PrimeTile: A Parametric Multi-Level Tiler for Imperfect Loop Nests

Albert Hartono[1], Muthu Manikandan Baskaran[1], Cédric Bastoul[2], Albert Cohen[2], Sriram Krishnamoorthy[3], Boyana Norris[4], J. Ramanujam[5], and P. Sadayappan[1]

[1]Department of Computer Science and Engineering, The Ohio State University
{hartonoa, baskaran, saday}@cse.ohio-state.edu

[2]ALCHEMY Group, INRIA Saclay
{cedric.bastoul, albert.cohen}@inria.fr

[3]Computational Sciences and Mathematics Division, Pacific Northwest National Laboratory
sriram@pnl.gov

[4]Mathematics and Computer Science Division, Argonne National Laboratory
norris@mcs.anl.gov

[5]Department of Electrical and Computer Engineering, Louisiana State University
jxr@ece.lsu.edu

Abstract

Tiling is a crucial loop transformation for generating high-performance code on modern architectures. Efficient generation of multilevel tiled code is essential to maximize data reuse in deep memory hierarchies. Tiled loops with parameterized tile sizes (not compile time constants) facilitate runtime feedback and dynamic optimizations used in iterative compilation and automatic tuning. Previous parametric multilevel tiling approaches have been restricted to perfectly nested loops, where all assignment statements are contained inside the innermost loop of a loop nest. Previous solutions to tiling for imperfect loop nests have been limited to the case where tile sizes are fixed. In this paper, we present an approach to parameterized multilevel tiling for imperfectly nested loops. The tiling technique generates loops that iterate over full rectangular tiles, making them suitable for compiler optimizations such as register tiling. Experimental results using a number of computational benchmarks demonstrate the effectiveness of the developed tiling approach.

1 Introduction

Tiling is a crucial transformation for achieving high performance, especially with deep multi-level memory hierarchies. It is a well known technique for improving data locality and register reuse. Tiling has received a lot of attention in the compiler community [8,12,17,29,31–33,40]. However, the majority of work only addresses tiling of perfectly nested loops. With perfectly nested loops, tiling is possible when a band of loops is fully permutable. From a code viewpoint, tiling involves strip mining, i.e. splitting loops into an adjacent pair of tile and intra-tile loops, and permutation to bring all intra-tile loops innermost. The condition for permutability of a band of loops is that all correspondingly permuted dependence vectors must be lexicographically positive.

With imperfectly nested loops too, tiling involves strip mining – loop splitting and permutation, along with loop distribution. The reasoning about legality of tiling of imperfectly nested loops requires a more general dependence model than dependence vectors. The earliest works to develop approaches to tile imperfectly nested loops [3,23–25] effectively mapped the instances of statements of an imperfectly nested loop (of possibly different nesting depths in the input code) into a common embedding iteration space and then performed tiling in the framework of the common embedding space. A critical unresolved challenge with this approach is that of developing an effective algorithm for generating efficient tiled code where lower dimensional statements are hoisted out so that they do not have more loops surrounding them than their inherent dimensionality - heuristics have been proposed, but no generally effective implementation has been developed to our knowledge.

A major breakthrough in efficient code generation for multiple statements in imperfectly nested loops was achieved by the development of the polyhedral scanning algorithm of Quillere et al. [28] and its refinement and optimization in CLooG [6]. An innovative scheme for efficient parametric tiling of arbitrary polyhedral statement domains was developed by Renganarayana et al. [30] and extended to efficiently generate multi-level parametric tiled code [21]. However, this work is only applicable to perfect loop nests. The first effective approach for tiling of imperfectly nested loops was recently developed in the Pluto polyhedral transformation framework [7]. However, it is only able to generate tiled code with fixed tile sizes. While the approach can handle multi-level tiling, the code generation complexity grows explosively; the latest distribution of Pluto was found to fail for some inputs when generating multi-level tiled code.

In this paper we develop an effective approach to parametric multi-level tiling of imperfectly nested affine loops. The importance of parametric tiling is exemplified by the highly successful ATLAS [37, 38] system for empirical tuning of BLAS kernels. ATLAS uses parametrically tiled BLAS kernels that are repeatedly executed on the target architecture for different problem sizes using an empirical search strategy that varies the tile sizes. The ATLAS system was manually engineered by experts with insights into tiling for optimization of BLAS kernels. There has been much recent interest in developing generalized tuning systems that can similarly tune and optimize codes input by users or library developers [5, 10, 35, 39]. An efficient parametric tiling tool is extremely valuable for generating input tiled codes for such empirical tuning systems.

The key to our approach is the exploitation of the power and effectiveness of the Quillere algorithm in generating untiled imperfectly nested code, along with use of input scattering functions (affine schedules for the different statements) satisfying a generalized tiling condition and a geometric approach to generation of parametric tiles by post-processing the abstract syntax tree (AST) of the loop structure generated by CLooG.

The paper is structured as follows. A discussion of various previous efforts on tiling is provided in Section 2. We explain the key ideas behind our approach in Section 3. A detailed discussion of the algorithm and implementation are presented in Section 4. We present experimental results in Section 5. We discuss the benefits and constraints of our approach in Section 6. We conclude in Section 7.

2 Related Work

As discussed in the previous section, there have been several previous efforts that have addressed the tiling of imperfectly nested loops as well as a very effective recently developed approach for generating tiled code for perfectly nested loops involving arbitrary polyhedral statement domains. Before presenting our approach to tiling of imperfectly nested loops, we provide some details on these previous efforts.

2.1 Code Generation in Polyhedral Frameworks

There has been significant progress over the last two decades on the development of powerful compiler frameworks for dependence analysis and transformation of regular programs (programs with with affine loop bounds and array access functions) [4, 7, 13, 15, 22, 27], using a polyhedral abstraction of statement domains and data dependences. However, until recently very little attention was given to code generation, although it has a significant impact on the effectiveness of the resulting code (we use the term "code generation" for the process of transforming a polyhedral representation of computations back into loop structures). Recent advances in code generation [6, 28] have made polyhedral approaches very powerful for transforming affine loop-based code. CLooG [6, 11] is a powerful state-of-the-art code generator that is widely used.

The code generation algorithm used in CLooG is based on the one developed by Quillere et al. [28]. The key idea of the algorithm developed by Quillere et al. is that the generation of loops (along with the control code) is performed by scanning a union of polyhedra (corresponding to statement domains, which may or may not be disjoint) using a imperfectly nested loop. The scanning is performed either in lexicographic order of the indices of the statement domains or in the order imposed by any given affine-by-statement schedules. The different statement domains are embedded into a common iteration space of dimension equal to the maximum dimension of loops in the input program. The code generation algorithm recursively projects the statement polyhedra onto the outermost dimensions of the common embedded iteration space, separates the projections into disjoint polyhedra, generates loops that scan the disjoint polyhedra, and arranges the loops in the lexicographic order or according to the given affine scheduling function.

2.2 Parametric Tiling of Perfect Loop Nests

Recent work from the Colorado State University [16, 21, 30, 36] has effectively addressed parametric multi-level tiling of perfect loop nests using the polyhedral model. The tiled code generator TLOG [36] generates parametric single level tiled code for perfect loop nests. The TLOG algorithm decouples the problem of generating tiled code into two sub-problems: 1) scanning the tile origins, and 2) scanning the points within a tile. A novel technique is used to scan the tile origins by generating a polyhedron (parameterized by tile sizes) that includes all tile origins, followed by scanning of the polyhedron using CLooG [30]. A very similar approach to decomposing the problem of tiled code generation into the above mentioned sub-problems and scanning of the tile origins using polyhedral techniques was earlier proposed by Goumas et al. [14]. However, the approach of Goumas et al. only handles fixed tile sizes. TLOG can only handle rectangular tiles, and hence it requires that the input program, if not originally rectangularly tileable, be transformed to make it rectangularly tileable. Empirical evaluation of TLOG on several benchmarks demonstrates that the code generation algorithm in TLOG is efficient and that the quality of the code generated by TLOG is very good [30]. Kim et al. [21] generalized the TLOG algorithm to develop HiTLOG [16] that can generate multi-level tiled code for perfectly nested loops where the tile sizes can be symbolic variables. A significant benefit of HiTLOG is that the cost of code generation for multi-level tiling is the same as the cost of single-level tiled code generation. Jimenez et al. [19, 20] addressed parametric tiled code generation for non-rectangular iteration spaces. The code generated using their approach suffers from significant code expansion, but involves lower overhead to scan through the full tiles in the code.

