles are applicable only when the tile sizes are fixed. Our method separates partial and full tiles when the tile sizes are left as symbolic parameters. Further, we allow separation of partial and full tiles at any arbitrary level of the tiling. The key construct that enables our method is the *inset* which is also a polyhedron parameterized by the tile sizes. We define the inset polyhedron $P_{in}$ such that any tile origins that lie within the inset polyhedron are tile origins for full tiles. We have shown that [28] the inset $P_{in}$ can be constructed as follows

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

where $Q_{ij} = \begin{cases} Q_{ij}, & \text{if } Q_{ij} \geq 0 \\ 0, & \text{if } Q_{ij} < 0 \end{cases}$

Recall that $\vec{s} = \vec{s} - \vec{1}$. Note that the $P_{out}$ 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 have shown [28] that $P_{out}$ is a fairly tight approximation of the outset. Notice that the outset is also a polyhedron, but with the tile sizes as additional parameters. This key property enables us to generate parameterized tile-loops, by building on all the theory and tools developed for generating loops that scan parameterized polyhedra.

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*, $\text{Lattice}(\vec{s})$, 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, $\text{Lattice}(\vec{s})$ is actually a parameterized *tile origin lattice*. We also do not require that the tile origin lattice start at any particular coordinate.

We start with the classic definition of single-level rectangular tiling [33]. Given an iteration space $P_{iter}$ and a vector $\vec{s}$ of fixed tile sizes, the tiled iteration space is given by

$$P_{tiled}^1 = \{(\vec{t}, \vec{z}) | \vec{s} \circ \vec{t} \leq \vec{z} - \vec{d} \leq \vec{s} \circ \vec{t} + \vec{s} - \vec{1}, \vec{z} \in P_{iter} \}$$

where $\vec{d}$ is an offset and the operator $\circ$ denotes componentwise multiplication of vectors. The tiles are enumerated by $\vec{t}$ and the points within a tile are represented by $\vec{z}$. The tiled iteration space denoted by $P_{tiled}$ is a polyhedron (as the tile sizes are fixed). Generating the tiled loop nest is now reduced to generating loops that scan the polyhedron $P_{tiled}^1$. 

3. MULTI-LEVEL TILING

As mentioned in the previous section the input is a perfect loop nest, and it is appropriately transformed so that rectangular tiling is valid. In this section, we describe two multi-level tiling approaches. The first one is an extension of the classic tiling transformation [33] to multiple levels and is restricted to the case of where the tile sizes are fixed. The second one is based on the concept of the outset (introduced in the previous section) and can be used when the tile sizes are symbolic parameters or fixed constants or mixed.

3.1 Multi-level tiling for fixed tile sizes

We start with the classic definition of single-level rectangular tiling [33]. Given an iteration space $P_{iter}$ and a vector $\vec{s}$ of fixed tile sizes, the tiled iteration space is given by

$$P_{tiled}^1 = \{(\vec{t}, \vec{z}) | \vec{s} \circ \vec{t} \leq \vec{z} - \vec{d} \leq \vec{s} \circ \vec{t} + \vec{s} - \vec{1}, \vec{z} \in P_{iter} \}$$

where $\vec{d}$ is an offset and the operator $\circ$ denotes componentwise multiplication of vectors. The tiles are enumerated by $\vec{t}$ and the points within a tile are represented by $\vec{z}$. The tiled iteration space denoted by $P_{tiled}$ is a polyhedron (as the tile sizes are fixed). Generating the tiled loop nest is now reduced to generating loops that scan the polyhedron $P_{tiled}^1$. 

The outset contains all non-empty tile origins and also other points which are not tile origins. The idea is to generate loops that scan the whole outset, and post process them so that they skip the iterations that are not tile origins. This skipping is done by (i) a shift in the first iteration of a loop so that the iterations start from a tile origin and (ii) adding the appropriate tile size as a stride to the loop. The loops thus generated for the 2D Stencil example are shown in Figure 2—the statements immediately preceding the loop headers compute the shifts each tile-loop.
There are standard tools such as Omega [16] and CLOOG [4] which can be used to generate such loops. Note that \( P^m_{\text{tile}} \) is a polyhedron only when the tile sizes are fixed and hence the approach is not applicable when the tile sizes are symbolic parameters.