2.3 Non-parametric Tiling of Imperfect Loop Nests

Ahmed et al. [2,3] was among the first to propose an approach for tiling imperfectly nested loops. The approach first determines an affine embedding for each statement into a “product space” of the iteration domains of each loop. The embedding is then optimized for locality by using another transformation matrix to achieve permutability of the dimensions. The embedding function and the transformation are sought to minimize reuse distances, based on a heuristic. The effect of the embedding function is to create a single perfectly nested loop (albeit of a much larger dimension than finally needed), with guards created for each statement to ensure correct execution. Eliminating unnecessary guards is extremely difficult. Lim et al. [23] used an affine partitioning framework for tiling; the framework builds on an affine partitioning algorithm they proposed earlier [24, 25]. Parametric tiling is not considered in their work. Some specialized works [9, 34, 41] exist for tiling a restricted class of imperfectly nested loops.

Bondhugula et al. developed Pluto [7, 26], a system for tiling arbitrary collections of imperfectly nested loops. The Pluto system finds tiling transformations that result in data locality optimization for sequential execution and communication minimization for parallel execution. The tiling is performed using a generalization of the validity condition for tiling (explained in the next section), originally presented for perfectly nested loops by Irigoin and Triolet [18]. The Pluto system requires the tile sizes to be fixed for code generation. A second drawback of the Pluto system is that it generates multi-level tiling only for up to two levels. In our experiments, we used a script that enables higher levels of tiling with Pluto, using the same approach to tiling as implemented within the tool (described in [7]).

```
for (i=lbi; i<=ubi;i+=sti) {
  for (j=lbj(i); j<=ubj(i); j+=stj)
    S(i,j);
}
```

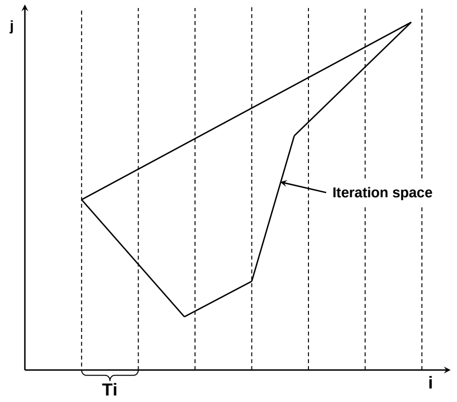
(a) A perfectly nested loop

```
/* tiled loop */
for (it=lbi; it<=ubi-(Ti-sti); it+=Ti) {
   ... (code tiled along j) ...
}
/* epilog loop */
for (i=it; i<=ubi; i+=sti) {
  for (j=lbj(i); j<=ubj(i); j+=stj)
    S(i,j);
}
```

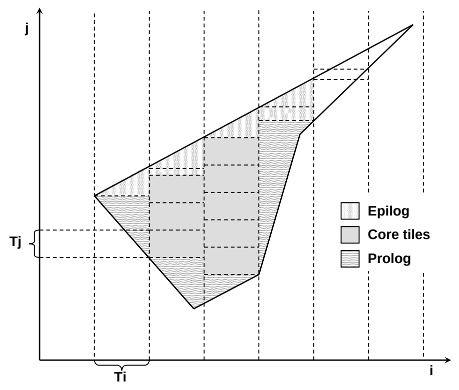
(b) Tiling the outermost loop i

```
for it
  [scan code to obtain lbv,ubv]
  if (lbv < ubv) {
    [prolog j]
    [tiled j]
    [epilog j]
  } else {
    [untiled j]
  }
}
[epilog i]
```

(c) After tiling loops i and j



(d) Iteration space (tiled along i axis)



(e) Iteration space (tiled along i and j axes)

```
/* tiled i */
for (it=lbi; it<=ubi-(Ti-sti);it+=Ti) {
  /* scan code */
  lbv = MIN_INT;
  ubv = MAX_INT;
  for (i=it; i<=it+(Ti-sti); i+=sti) {
    lbv = max(lbv, lbj(i));
    ubv = min(ubv,ubj(i));
  }
  if (lbv<ubv) {
    /* prolog j */
    for (i=it; i<=it+(Ti-sti); i+=sti)
      for (j=lbj(i); j<=lbv-stj;j+=stj)
        S(i,j);
    /* tiled j */
    for (jt=lbv; jt<=ubv-(Tj-stj);jt+=Tj)
      for (i=it; i<=it+(Ti-sti); i+=sti)
        for (j=jt; j<=jt+(Tj-stj); j+=stj)
          S(i,j);
    /* epilog j */
    for (i=it; i<=it+(Ti-sti); i+=sti) {
      for (j=jt; j<=ubj(i); j+=stj)
        S(i,j);
    }
  } else {
    /* untiled j */
    for (i=it; i<=it+(Ti-sti); i+=sti)
      for (j=lbj(i); j<=ubj(i); j+=stj)
        S(i,j);
  }
}
/* epilog i */
for (i=it; i<=ubi;i+=sti)
  for (j=lbj(i); j<=ubj(i); j+=stj)
    S(i,j);
```

(f) Detailed parametric tiled code

Figure 1: Parametric tiling of a perfectly nested loop

3 Overview of Approach

Before providing a detailed description of our tiling algorithm, we first use a number of examples, figures and pseudocode fragments to explain the key ideas behind the approach we have developed. We first begin by discussing the approach to generation of parametric full tiles in the context of perfectly nested loops. We then discuss the conditions under which we can tile imperfectly nested loops, followed by a sketch of our approach to geometric separation of tiles for imperfectly nested loops.

3.1 Parametric Tiling for a Single Statement Domain.

In order to facilitate the presentation of the algorithm for tiling imperfectly nested loops, we first explain how the approach works in the simpler context of a single statement domain. It is quite different from the tiling approach implemented in HiTLOG [21, 30]. Consider the simple 2D perfectly nested loop shown in Figure 1(a). The perfect loop nest contains an inner loop j whose bounds are arbitrary functions of the outer loop variable i. Consider a non-rectangular iteration space displayed in Figure 1(d), corresponding to the perfect loop nest in this example. Since loop i is outermost, strip-mining or tiling this loop is straightforward (that is, to partition the loop i's iteration space into smaller blocks whose size is determined by the tile size parameter Ti). Figure 1(d) shows the partitioning of the iteration space along dimension i. Figure 1(b) shows the corresponding code structure, with a first segment covering as many "full" tiling segments along i as possible (dependent on the parametric tile size Ti). The outer loop in the tiled code is the inter-tile loop that enumerates all tile origins. Following the full-tile segment is an epilog section that covers the remainder of iterations (to be executed untiled). The loop enumerates the points within the last incomplete group of outer loop iterations that did not fit in a complete i-tile of size Ti.

For each tiling segment along i, full tiles along j are identified. For ease of explanation, we show a simple "explicit scanning" approach to finding the start of full tiles, but the actual implementation computes it directly for affine loop bounds by evaluating the bound functions at corner points of the outer tile extents. The approach we develop is also applicable to general loops with arbitrary non-affine and non-convex bounds, by using explicit scanning. The essential idea is that the largest value for the j-lower bound (lbv) is determined over the entire range of an i-tile and it represents the earliest possible j value for the start of a full ij tile. In a similar fashion, by evaluating the upper-bound expressions of the j loop, the highest possible j value (ubv) for the end of a full tile is found. If lbv is greater than ubv, no full tiles exist over this i-tile range. In Figure 1(e), this is the case for the last two i-tile segments. For the first i-tile segment in the iteration space (the second vertical band in the figure, the first band being outside the polyhedral iteration space), lbv equals ubv. For the next two i-tile segments, we have some full tiles, while the following i-tile segment has ubv greater than lbv but by a lesser amount than the tile size along j.

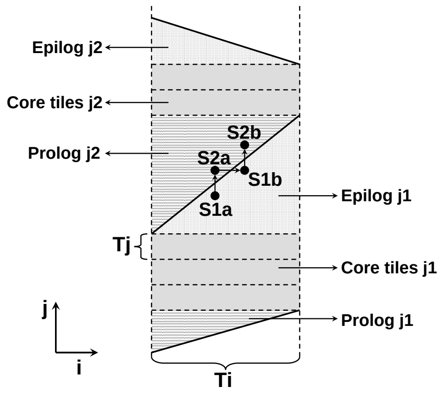
The structure of the tiled code is shown in abstracted pseudo-code in Figure 1(c), and with explicit detail in Figure 1(f). At each level of nesting, for a tile range determined by the outer tiling loops, the lbv and ubv values are computed. If ubv is not greater than lbv, an untiled version of the code is used. If lbv is less than ubv, the executed code has three parts: a prolog for j values up to $lbv - stj$ (where stj is the loop stride in j dimension), an epilog for j values greater than or equal to jt (where jt is the inter-tile loop iterator in j dimension), and a full-tile segment in between the prolog and epilog, to cover j values between the bounds. The code for the full-tile segment is generated using a recursive procedure that traverses the levels of nesting. The detailed tiled code for this example is shown in Figure 1(f).

3.2 Tiling of Multi-Statement Domains

The iteration-space view of legality of tiling for a single statement domain is expressed as follows: a hyperplane H is valid for tiling if $Hd_i \geq 0$ for all dependence vectors d_i [18]. This condition states that all dependences are either along the tiling hyperplane or enter it from the same side. This sufficient condition ensures that there

```
for (i=lbi;i<=ubi;i+=sti) {
  for (j1=lbj1(i);j1<=ubj1(i);j1+=stj1) {
    S1(i,j1);
  }
  for (j2=lbj2(i);j2<=ubj2(i);j2+=stj2) {
    S2(i,j2);
  }
}
```

(a) An imperfect loop nest structure



(b) One tile segment along i dimension

```
for it
  [scan code to obtain lbv1,ubv1]
  if (lbv1 < ubv1) {
    [scan code to obtain lbv2,ubv2]
    [prolog j1]
    [tiled j1]
    if (lbv2 < ubv2) {
      [epilog j1 + prolog j2]
      [tiled j2]
      [epilog j2]
    } else {
      [epilog j1 + untiled j2]
    }
  } else {
    [scan code to obtain lbv2,ubv2]
    if (lbv2 < ubv2) {
      [untiled j1 + prolog j2]
      [tiled j2]
      [epilog j2]
    } else {
      [untiled j1 + untiled j2]
    }
  }
}
[epilog i]
```

(c) The final tiled code

Figure 2: Tiling an imperfectly nested loop

cannot be cyclic dependences between any pair of tiles generated using families of hyperplanes that each satisfy the validity condition.

For a collection of polyhedral domains corresponding to a multi-statement program (from imperfectly nested loops), the generalization of the above condition is: a set of affine-by-statement functions ϕ (corresponding to each statement in the program) represents a valid tiling hyperplane if $\phi_t(\vec{t}) - \phi_s(\vec{s}) \geq 0$ for each pair of dependences $(\vec{s}, \vec{t})$ [7]. The affine-by-statement function ϕ maps each instance of each statement to a point in a dimension of a target iteration space. A set of linearly independent ϕ functions maps each instance of each statement into a point in the multi-dimensional target space. If each ϕ function satisfies the above generalized tiling condition, the multi-statement program can be rectangularly tiled in the transformed target iteration space. If only a (contiguous) subset of the ϕ functions satisfies the generalized tiling condition, tiles can be formed using families of hyperplanes from that subset.

Efficient code generation for multi-statement domains was a significant challenge until the Quillere algorithm was developed. Its implementation in CLooG is now widely used for generating code for multi-statement domains. The Pluto system uses CLooG for generating (non-parametric) tiled code for imperfectly nested loop programs. However, Pluto cannot generate parametric tiled code for imperfectly nested loops. The key idea behind our new approach to parametric tiling of imperfect loop-nests is to combine the power of the Quillere algorithm (in sorting and separating polyhedra corresponding to multiple-statement domains) with a geometric approach to tile separating, using the AST structure generated by the Quillere algorithm for non-tiled imperfectly nested loop code generation.

As elaborated later in the next section, first an input program is transformed to a target domain using scattering functions that satisfy the above generalized tiling condition. For this purpose, we simply use scattering functions generated by the Pluto system, but any set of schedules that satisfy the generalized tiling condition can be used instead. The imperfectly nested loop structure generated by use of the Quillere algorithm is scanned to generate the tiled code structure as described in the next sub-section.

3.3 Geometric Separation of Tiles for Overlapping Statement Domains

We use a simple imperfectly nested loop example shown in Figure 2 to illustrate the approach to geometric tile separation. The imperfectly nested loop i considered in this example contains two inner loops with loop bounds that are functions of loop iterator i and other global parameters such as tile sizes and input problem sizes. The Quillere algorithm generates efficient (non-tiled) loop code for multi-statement polyhedral domains arising from imperfectly nested loops. Where feasible, the sorting of polyhedra within the Quillere algorithm enables separation of statements in the point-wise (non-tiled) code. The key tiling question for this two-statement example is: if the two statements S1 and S2 have been separated out in the point-wise code by the Quillere algorithm, under what conditions can we also separate out tiles corresponding to these two statements? Our answer to this question is to use the lower and upper bound values for the two statements, (computed in a similar manner to the perfect-nest example above) and exploit the fact that all dependences are lexicographically non-negative in all the tiling dimensions (due to satisfaction of the generalized tiling condition).

For the example shown in Figure 2(a), since $lbv1$ is less than $ubv1$, we have a separable set of tiles for S1, and since $lbv2$ is less than $ubv2$, there are also separable tiles for S2. The prolog of S2 and epilog of S1 need to be combined and interleaved to ensure satisfaction of any dependences between S1 to S2 or vice versa. The pseudocode in Figure 2(c) shows the different possible cases to be considered and the code corresponding to the four combinations.

4 Algorithm for Parametric Tiling of Imperfectly Nested Loops

In this section, we present details of the approach to parametric multilevel tiling. Given a sequence of arbitrarily nested affine loops, tiling involves three steps:

1. Pre-processing: Extraction of statement polyhedra, computation of dependence polyhedra, and generation of a valid affine schedule where a band of the scheduling functions satisfies the generalized tiling condition;

2. AST generation with preserved embedding information: Use of the Quillere algorithm to scan the statement polyhedra and generate imperfectly nested loop structure (AST), with preservation of complete embedding information;

3. Recursive traversal of the AST to generate parametric tiled code.

4.1 Preprocessing and AST Generation

The input to the tiling algorithm is a sequence of arbitrarily nested loops with loop bounds and array accesses that are functions of outer loop variables and program parameters. First, the imperfect loop nest is made rectangularly tileable, using skewing and other unimodular transformations. The required transformations can be captured in the forms of scattering functions (or affine scheduling functions). While any suitable scheduling functions that satisfy the generalized tiling condition (described in Sec. 3.2) may be used, for our experiments described later, we have just used the scattering functions generated by Pluto [7].

The second step in the tiling procedure is the generation of an AST for the imperfectly nested loop structure generated by application of the Quillere polyhedral scanning algorithm. We use an adaptation of the implementation of the Quillere algorithm in CLooG to ensure that all embedding information for all statements is explicitly preserved in the AST. Normally, CLooG generates optimized imperfectly nested code structures where each statement is only enclosed by as many loops as its inherent dimensionality. Thus, if a multi-statement domain corresponding to a 2D statement and a 3D statement were scanned, the output code would be

```
for (t=0;t<=T−1;t++) {
  for (i=1;i<=N−2;i++) {
    B[i]=(A[i−1]+A[i]+A[i+1])/3; /* S1 */
  }
  for (i=1;i<=N−2;i++) {
    A[i]=B[i]; /* S2 */
  }
}
```

(a) Original code

$S1 : (t', i') = (t, 2 * t + i)$
$S2 : (t', i') = (t, 2 * t + i + 1)$

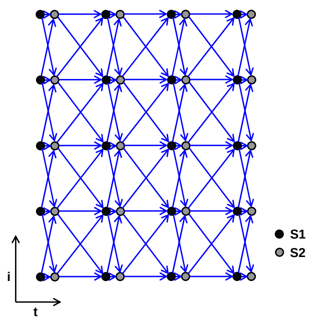
(b) Affine transformation (skewing)

```
for (t=0;t<=T−1;t++) {
  B[1]=(A[1+1]+A[1]+A[1−1])/3;
  for (i=2*t+2;i<=2*t+N−2;i++) {
    B[−2*t+i]=(A[1+−2*t+i]+A[−2*t+i]
                +A[−2*t+i−1])/3;
    A[−2*t+i−1]=B[−2*t+i−1];
  }
  A[N−2]=B[N−2];
}
```