We can extend the definition to multiple levels of tiling as follows. Given an iteration space \( P_{\text{iter}} \) and a list of tile size vector \( \vec{s}_1, \ldots, \vec{s}_m \), a multi-level tiling can be described in a similar way.

\[
P^m_{\text{tile}} = \{(\vec{t}_1, \ldots, \vec{t}_m, \vec{z}) \mid \forall i = 1, \ldots, m-1: \\
\vec{s}_i \circ \vec{t}_i \leq t_i + 1 - a_i + 1 \leq \vec{s}_i \circ \vec{t}_i + \vec{s}_i - \vec{1}, \\
\vec{s}_m \circ \vec{t}_m \leq \vec{z} - a_m \leq \vec{s}_m \circ \vec{t}_m + \vec{s}_m - \vec{1}, \vec{z} \in P_{\text{iter}} \}
\]

where \( a_i \) is an offset at the appropriate level. All tile sizes are integer constants. Also, note that actual tile sizes are a product of all inner tile sizes because tiling at level \( k \) is a tiling on the \( (k + 1) \) tiled space, not the original iteration space. Although this formulation is a direct extension of Xue’s definition of single level tiling [33], to the best of our knowledge, this is first formalization and presentation of it—other formulations [15] of multi-level tiling are based on the strip-mine and interchange view of tiling. Now given the fact that this set \( P^m_{\text{tile}} \) is a polyhedron, the scanning loops can be easily generated by existing tools, such as Omega test and CLOOG. Our generator for this method uses CLOOG.

### 3.2 Multi-level tiling using the outset

Another view of tiled loop generation is based on the outset method as described in the previous section, where the coordinates of the tile origins are obtained by intersecting the outset \( P_{\text{out}} \) with a parameterized lattice \( \text{Lattice}(\vec{s}) \). This method does not require the tile sizes to be fixed. Multi-level tiling in this method can be viewed geometrically as shown in Figure 4. We start with the first level of tiling of the iteration space and each first level tiles are further tiled to achieve the second level of tiling. In Figure 4, the first level of tiling uses \( 4 \times 4 \) tiles and the second level uses \( 2 \times 2 \) tiles. The geometric view not only aids visualization but also gives a mathematical view of the multi-level tiling: the tile origins at a given level \( k \) of tiling can be viewed as the intersection of the tiles at the previous \( (k - 1) \) level and the lattice parameterized by the tile sizes of level \( k \).

To exploit the geometric view for tiled loop generation we need to handle one important issue. Consider the outer level of tiling shown in Figure 4. There are three partial outer-tiles and one full outer-tile. When we apply another inner-level of tiling the outer-tiles become the iteration space for them, and we need to handle the different shapes of the partial outer-tiles. We handle this by (over) approximating the partial outer-tiles by full tiles. Such an approximation allows a uniform treatment of the further levels of tiling. The 2-level tiled loop nest generated using this method for the example is shown in Figure 5. Note that the tile-loops at the second level treat partial tiles as full tiles. The general structure of the multi-level tiled loops generated using this method is shown in Figure 6. The outermost tile-loops are generated using the outset and all inner-level tile-loops are generated using the bounds of a full-tile, referred as Box-tile-loops. The innermost loop nest is the point-loops which have the both the tile bounds and the iteration space bounds. We expect the execution time overhead due to the approximation of inner-level partial tiles by full tiles to be insignificant.

![Figure 5: A loop nest corresponding to the multi-level tiling in Figure 4](image)

![Figure 6: Structure of multi-level tiled loops generated with the outset method when partial and full tiles are not separated.](image)

Our expectation is confirmed by our experimental results as discussed in Section 6.2.