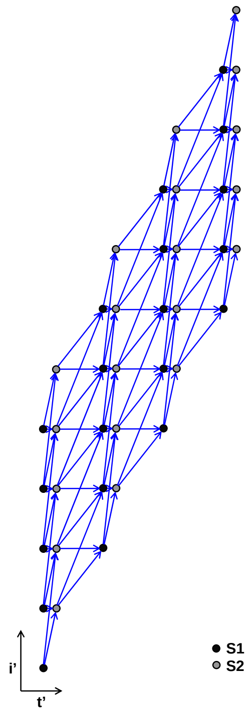
(c) Skewed code

```
for (t=0;t<=T−1;t++) {
  for (i=2*t+1;i<=2*t+1;i++) {
    B[1]=(A[1+1]+A[1]+A[1−1])/3;
  }
  for (i=2*t+2;i<=2*t+N−2;i++) {
    B[−2*t+i]=(A[1+−2*t+i]+A[−2*t+i]
                +A[−2*t+i−1])/3;
    A[−2*t+i−1]=B[−2*t+i−1];
  }
  for (i=2*t+N−1;i<=2*t+N−1;i++) {
    A[N−2]=B[N−2];
  }
}
```

(d) Skewed code with one−time loops



(e) Original iteration space



(f) Skewed iteration space

Figure 3: Example of preprocessing: 1D Jacobi code

imperfectly nested, with only two surrounding loops for the 2D statement and three surrounding loops for the 3D statement. However, this optimized code structure loses the embedding information for the 2D statement within the 3D embedded domain. But this embedding information is essential for our approach to geometric separation of multi-statement tiles. We have therefore adapted the CLooG code generator to explicitly generate redundant ”one-trip-count” loops so that all statements have as many surrounding loops as the dimensionality of the embedding target space, thereby explicitly preserving the embedding of all statements. As we discuss later, a post-processing step after tiling removes these redundant one-iteration loops in the final generated code to make it efficient.

Figure 3 illustrates the preprocessing steps and the effect of our adaptation of CLooG to create redundant loops for explicit representation of complete embedding information for all statement domains of different dimensionalities. Figure 3(a) shows a 1D Jacobi stencil computation. Figure 3(e) shows the iteration space for the computation. It can be seen that there are data dependences with negative components, i.e., the code is not tileable. Skewing can be done to eliminate these backward dependences. As seen in Figure 3(f), the iteration space after skewing no longer has backward data dependences and rectangular tiling now becomes valid. Figure 3(b) shows the affine transformation generated by Pluto and Figure 3(c) shows the output code after the transformation is processed using CLooG. It can be seen that there is a doubly nested loop along with two “peeled” 1D statements `B[1]=(A[1+1]+A[1]+A[1-1])/3` and `A[N-2]=B[N-2]`. The output from the adapted version of CLooG to explicitly preserve the embedding information for these two 1D loops in the 2D embedding domain is shown in Figure 3(d). Note that the iterator variable of the redundant one-time loop is never actually used inside the scope of the loop body as its value has already been propagated properly by CLooG. Consequently, later removal of these one-time loops is always safe. As a post-processing optimization after tiling, these dummy one-time loops get removed from the final tiled code.

Algorithm 1 Parametric tiling algorithm

```
INPUT $inputStmts$ : sequence of statements to be tiled, $precedingStmts$ : sequence of statements that precede $inputStmts$,
    $outerLoops$ : sequence of outer tiled loops
 1: $unprocStmts =$ all statements in $precedingStmts$ that are marked as unprocessed
 2: $procStmts = precedingStmts - unprocStmts$
 3: if $inputStmts$ is empty then
 4:     Merge $unprocStmts$ and enclose them with intra-tile loops that correspond to $outerLoops$, and insert the formed loop nest
        into $procStmts$. Then return $procStmts$.
 5: end if
 6: $firstStmt =$ the first statement in $inputStmts$
 7: $remainingStmts = inputStmts - \langle firstStmt \rangle$
 8: if $firstStmt$ is not a loop then
 9:     Mark $firstStmt$ as an unprocessed statement, and then return Tile($remainingStmts$, $precedingStmts + \langle firstStmt \rangle$,
        $outerLoops$).
10: end if
11: $curLoop = firstStmt$
12: Generate an explicit scanning code ($scanCode$) to find the latest starting value ($startVal$) and the earliest ending value ($endVal$)
    of $curLoop$ variable, for the ranges of $outerLoops$ tiles.
13: Generate a prolog code ($prologCode$) that traverses $unprocStmts$ and the opening boundary tile of $curLoop$, enclosed with
    intra-tile loops corresponding to $outerLoops$.
14: $tiledStmts$ = Tile(all statements inside $curLoop$ body , $\langle\rangle$, $outerLoops + \langle curLoop \rangle$)
15: Use $tiledStmts$ to generate an inter-tile loop ($tiledCode$) for $curLoop$ that starts from $startVal$ to $endVal$.
16: Generate an epilog code ($epilogCode$) that traverses statement instances that come after $tiledCode$ and are inside the closing
    boundary tile of $curLoop$. Mark $epilogCode$ as an unprocessed statement.
17: if both the lower and upper bounds of $curLoop$ are free of $outerLoops$ iterators then
18:     Return Tile($remainingStmts$, $\langle scanCode \rangle + procStmts +$ $\langle prologCode, tiledCode, epilogCode \rangle$, $outerLoops$).
19: else
20:     $tileableStmts =$ Tile($remainingStmts$, $\langle prologCode, tiledCode, epilogCode \rangle$, $outerLoops$)
21:     $untileableStmts =$ Tile($remainingStmts$, $unprocStmts + \langle curLoop \rangle$, $outerLoops$), where $curLoop$ has already been
        marked as an unprocessed statement.
22:     Generate an if-statement ($ifCode$) as follows: if ($startVal < endVal$) $tileableStmts$ else $untileableStmts$.
23:     Return $\langle scanCode \rangle + procStmts + ifCode$.
24: end if
OUTPUT The sequence of transformed statements after tiling
```

4.2 Tiled Loop Generation Algorithm

This section discusses the tiling algorithm used for generating parametric one-level tiled code for imperfect loop nests. In a later section, we discuss an extension of the approach to generate multi-level tiled code.

The algorithm recursively decomposes a given imperfect loop nest into code segments that scan partial tiles and full rectangular tiles. At each step, our task is: given an imperfect loop nest at a particular nesting level and a set of outer tiled loops, divide the current loop into a sequence of loop nests that scan a set of tiles whose union covers all points (in the iteration space corresponding to the given loop nest, for the ranges of the outer tiles).

As discussed earlier in Section 3.1, three types of code segments are generated to scan disjoint partial and full tiles: prolog, tiled-loop, and epilog. While prolog and epilog correspond to scanning partial tiles, tiled-loop corresponds to scanning full tiles. It is possible that a tiled statement polyhedron has empty full tiles. Hence, we need to generate a scan code and an if-statement to verify the presence of full tiles. For ease of explanation, let us first assume that full tiles always exist in each tiled statement polyhedron. This means that we need not generate the scan code and the if-statement, simplifying the tiled code generation problem.

The tiled code generation procedure takes three objects as input: a sequence of ASTs at the same nesting level that need to be tiled (A), all preceding ASTs that have previously been processed (P), and a set of all outer tiled loops (O) ordered from outermost to innermost. The output of the recursive procedure is a sequence of transformed loops that scan the tiled iteration space. We use *Tile* to denote the recursive tiling procedure, a to denote the first AST in A, and R for the remaining ASTs in A (that are at index positions greater than or equal to two). Given the three input arguments A, P, and O, the following recursive procedure solves our simplified tiling problem.

1. If A is empty, then return P.
2. If a is not a loop, then recurse to the next AST in A (that is, Tile(R,$P + \langle a \rangle$, O)) and return the output.
3. Otherwise, perform a tail recursion on all statements that are inside A's loop body (say s) by calling Tile(s, $\langle\rangle$, $O + \langle a \rangle$).
4. Take the transformed loops obtained from the previous step, and selectively merge all adjacent prolog and epilog into a single loop. After that, use the transformed loops to construct a tiled loop (say T_l).
5. Generate a prolog loop (say P_l) and an epilog loop (say E_l).
6. Recurse to the next AST in A, by calling Tile(R,$P + \langle P_l, T_l, E_l \rangle$, O) and then return the output.

In the simplified algorithm, $\langle \ldots \rangle$ notation represents a sequence of objects. The "$+$" operator is used to combine two sequences and the "$-$" operator is used to remove from the first sequence operand all elements that exist in the second sequence operand.

Note that the generation of inter-tile and intra-tile loops is implicit in our simplified algorithm. These loops can be derived from the given a and O. Details on generating inter-tile and intra-tile loops will be provided in the detailed tiling algorithm later in this section.

The key idea of the tiling algorithm is to use a recursion to perform a *depth-first traversal* of the input AST, and to recursively convert each loop node into a sequence of loops: a prolog loop, a tiled loop, and an epilog loop. At each step of the recursive procedure, if all nodes at a certain nesting level have already been transformed, all contiguous prolog and epilog originated from different statement polyhedra will be fused and interleaved (Step 4). This is to ensure that any data dependences between statement instances in the two distinct iterations domains are not violated. After that, all of these transformed loops are combined together inside a tiled loop. Step 5 of the algorithm generates a prolog and an epilog. The generated prolog, tiled loop, and epilog are then returned as output of the procedure.

The general tiled code generation algorithm is shown in Algorithm 1. The same fundamental idea of the simplified algorithm is used. The generation of inter-tile and intra-tile loops is now included in the detailed algorithm in Steps 4, 13, and 15. The tiling algorithm generates a scan code in Step 12 and also an if-statement in Steps 20-22, both used to identify at run time whether the loop iteration domain contains full tiles. Two recursive calls are made to produce an untiled version and a tiled version of the loop. Each generated loop version then becomes a separate statement block of the constructed conditional statement. By recursively creating an if-statement for each loop at one nesting level, a nested if-statement structure will be generated in the final tiled code. Consider a 2D imperfect loop nest that has n non-rectangular inner loops. Applying our tiling procedure to the 2D loop nest will produce a tiled code with a total of 2^n possible tile cases. Note that although it is possible to use a simple loop to traverse all loop nodes at the same nesting level, we choose to use a *tail recursion*. The generation of nested if-statements becomes simple with a tail recursion. The output of tiling the current loop node (i.e., a sequence of transformed loops) can be replicated and passed as an input argument to the next level of recursion for creating all possible tile cases at the innermost level of the nested if-statements.

In the special case of rectangular loop nests, the bounds of the current loop are free of any outer loop variables, so that the tiled iteration space has no opening boundary tiles. Consequently, no prolog loop gets generated. Also, the scan code and the if-conditional statement need not be produced since the rectangular iteration domain can always be partitioned into full and partial rectangular tiles, using a tiled loop and an epilog.

4.3 Enhancements to the Core Algorithm

We now address a number of enhancements to the core algorithm that are implemented in the tiled loop generator.

4.3.1 Multi-level Tiling

The core tiling algorithm described in Section 4.2 generates efficient parametric tiled code by separating each statement polyhedron into partial and full tiles. Multi-level tiling is important to exploit locality in a deep memory hierarchy. Since the tiling algorithm separates all partial and full tiles and the tiled loops that iterate over full tiles can be known at compile time, the main tiling algorithm can be extended to generate multilevel tiled code. From the implementation perspective, the extension involves adding another argument to the tiling procedure that describes the number of level of tilings ($tileLevel$), and modifying the termination step (Step 4 in Algorithm 1) of the recursive tiling procedure. The termination step generates the set of intra-tile loops that correspond to the given sequence of outer tile loops, to visit all points inside a tile. Hence, instead of generating intra-tile loops, we modify this step to generate $tileLevel - 1$ sets of inter-tile loops, starting from level $tileLevel - 1$ (as outermost) to level one (as innermost), and finally to generate another set of intra-tile loops at the innermost level.

4.3.2 Optimizing Boundary Tiles

The approach discussed so far does not optimize the boundary tiles. This may result in lower performance as the total area of all boundary tiles can generally be very large especially when large tile sizes are used at the outermost level of tiling. Thus, it is important to optimize the boundary tiles to achieve high-performance tiled code. Loops that scan partial tiles are the prolog and epilog codes, both generated at Steps 13 and 16 of Algorithm 1, respectively. After prolog and epilog codes are completely generated (enclosed with intra-tile loops at Steps 13 and 4 respectively), we can further tile the boundary tiles by calling the multi-level tiling procedure with a tile size that is used at the next lower level of tiling (i.e., $tileLevel - 1$). In this way, all newly formed boundary tiles can be recursively tiled and refined into smaller full and partial tiles. Figure 4 shows an example of an iteration domain recursively tiled for three levels of tiling. From the figure we can clearly see the difference in terms of the total area of untiled partial tiles, between using and not using boundary tile optimization.

4.3.3 Static Determination of the Start and End of Full Tiles

In regular programs where loop bounds are affine functions of outer loop iterators and global parameters, it is possible to statically determine the upper and lower bounds of possible full tiles that are parameterized by tile sizes, along each dimension of the iteration space. We determine the start (or lower bound) and end (or upper bound) of full tiles along each dimension, starting from the outermost to the innermost. When we calculate

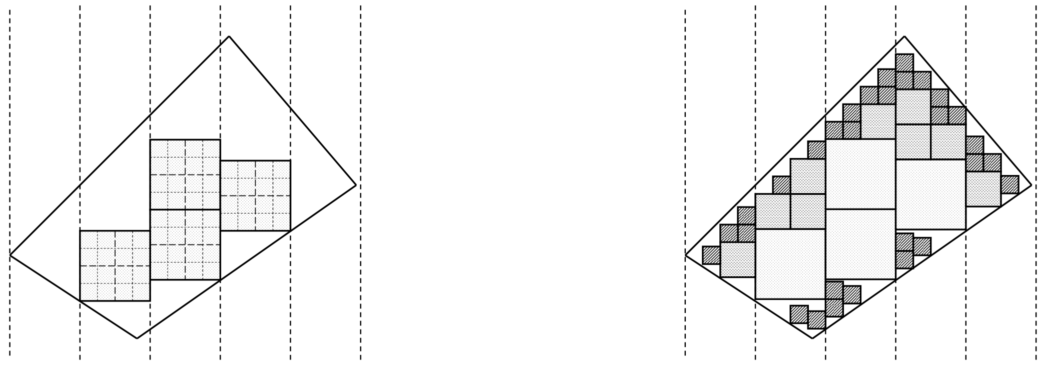


Figure 4: Iteration space tiled for three levels of tiling; without boundary tile optimization (left) and with boundary tile optimization (right)

the lower bound of full tiles along a dimension, we utilize the fact that the lower and upper bounds of the tiled loops of the outer dimensions are already known and that the loop iterator corresponding to the dimension is an affine function of the loop iterators of the outer dimensions. The start of the full tiles is evaluated from the affine lower bound expression by substituting the loop iterators of the outer dimensions with the lower or upper bound of the tiled loops of the outer dimensions according to the following condition: if the coefficient of a loop iterator in the affine bound expression is positive, then we substitute it with the upper bound of the tiled loop along the dimension, and vice versa if the coefficient is negative. In a similar manner, we determine the end of the full tiles.

4.3.4 Optimizing One-Time Loops

As mentioned earlier in this section, loops running exactly once are explicitly inserted into the input AST for the tiled loop generator. Generating tiled code for loop structures that contains one-time loops using the tiling algorithm will of course generate correct results. However, we can substantially reduce both the code generation time and the efficiency of the generated code by post-processing to eliminate the one-time loops. In Steps 20-22, an explicit if-condition is generated to check whether the current loop is tileable or not. This check requires finding the start and end of full tiles, derived from directly evaluating the bound functions at the corner points of given ranges of outer tiles. If the bounds are the same ($lb = ub$), which is true for one-time loops, then the start of the full tiles will always be greater than or equal to the end of the full tiles. This implies that the current loop is not tileable. Hence, the corresponding true case of the generated if-statement will never be evaluated and, therefore, will not be generated and used during code generation time. In other words, all one-time loops are always left completely untiled.