Multi-level tiling based on outset can be formalized as follows. Given an iteration space \( P_{\text{iter}} \) and a list of tile size vector \( \vec{s}_1, \ldots, \vec{s}_m \), the tiled iteration space can be expressed as follows:

\[
P^m_{\text{tile}} = \{(\vec{t}_1, \ldots, \vec{t}_m, \vec{z}) \mid \forall i = 2, \ldots, m: \\
\vec{t}_i \in P_{\text{out}} \cap \text{Lattice}(\vec{s}_1, \vec{1}), \\
\vec{z} \in P_{\text{iter}} \cap \text{tile}(\vec{t}_1, \vec{s}_1) \cap \cdots \cap \text{tile}(\vec{t}_m, \vec{s}_m), \\
\vec{t}_i \in \text{tile}(\vec{t}_1, \vec{s}_1) \cap \cdots \cap \text{tile}(\vec{t}_{i-1}, \vec{s}_{i-1}) \cap \\
\text{Lattice}(\vec{s}_1, \vec{t}_{i-1}) \}
\]

where \( \text{Lattice}(\vec{s}_1, \vec{t}_{i-1}) \) is the set of points generated by \( \vec{s}_1 \circ \vec{z} + \vec{t}_{i-1} \) for any integer vector \( \vec{z} \) and \( \vec{s}_1 \) can be a vector of either symbolic tile size parameters, constants, or mixture of both. Note that the offset of the lattice depends on the origin of each tile at the previous level. Given a tile, \( \text{tile}(\vec{t}_i, \vec{s}_i) \), the first tile at level \( (i + 1) \) that is contained in \( \text{tile}(\vec{t}_i, \vec{s}_i) \) must be \( \text{tile}(\vec{t}_i, \vec{s}_{i+1}) \) because \( \vec{t}_i \) is still the lexicographical minimum of \( (\vec{t}_i, \vec{s}_i) \). Otherwise, some points in the iteration space will not be scanned. Correctness of this formulation follows directly from the fact that \( P_{\text{out}} \) contains origins of the tiles whose union is super-set of \( P_{\text{iter}} \). Further, by including in the formulation, the constraints that define \( P_{\text{iter}} \) we guarantee that only valid iteration points in the tiles are enumerated. Also note that the formulation does not impose the restriction that outer tile sizes are multiples of inner tile sizes.

In most practical cases, tile sizes \( \vec{s}_i \) are component-wise multiples of \( \vec{s}_{i-1} \) for all \( i = 1, \ldots, m - 1 \). The constraints of the tiled iteration space in (2) for this case can be simplified to:

\[
P^m_{\text{tile}} = \{(\vec{t}_1, \ldots, \vec{t}_m, \vec{z}) \mid \forall i = 2, \ldots, m: \\
\vec{t}_1 \in P_{\text{out}} \cap \text{Lattice}(\vec{s}_1, \vec{1}), \vec{z} \in P_{\text{iter}} \cap \text{tile}(\vec{t}_m, \vec{s}_m), \\
\vec{t}_i \in \text{tile}(\vec{t}_{i-1}, \vec{s}_{i-1}) \cap \text{Lattice}(\vec{s}_1, \vec{t}_{i-1}) \}
\]
4. SEPARATING PARTIAL & FULL TILES

As discussed earlier, separation of partial and full tiles has several applications. In this section, we discuss how the inset (introduced in Section 2.2) is used for separation. Separation at any level \( k \) implies that the further tilings (for levels \( k + 1 \ldots m \)) are performed only on full tiles of level \( k \). The partial tiles of level \( k \) are not further tiled. Consider the number of full and partial outer-tiles in Figure 4. There is one full outer-tile and three partial outer-tiles. If we separate full tiles from partial tiles at the outer level of tiling, then there are only four full inner-tiles, since only the full outer-tiles are tiled further. However, we can see that there are 10 full inner-level tiles in the iteration space. By separating the partial and full tiles at the inner-level (and not at the outer-level) we can actually recognize all the 10 inner-level tiles as full. However, separation at the inner-level leads to more inner-level full tiles but also results in enumeration of more empty inner-tiles. Hence, there is a trade-off between more inner-level tiles versus enumeration of empty tile origins. Further, we can also apply splitting multiple times if needed.