5 Experimental Evaluation

In this section, we discuss various experiments carried out to assess the effectiveness of the developed tiling approach. We base our comparison over two state-of-the-art tiled-code generators, Pluto [26] and HiTLOG [16]. As discussed earlier, Pluto is a polyhedral transformation framework that can generate tiled code for imperfectly nested loops using fixed tile sizes. HiTLOG is a polyhedral tiled code generator that can generate parametric tiled code for perfectly nested loops. We primarily compare our tiled code generator, PrimeTile, with Pluto (version 0.4.1), which is the only system we are aware of that can generate tiled code for imperfectly nested loops. In addition, for the special case of perfectly nested loops, we also compare PrimeTile with HiTLOG. We note that PrimeTile is more general and powerful than both these compared systems: 1) Pluto can only generate code with fixed tile sizes for imperfectly nested loops, while PrimeTile generates parametric tiled code, suitable for direct use in generalized versions of tuning systems such as ATLAS; 2) HiTLOG can generate

Name	Description	Imperfect nest	Require skewing	Max. loop depth	Input problem sizes
LU	LU factorization	Yes	No	3	$N = 2500$
2D FDTD	2D Finite Difference Time Domain method	Yes	Yes	3	$T = 2000, N = 2000$
1D Jacobi	1D Jacobi method	Yes	Yes	2	$T = 2000, N = 6 \times 10^6$
Cholesky	Cholesky factorization	Yes	No	3	$N = 5000$
TriSolver	Triangular solver	Yes	No	3	$N = 3000$
Seidel	3D Gauss Seidel	No	Yes	3	$T = 2000, N = 2000$
DSYRK	Symmetric rank k update	No	No	3	$N = 3000$
DTRMM	Triangular matrix multiplication	No	No	3	$N = 3000$

Table 1: Benchmarks used in the experiments

parametric tiled code only for perfectly nested affine loops, while PrimeTile generates parametrically tiled code for arbitrary imperfectly nested affine codes.

We use a set of eight benchmarks that include linear algebra kernels and stencil computations, as listed in Table 1. The collection of benchmarks includes three perfectly nested cases and five imperfectly nested cases. As pointed out earlier, due to data dependences, skewing and other unimodular transformations may be needed to make rectangular tiling valid; the need for such skewing transformation is indicated in the fourth column. The next column displays the maximum depth level of each benchmark. The last column describes the input sizes used for the experiments. In order to perform a fair comparison of PrimeTile, Pluto and HiTLOG, we made use of a convenient feature of the Pluto system – the option `--opt`, which causes Pluto to simply transform the code without tiling, but using exactly the same hyperplanes (scattering functions) that would have been used to generate tiled code if the `--tile` option had been used. This ensures that any skewing needed to ensure rectangular tileability of the code is performed. Intermediate CLooG files generated by Pluto are used as inputs for HiTLOG and PrimeTile, ensuring that all three systems perform tiling on an identically pre-processed version of the input code.

All experiments were run on a multicore Intel Xeon workstation. The workstation has dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64). We use version 4.2.4 of g++ (GCC) with -O3 optimization flag to compile the generated codes.

5.1 Efficiency of Code Generation

This section evaluates the efficiency of the code generation process with the developed tiling tool. We show how well each tiling method scales with respect to the number of tiling levels in the generated code. Two tiled versions are generated using PrimeTile. One version (labeled "PrimeTile(f)") is the tiled code in which boundary tiles are also recursively tiled (using smaller tile sizes). The other version (labeled "PrimeTile(n)") represents tiled code in which boundary tiles are not tiled at all. The time taken for code generation for the five imperfect nest benchmarks and the three perfect nest benchmarks are given in Table 2 and Table 3, respectively. Due to the fact that PrimeTile and Pluto can handle imperfectly nested loops, the time taken to generate code increases with the number of levels of tiling, while the time taken by HiTLOG remains almost the same for increasing levels of tiling. The time taken to generate the PrimeTile(n) version increases insignificantly for increasing levels of tiling. However, there is a trade-off with performance compared to the PrimeTile(f) version as illustrated later in the section. PrimeTile is implemented with the ability to control the depth of tiling recursion for the boundary tiles. Hence, we can generate tiled versions (between PrimeTile(n) and PrimeTile(f)) that vary in the level of tiling performed for the boundary tiles.

A drawback of the Pluto system, as indicated in Section 2, is that it generates multi-level tiling only for up to two levels. In our experiments, we use a script that extends Pluto for additional levels of tiling. Pluto supplies the code generator (CLooG) with higher dimensional iteration domains using tile shape constraints, using the approach proposed by Ancourt and Irigoin [4], and duplicated scattering functions for the tile space.

	1 level of tiling			2 levels of tiling			3 levels of tiling			4 levels of tiling		
	Pluto	Prime-Tile(f)	Prime-Tile(n)	Pluto	Prime-Tile(f)	Prime-Tile(n)	Pluto +script	Prime-Tile(f)	Prime-Tile(n)	Pluto +script	Prime-Tile(f)	Prime-Tile(n)
LU	0.03	0.27	0.27	0.20	0.48	0.28	-	1.59	0.28	-	6.88	0.29
2D FDTD	0.25	0.56	0.56	3.02	1.84	0.57	-	9.24	0.58	-	63.66	0.59
1D Jacobi	0.03	0.28	0.30	0.06	0.37	0.29	0.31	0.68	0.30	3.87	1.62	0.30
Cholesky	0.07	0.37	0.37	0.74	0.82	0.39	13.74	2.79	0.42	-	11.21	0.45
TriSolver	0.08	0.31	0.32	1.77	0.52	0.32	-	1.28	0.34	-	3.77	0.37

Table 2: Tiled code generation times (in seconds) of different tiling methods on imperfect nest benchmarks

	1 level of tiling				2 levels of tiling				3 levels of tiling				4 levels of tiling			
	Pluto	Prime-Tile(f)	Prime-Tile(n)	Hi-TLOG	Pluto	Prime-Tile(f)	Prime-Tile(n)	Hi-TLOG	Pluto +script	Prime-Tile(f)	Prime-Tile(n)	Hi-TLOG	Pluto +script	Prime-Tile(f)	Prime-Tile(n)	Hi-TLOG
Seidel	0.02	0.32	0.320	0.09	0.06	0.85	0.325	0.09	9.57	4.10	0.34	0.10	-	221.01	0.35	0.10
DSYRK	0.02	0.26	0.26	0.12	0.05	0.34	0.26	0.11	2.63	0.69	0.27	0.11	-	1.81	0.28	0.10
DTRMM	0.02	0.26	0.26	0.11	0.10	0.38	0.26	0.09	28.42	0.85	0.27	0.09	-	2.49	0.28	0.09

Table 3: Tiled code generation times (in seconds) of different tiling methods on perfect nest benchmarks

We manually analyze the CLooG input file generated by Pluto for each benchmark. We then create a script (for each benchmark) that parses the CLooG input file and modifies the extracted domains and scattering functions with additional tiling dimensions using the same method that Pluto uses for up to two levels of tiling.

Pluto fails to generate tiled code for more than four levels of tiling (because of overflow errors in calls to Polylib within CLooG). Our system successfully generates tiled code for any number of levels of tiling (we tried up to 8 levels). Except for the case of one-level tiling, PrimeTile is generally faster than Pluto. PrimeTile is implemented in Python, and hence is inherently slower than the C-based code generator executables in Pluto and HiTLOG. This is why PrimeTile is slower than Pluto for one-level tiling.

5.2 Performance of Generated Tiled Code

In this section, we assess the efficiency of the tiled code generated by PrimeTile. We generate tiled code using one level of tiling and two levels of tiling for various tile sizes. When one level tiling is used, the considered tile sizes are 2^n for n ranging from 1 to 10. In the two level tiling case, the sizes of the outer tile ($T2$) are 128, 256, 512, 768, and 1024, whereas the inner tile sizes ($T1$) range from 2 to $T2/2$. We use square tiles just for ease of experimentation. We generate tiled code using the three different tiled code generators for all combinations of the tile sizes under consideration. We then measure the execution time of each generated code, and finally select the best performing tiled code. Table 4 lists the best execution times of tiled codes generated by using different tiling techniques.