The general structure of such a multi-level tiled loop nest with separation of partial and full tiles at an arbitrary level \( k \) is shown on Figure 7. Note that the partial tiles at level \( k \) are not further tiled and they execute the standard point-loops. On the other hand, the full tiles of level \( k \) are further tiled and their body contain a special form of point-loops called box-point-loops. These box-point-loops are the loops in which the iteration space bounds are omitted.

To recall, the inset \( P_{m} \) represents the set which contains all the full-tile origins. Let us denote by \( P_{m}(\vec{s}_{k}) \) the inset computed using the tile sizes of level \( k \) and the iteration space \( P_{iter} \). Now we can check at any level \( l \) whether a tile origin represents a full tile or not by checking whether it belongs to \( P_{m}(\vec{s}_{l}) \) or not. This is the key idea underlying our separation algorithm. For any user specified level \( k \) of separation we generate the outset \( P_{m}(\vec{s}_{k}) \) and use it to test whether a tile is full or partial. This test corresponds to the \( \text{FULL}(L_{k}-\text{tile}) \) test in Figure 7.

When the separation happens at level \( k \), the set of points in the full tiles at level \( k \) can be described as follows:

\[
P_{\text{full}}^{k} = \{(t_{1}, \ldots, t_{m-1}, t_{m+1}) | \forall i = 2, \ldots, m+1 : \}
\]
\[
\vec{t}_{i} \in P_{out} \cap \text{Lattice}(\vec{s}_{i}, \vec{o}), \vec{t}_{k} \in P_{in}^{k},
\]
\[
\vec{t}_{k} \in \text{tile}(t_{k-1}, s_{k-1}) \cap \text{Lattice}(\vec{s}_{k}, t_{k-1}) \}
\]

where \( s_{m+1} = 1 \). The set of points in the partial tiles can be described as follows:

\[
P_{\text{partial}}^{k} = \{(t_{1}, \ldots, t_{k}, \vec{z}) | \forall i = 2, \ldots, k : \}
\]
\[
\vec{t}_{i} \in P_{out} \cap \text{Lattice}(\vec{s}_{i}, \vec{o}), \vec{t}_{k} \notin P_{in}^{k},
\]
\[
\vec{z} \in P_{iter} \cap \text{tile}(\vec{t}_{k}, \vec{s}_{k}),
\]
\[
\vec{t}_{k} \in \text{tile}(t_{k-1}, s_{k-1}) \cap \text{Lattice}(\vec{s}_{k}, t_{k-1}) \}
\]

Different levels of separation may be preferred, based on the context in which separation is used. For example, for a 2-level tiling in the context of caches and registers an inner-level of tiling might be preferred. An example of this is...
Algorithm 1 An algorithm for generating multi-level tiled loops based on outset approach

INPUT: $P_{iter}$: Iteration space matrix, tileSizes[$1...m$]: tile size (integer or symbolic parameter) vector, tileIndexes[$1...m$]: tile index name vector, split: a boolean value whether full and partial tiles are split, splitLevel: level at which full and partial tiles are split.

BEGIN
Matrix outset, inset;
VectorOfString pLoops, comLoops;

// Compute P_out
1: outset = computeOutset($P_{iter}$, tileSizes[1], tileIndexes[1]);

// Scan P_{iter}, add tile bounds with appropriate level
2: If (split == true)
3: pLoops = generatePointLoops($P_{iter}$, tileSizes[m], tileIndexes[m]);
4: else
5: pLoops = generatePointLoops($P_{iter}$, tileSizes[splitLevel], tileIndexes[splitLevel]);

// Compute P_in when split is greater than 0
6: If (split == true)
7: inset = computeInset($P_{iter}$, tileSizes[splitLevel], tileIndexes[splitLevel]);

// Combine point-loop, box-loop and guard for split
8: comLoops = combine(pLoops, tileSizes[1...m], tileIndexes[1...m], splitLevel, inset);

// Generate loops that scans outset while printing
// comLoops instead of point-loop
9: printScanningLoops(outset, comLoops);
END

shown in our experiments on cache and register tiling.