We further optimize the best performing tiled code to show the benefits of having full tiles. For the tiled code generated by PrimeTile and HiTLOG, we perform another low level of tiling on the best tiled code, unroll the full tiles at the innermost level, and apply scalar replacement optimization to improve locality at the register level. We try all possible combinations of register tile sizes with values of 1,2, and 4. In Table 4, we show the the best execution times of tiled codes that are enhanced using unrolling and scalar replacement for PrimeTile and HiTLOG (rows corresponding to tiling methods with suffix “(regtile)”). Pluto provides loop unroll-and-jam facility that is adjustable using `--ufactor=<factor>` optimization flag. We use unroll factors ranging from 1 to 10 to generate unroll-and-jammed code from Pluto. However, Pluto does not apply scalar replacement optimization. Hence, for fair comparison with Pluto, we also present the performance results of register tiled code without scalar replacement, for PrimeTile and HiTLOG (rows in Table 4 corresponding to tiling methods with suffix “(unroll)”). The best tile sizes and best unroll factors used to obtain the optimized code in Table 4 are given in Table 5 and Table 6, respectively.

For imperfectly nested loops, the performance of tiled code generated by Pluto is comparable to that generated by PrimeTile. For perfectly nested loops, the performance results demonstrate that the parametric multilevel tiled code generated by PrimeTile and HiTLOG consistently outperform the fixed multilevel tiled code

	LU	2D FDTD	1D Jacobi	Cholesky	TriSolver	Seidel	DSYRK	DTRMM
Pluto	12.3	68.0	26.8	40.7	30.5	108.6	36.7	38.7
PrimeTile(n)	11.5	74.4	26.5	40.5	30.9	87.1	24.3	34.6
PrimeTile(f)	11.0	70.3	26.3	38.4	29.3	86.5	23.0	33.1
HiTLOG	-	-	-	-	-	89.1	22.9	34.1
Pluto(unroll/jam)	8.5	61.3	22.1	38.4	30.5	77.0	15.1	27.5
PrimeTile(unroll)	8.0	54.5	18.9	28.9	18.4	75.1	11.5	17.9
PrimeTile(regtile)	6.2	58.6	13.5	25.0	16.7	76.6	16.2	21.9
HiTLOG(unroll)	-	-	-	-	-	78.5	11.4	19.7
HiTLOG(regtile)	-	-	-	-	-	77.8	16.3	23.9

Table 4: Best execution times (in seconds) of different tiling methods on all benchmarks

	LU	2D FDTD	1D Jacobi	Cholesky	TriSolver	Seidel	DSYRK	DTRMM
Pluto	(768,32)	(256,32)	(0,1024)	(1024,32)	(128,16)	(256,8)	(128,16)	(256,16)
PrimeTile(n)	(0,16)	(0,16)	(0,64)	(0,32)	(0,16)	(0,4)	(0,16)	(0,8)
PrimeTile(f)	(256,16)	(128,16)	(768,64)	(128,16)	(128,16)	(128,4)	(128,16)	(128,4)
HiTLOG	-	-	-	-	-	(128,4)	(128,16)	(128,8)

Table 5: Best tile sizes used to generate the codes in Table 4. All tile sizes are specified in the form of (outer tile size, inner tile size). Zero indicates no tiling at that particular level.

from Pluto. The results also indicate that the execution times of tiled code generated by PrimeTile and HiTLOG are almost the same.

Due to the loop unrolling and/or scalar replacement optimizations, the register tiled code performs significantly better then the code that is tiled only for different levels of caches. Also, the best unroll-jammed code from Pluto performs worse compared to the unrolled code from PrimeTile or HiTLOG. Upon examining the code generated by Pluto, it appears that the reason for this may be because Pluto does not separate full tiles and partial tiles at different levels of tiling, prohibiting other potential optimizations such as loop unrolling that can improve register locality.

Figures 5, 6, 7, and 8 show the execution times of tiled code for LU, 2D FDTD, Seidel, and DSYRK, respectively. We show the execution times for an outer tile size that yields the best performance for each tiling approach and five different inner tile sizes for each outer tile size (including the one giving best performance). As shown in the graphs, the performance of PrimeTile(n) code version is the same as that of PrimeTile(f) version for one level tiling cases, since both codes are identical. As outer tile sizes increase in two level tiling cases, PrimeTile(f) version performs better than PrimeTile(n) version. This is because the area of partial tiles in PrimeTile(n) version that are not optimized by tiling becomes larger with higher outer tile sizes. It can be observed in Table 5 that always only one level of tiling produces the best tiled code of PrimeTile(n) version. Furthermore, the inner tile size of the best two-level tiled code of PrimeTile(f) version is in general the same as that of PrimeTile(n). From the graphs, it is clear that for small inner tile sizes, tiled codes generated using PrimeTile(f) are better than Pluto generated fixed tiled codes, and for larger inner tile sizes, they are comparable to the fixed tiled codes generated by Pluto. For perfect nest benchmarks, PrimeTile performs better when inner tile sizes are very small (2 and 4), and the performance becomes comparable to the HiTLOG tiled code as the

	LU	2D FDTD	1D Jacobi	Cholesky	TriSolver	Seidel	DSYRK	DTRMM
Pluto(unroll/jam)	3×3	5×5	6×6	7×7	2×2	6×6	8×8	7×7
PrimeTile(unroll)	$2 \times 1 \times 2$	$4 \times 1 \times 4$	4×1	$4 \times 1 \times 1$	$4 \times 4 \times 1$	$2 \times 1 \times 2$	$4 \times 4 \times 4$	$4 \times 4 \times 2$
PrimeTile(regtile)	$4 \times 4 \times 1$	$1 \times 1 \times 2$	4×4	$4 \times 4 \times 1$	$4 \times 4 \times 1$	$4 \times 4 \times 4$	$1 \times 4 \times 4$	$4 \times 1 \times 4$
HiTLOG(unroll)	-	-	-	-	-	$2 \times 1 \times 2$	$4 \times 4 \times 4$	$4 \times 4 \times 2$
HiTLOG(regtile)	-	-	-	-	-	$2 \times 1 \times 2$	$1 \times 4 \times 4$	$4 \times 1 \times 4$

Table 6: Best unroll factors used to generate the codes in Table 4

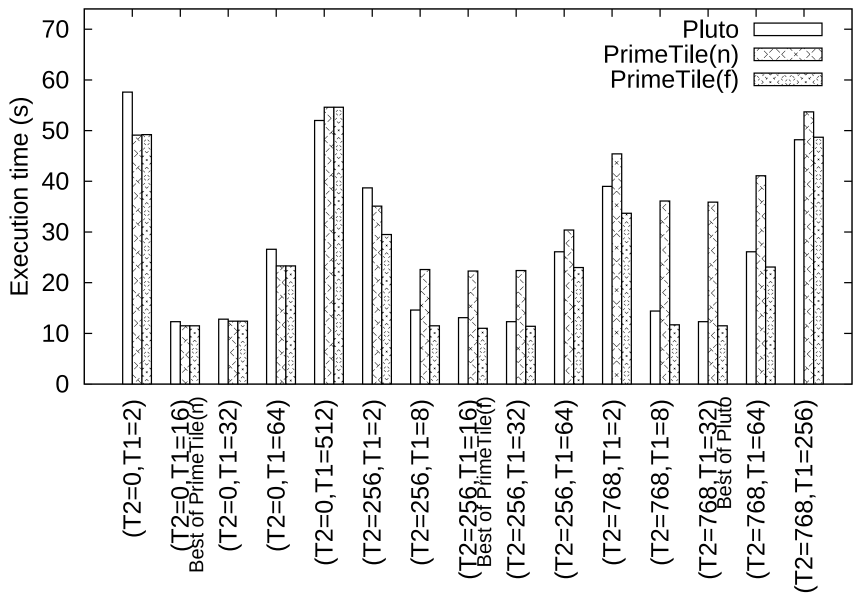


Figure 5: Execution times for LU code generated by Pluto and PrimeTile

inner tile sizes increase. The higher performance difference for smaller tile sizes is due to the higher control overhead caused by the evaluation of min and max expressions and if-conditions in both Pluto and HiTLOG, as explained later in this section.

5.3 Evaluation of Code Quality

In this subsection, we evaluate the quality of the code generated by the various tiling methods using some static and dynamic performance metrics, including the number of lines of code and the dynamic control overhead due to evaluation of if-conditions and max and min expressions. A detailed evaluation on the code quality based on these metrics is presented in Table 7. It can be noted that the code generated using PrimeTile(n) version is as compact as that generated using Pluto and HiTLOG. The expanded code generated by PrimeTile(f) version, however, as discussed earlier in the section, yields better performance. The dynamic control overhead due to if-conditions and max and min expressions is orders of magnitude higher in codes generated by Pluto and HiTLOG than PrimeTile. Pluto does not generate if-conditions for perfect nest cases, but the dynamic count of max and min expressions is very high as in the case of imperfect nests.