5. THE LOOP GENERATION ALGORITHM

Now we present our algorithm for generating multi-level tiled loop nests with parameterized, fixed, or mixed tile sizes. It is given in Algorithm 1 and its input is the original iteration space, number of levels of tiling, whether the loops are to be split for partial vs. full tile separation, and if so, what is the level at which this split needs to be performed. The output of the algorithm is the multi-level tiled loop nest.

We illustrate the steps of the algorithm on the 2D Stencil example. We seek to generate a 2-level tiled loop nest where full and partial tiles are split at the first level. We first compute an outset of the iteration space with the outer-tile sizes. Then, we generate the point loops whose bounds consists of iteration space bounds and the surrounding tile bounds. The split level determines the tile bounds used in the point-loops generation as shown in lines 2-5 of the algorithm. These loops are generated by a call to CLOOG. Next, we compute the inset of iteration space with respect to the split level (here, first) tile sizes and indices as shown in lines 6-7. The bounds of the inset are done via simple pretty-printing using the tile indices and sizes. The complete multi-level tiled loop nest with separation of partial and full tiles is then generated we can generate all the loops. The construction of the outset can be done through a simple pretty printing using the already generated point loops. The construction of the outset is done via a call to CLOOG. The body of the loop is by $S_1$.

\[
P_{in} = \{(t_k, t_i) | 1 \leq t_k; t_i + s_k - 1 \leq N_k; t_k + s_i \leq t_i + s_i - 1 \leq t_k + N_i\} \quad (4)
\]

where $s_k$ and $s_i$ are symbolic tile size parameters along $k$ and $i$ dimensions, respectively. The guard for splitting partial and full tiles is obtained directly from the inset. The complete multi-level tiled loop nest for the 2D Stencil example with separation at the first level is shown in Figure 8. At line 9 we see that the guard is a direct translation from the inset in (4).

Once the point-loops and inset based on a split level are generated we can generate all the loops. The construction of the inner-level tile-loops, the guards and the box-tile-loops can be done through a simple pretty printing using the appropriate bounds. Combining these with the already generated the point-loops (as shown in line 8) we get all the loops except the outer-most tile-loops. This is generated by a call to CLOOG to generate loops that scan the outset and post-processing it to add lower bound shifts and strides. The resulting tile-loops are shown in lines 2-5 of Figure 8. Finally we compose these outermost tile-loops to obtain the complete tiled loop nest with separation of partial and full tiles.

5.1 Complexity & scalability of the algorithm

Let us first consider the case where no full vs. partial tile separation is performed. At the high level, the key steps are computing the outset to generate the outermost tile-loops and constructing all the box-tile-loops and constructing the point-loops. The construction of the outset can be done in time linear on the number of bounds on the original loop nest. Further, the construction of the box-tile loops is a simple pretty-printing using the tile indices and sizes. The construction of the point-loops and the tile-loops using the outset are done via CLOOG. The complexity of each of these calls to CLOOG is exponential in the number of bounds of the original loop nest and not the number of bounds in the tiled loop nest. Hence, the entire multi-level tiled loop nest
construction involves two calls to an exponential function and a couple of functions that are linear on the number of bounds on the original loop nest and the number levels of tiling. The key point to note is that the number of calls to the exponential function do not depend on the number of levels. In fact, for any arbitrary number of levels of tiling exactly two calls are made to the exponential-time function. Now, if we consider separation of partial and full tiles, all that is required is the computation of the inset (which can be done in linear time) and the pretty printing of it as a guard. On the whole, the time complexity of our algorithm is determined by the time taken by the two calls to cloog, and is constant with respect to the number of levels of tiling. The experimental results in Section 6.1 confirm this, and also further validate our claim that we can generate multi-level tiled loops at the cost of a single-level tiled loops.

In contrast the time for the classic method depends on the number $m$ of multi-level tiling. For an original loop nest of depth $d$, the number of dimensions and constraints increase by $d$ and $2d$, respectively, as the level of tiling increase (assuming all the dimensions are tiled). This results in an exponential space/time complexity which grows with the number of levels of tiling. The experimental results in Section 6.1 show how this exponential growth with respect to number of levels renders the technique inapplicable beyond two levels of tiling. The multi-level tiled loop generation method proposed by Jiménez et al. [15] has an exponential time complexity at each level of tiling, and this grows linearly with the number of levels of tiling.

6. EXPERIMENTAL VALIDATION

We implement three different multi-level tiled loop generators. The first generator is for the case when the tile sizes are fixed, and uses the classic tiling method discussed in Section 3.1. The second generator is capable of generating tiled code with the tile sizes that are fixed or parameterized or mixed and is based on the method discussed in Section 3.2. The third generator implements the additional feature of splitting (or separating) partial and full tiles at some user specified level. The generators are implemented in C++. The Cloog [4] loop generator is used internally to generate the point-loops and the loops that scan the outset. Our technique is independent of the internal code generator and for example, we could use Omega [16] instead of Cloog. We chose Cloog for its robustness across several benchmarks and its code generation speed (up to $4 \times$ faster than Omega [4]).

To evaluate the generation efficiency and the quality of the generated code we conduct three sets of experiments. The benchmarks used for the experiments are given in Table 1. The benchmarks 2D Stencil and 3D Stencil correspond to a Gauss-Seidel style stencil where a 1D array (or 2D array resp.) is updated over a time step loop. For these two benchmarks, we first applied skewing to make rectangular tiling valid and then used the skewed iteration space as input to our generator. The skewing makes the iteration space non-rectangular. The benchmark LUD is LU decomposition without pivoting. The benchmarks SSYRK and STRMM are routines from BLAS3 and correspond to symmetric rank $k$ update and the triangular matrix product computations, respectively. The loop nest depth of the benchmarks is shown in the third column of Table 1 and for the experiments, all the loops are tiled at all the levels for all the benchmarks. The three sets of experiments we conduct are aimed at evaluating the (i) the generation efficiency of loop generators, (ii) the cost of parameterization, i.e., what is the execution time cost for not fixing the tile sizes and leaving them as parameters, and (iii) the effect of the level at which partial and full tiles are separated. The following sections discuss each of these experiments.

6.1 Generation efficiency

We evaluate two aspects of the generation efficiency. First, we evaluate how our method scales with respect to the number of levels of tiling. Second, we compare the generation times for the parameterized and the fixed method. The second comparison also evaluates the overhead due to the over-approximation of the inner-level partial tiles by full tiles (cf. Section 3.2). All the generation efficiency experiments were run on an Intel Core2 Duo processor running at 1.86 GHz with an L2 cache of size 2 MB. We used g++ 4.1.1. with -O3 optimization level to compile our loop generators. The timings use gettimeofday(). Our code generator supports arbitrary (hyper-)rectangular tiles. For ease of experimentation we have used square tile sizes.

The generation times for the five benchmarks, 2D Stencil, LUD, SSYRK, 3D Stencil, and STRMM are shown in Figures 9, 10, 11, 12, and 13. The $x$-axis represents the number of levels of tiling and the $y$-axis represents the generation time (including file IO) in milliseconds. The generation time

![Figure 9: Generation time for multi-level tiling of 2D Stencil.](image-url)

<table>
<thead>
<tr>
<th>Description</th>
<th>Loop depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Stencil</td>
<td>Gauss-Seidel Style 2D stencil computation</td>
</tr>
<tr>
<td>LUD</td>
<td>LU decomposition of a matrix without pivoting</td>
</tr>
<tr>
<td>SSYRK</td>
<td>Triangular matrix multiplication</td>
</tr>
<tr>
<td>STRMM</td>
<td>Symmetric Rank $k$ Update</td>
</tr>
<tr>
<td>3D Stencil</td>
<td>Gauss-Seidel Style 3D stencil computation</td>
</tr>
</tbody>
</table>

Table 1: Benchmarks used for evaluating generation efficiency and code quality.
Figure 10: Generation time for multi-level tiling of LU decomposition.

Figure 11: Generation time for multi-level tiling of symmetric rank $k$ update (SSYRK).

Figure 12: Generation time for multi-level tiling of 3D Stencil.

Figure 13: Generation time for multi-level tiling of triangular matrix multiplication (STRMM).

Figure 14: Generation time for multi-level tiling of classic method. The $x$-axis of the graph is the number of loops in the tiled loop nest. The $y$-axis is the code generation time in seconds.

labeled No Split refers to the case where there is no-splitting of partial and full tiles and the other two – SplitLevel=1 and SplitLevel=Innermost – represent the generation where the splitting is done at level 1 (outermost) and at the innermost level, respectively. Note that the case of a single level of tiling with no splitting corresponds to the experiments from our previous work [28] on parameterized single level tiled loop generation. The main observation from the graphs is that the generation time is fairly flat as the number of tiling levels increase. Almost all the generation times are within the range of 40 to 60 milliseconds. This experimentally confirms our claim that our technique provides a method that can generate multi-level tiled loops at the price of a single-level tiled loop nest. Further, the graphs also show that splitting does not introduce any additional cost.

The generation times for the classic method for fixed tile sizes is shown in Figure 14. Note that the $x$-axis shows the number of loops in the tiled loop nest and not the number of levels tiled. For example, when a 3D loop nest is tiled...
two levels we will have 9 loops on the tiled loop nest. We show the number of loops in the tiled loop nest, because it is a finer granularity than the number of levels of tiling and shows clearly the exponential (w.r.t. the number of loops) nature of the method. The graph clearly shows that the generation time grows exponentially when the number of loops is 9 or higher. Hence, we could not obtain the generation times beyond two levels of tiling for this method. Although, it is not clear in the graph, the generation time grows exponentially even with smaller number of loops, but the difference of generation time among them is negligible.

### 6.2 Cost of parameterization

We evaluate the cost of parameterization by comparing the execution time of tiled code with fixed tile sizes and parameterized tile sizes. We use two levels of tiling one for the TLB and another for cache. This choice is motivated by our goal to compare two-level fixed and parameterized tiled codes where the differences due to the loop bounds computation can be easily quantified. Other choices for two level tiling such as tiling for parallelism and caches or tiling for caches and registers introduce many factors that influence the execution time and hence measuring the execution time difference due to the loop bounds computation becomes hard. The experiments are done on an Intel Pentium 4 at 3.2 GHz a 512 K L2 Cache and a TLB with 64 entries and pages of size 4K. We used g++ 4.1.1. compiler with -O3 optimization.

Figures 15, 16, 17 and 18 show the execution times of the two-level tiled loops for the 2D Stencil, LU decomposition, SSYRK and 3D Stencil benchmarks, respectively. For the results the shown in the graphs the inner (cache) tile sizes were varied from 2 to 512 and the outer (TLB) tile size is fixed at 512. We also experimented with other outer (TLB) tile sizes and the results (omitted for brevity) are similar to the ones presented here. We can observe that for small tile...
proposed a dual representation algorithm [26] for scanning. SUIF [32] tool includes a similar algorithm. Quillere et al. developed the widely distributed Omega library [24]. The this by extending the Ancourt-Irigoin technique, and to- requires scanning general code generation problem for affine control loops re- hexahedra with vertices and rays in addition to constraints. The an algorithm that exploits the dual representation of poly- over inequality constraints. Le Verge et al. [20, 21] proposed a single polyhedron, based on Fourier-Motzkin elimination performance (around 13 seconds) when compared to others. the classic method induce higher overhead and hence result in slower execution time. Overall, the cost of parameterization seems to be negligible and hence we conclude that parameterized tiled codes should be the preferred choice.

6.3 Effect of separation level

We evaluate the effect of separating partial and full tiles at different levels tiling. We use the STRRM benchmark, and we tield it two levels: one for cache and another for reg- isters. The register tiles were fully unrolled and the array references were replaced by scalars to facilitate register pro- motion. The running times for two different cubic register tile sizes \((2 \times 2 \times 2)\) and \((3 \times 3 \times 3)\) are shown in Figure 19. Also shown is the running time for one level of tiling for caches. First, the results clearly show that tiling for both cache and registers gives better performance. Second, they also show how splitting at the second level achieves the best performance (around 13 seconds) when compared to others.

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. [20, 21] 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 re-quires scanning \textit{unions} of polyhedra. Kelly et al. [16] solved this by extending the Ancourt-Irigoin technique, and to- gether with a number of sophisticated optimizations, de-veloped the widely distributed Omega library [24]. The SUIF [32] tool includes a similar algorithm. Quillere et al. proposed a dual representation algorithm [26] for scanning the union of polyhedra, and this algorithm is implemented in the CLooG code generator [4] 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 Triolot’s’ proof that the tiled iteration space is a polyhedron if the tile sizes are constants [13]. 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 dou- bly 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.

There has been relatively little work for the case where tile sizes are symbolic parameters, except for the very simple case of \textit{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 [33] has been to simply \textit{extend} the iteration space to a rectangular one (i.e., to consider its bounding box), apply the orthogonal technique with ap- propriate guards to avoid computations outside the original iteration space.

Amarasinghe and Lam [1, 2] implemented, in the SUFI 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. [11] proposed an extension to the polyhedral model, in which they allow arbitrary rational polynomials as co-efficients in the linear constraints that define the iteration space. Their generality comes at the price of requiring computa- tionally 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 num- ber of non-linear parameters.

Jiménez et al. [14] develop code generation techniques for register tiling of non-rectangular iteration spaces. They gen- erate code that traverses the bounding box of the tile iter- ation 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.

All the above techniques are for a single level of tiling. Rivera and Tseng [29] studied the effect of multiple levels of tiling for improving locality on multi-level caches. Multi- level tiled loop generation was not their focus. For simple rectangular iteration spaces, multi-level tiled loop gener- ation is straightforward and has been used by several tools. However, for arbitrary polyhedral iteration spaces, there has not been much work. Jiminéz et al. [15] propose a technique for arbitrary polyhedral iteration spaces but for the fixed tile sizes case. Their technique is based on the strip-mine and interchange view of tiling. Their technique has a ex-ponential complexity that grows with the number of levels of tiling. First, our technique can handle both fixed as well as parameterized tile sizes. Second, the exponential time complexity of our algorithm is fixed and does not grow with the number of levels. Third, we also propose a method to separate full and partial tiles at any arbitrary level.

![Figure 18: Total execution time for 3D Stencil for a data array of size 2048 × 2048 over 2048 time steps. The x-axis shows the inner (cache) cubic tile sizes. The outer (TLB) tile size is fixed at 512.](image-url)
Figure 19: Total execution time for triangular matrix multiplication for matrices of size $2048 \times 2048$. Two levels of tiling for cache and registers is used. The x-axis shows the cubic cache-tile sizes. The graph on the left is for a register-tile size of $2 \times 2 \times 2$ and the one on the right is for $3 \times 3 \times 3$.

8. CONCLUSION AND FUTURE WORK

Multi-level tiling is an important technique for mapping iterative computations to computer architectures with many levels of parallelism and memory hierarchy. In this paper, we have described a method for automatically generating multi-level tiled code for any polyhedral iteration space where the tile sizes can be fixed or parameterized at each level. We have shown that parameterized multi-level tiled code can be generated at the same cost as a single-level tiled code. The code generation scheme can be easily incorporated into existing general compilers and domain-specific code generators. To the best of our knowledge, ours is the first technique proposed for multi-level tiled loop generation with parameterized tile sizes and also the first method to separate partial and full tiles when the tile sizes are not fixed.

As a future work we plan to extend our techniques to case of imperfect loop nests—first for a single level and then to multiple levels. Another direction we wish to pursue is the generation of complete multi-level tiled code with both the tiled loops and the appropriate transformed loop body.

9. REFERENCES