6 Discussion and Perspective

In this section, we provide a discussion on the benefits and constraints of the tiling approach that we have developed. Our philosophy has been to work at the same level of separation between the transformation phase and the code generation phase as currently manifested in polyhedral/affine compiler optimization frameworks. Let us consider CLooG as a concrete example of a powerful polyhedral code generator. It takes as input a description of the statement domains and the schedule (scattering functions in CLooG terminology). CLooG generates valid code for any dependence preserving affine schedule, but the dependence information is not explicitly present in the input to CLooG. In other words, the polyhedral transformation phase that precedes the code generation phase works with dependence abstractions in generating a valid schedule, but only provides the generated schedule to CLooG. Thus, CLooG only works with the constraints imposed by the scheduling functions, and the information on the actual data dependences is not available to CLooG.

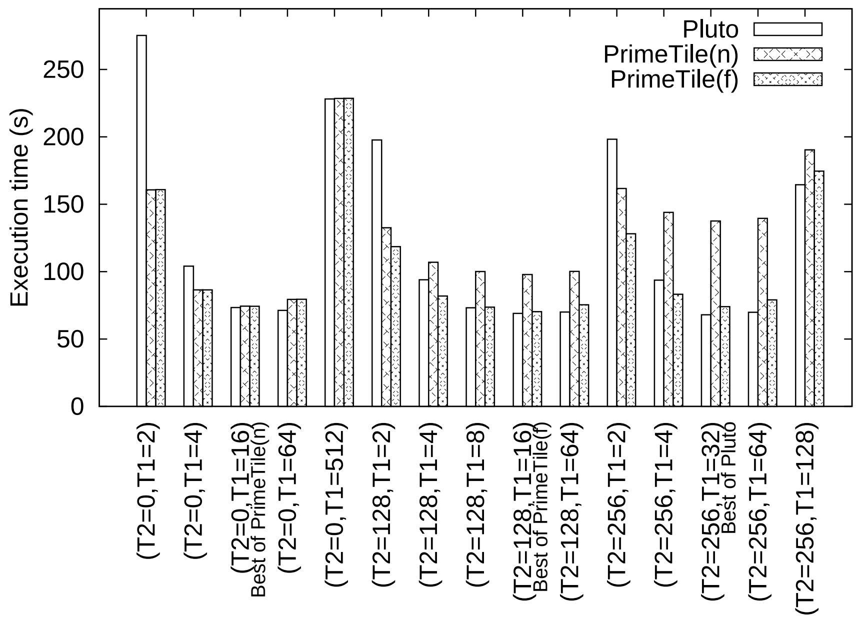


Figure 6: Execution times of 2D FDTD code generated by Pluto and PrimeTile

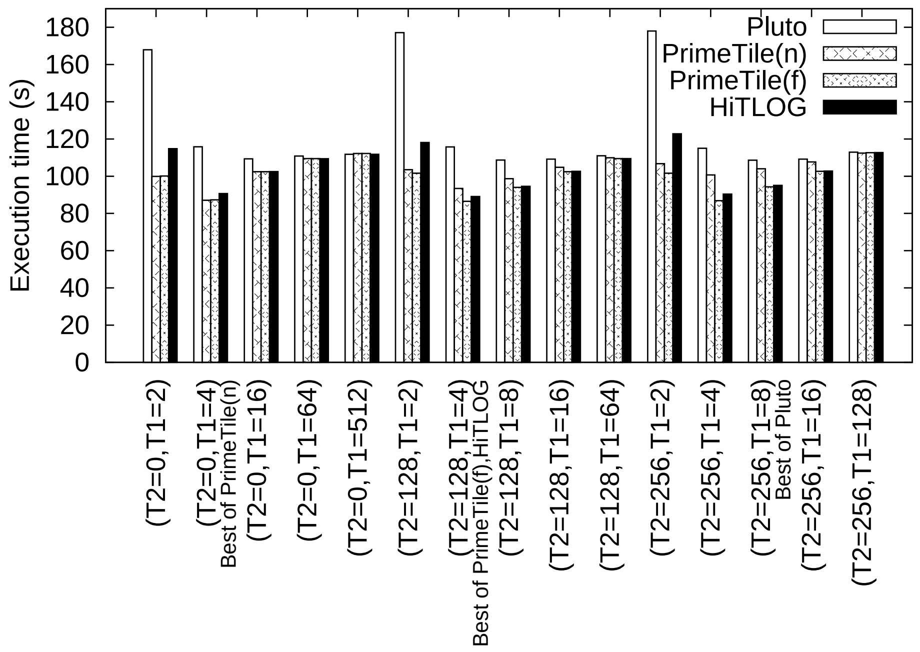


Figure 7: Execution times of Seidel code generated by Pluto, PrimeTile, and HiTLOG

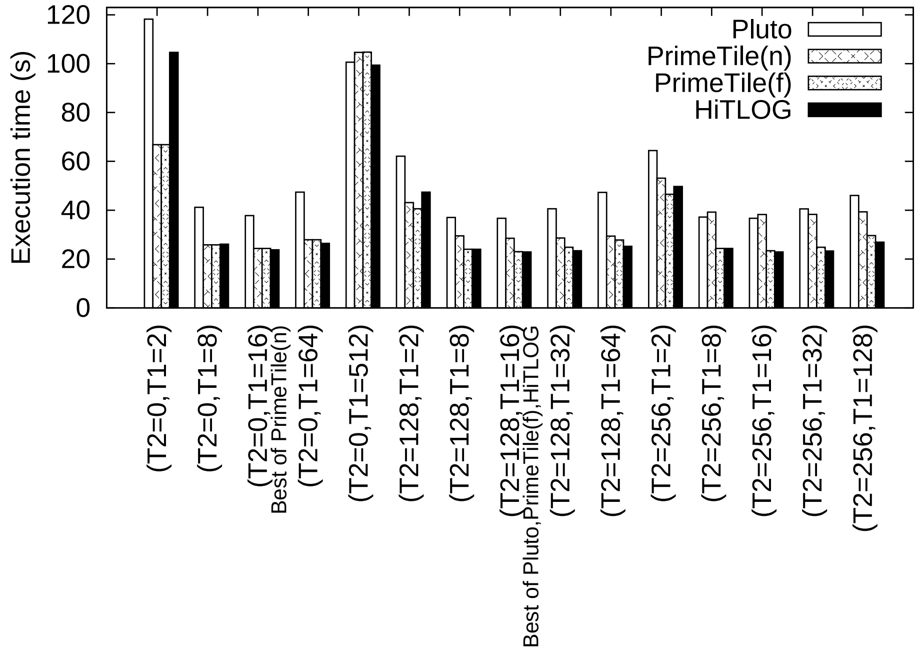


Figure 8: Execution times of DSYRK code generated by Pluto, PrimeTile, and HiTLOG

				Static			Dynamic		
		#lines	#bytes	#mins	#maxs	#ifs	#mins	#maxs	#ifs
LU	Pluto	45	1.6K	13	15	2	1.2G	6.05M	167.4M
	PrimeTile(n)	87	2.8K	0	0	3	0	0	12.2K
	PrimeTile(f)	423	16.9K	0	0	12	0	0	12.2K
2D FDTD	Pluto	397	26.4K	110	93	29	74.5M	40.9M	7.4M
	PrimeTile(n)	226	12.1K	0	0	9	0	0	15.6K
	PrimeTile(f)	682	42.4K	0	0	17	0	0	29.4K
1D Jacobi	Pluto	59	2.2K	12	12	7	35.3M	187.5K	46.9K
	PrimeTile(n)	71	2.1K	0	0	4	0	0	34
	PrimeTile(f)	200	7.7K	0	0	8	0	0	60
Cholesky	Pluto	122	6.2K	17	27	16	722.3M	259.6M	7.2M
	PrimeTile(n)	125	4.2K	0	0	3	0	0	12.2K
	PrimeTile(f)	503	20.6K	0	0	13	0	0	94.6K
TriSolver	Pluto	128	4.9K	21	14	13	1.44G	58.08M	33.4M
	PrimeTile(n)	104	3.2K	0	0	3	0	0	34.97K
	PrimeTile(f)	359	13.7K	0	0	7	0	0	35.5K
Seidel	Pluto	31	2.0K	26	26	1	1.5G	1.6G	1
	PrimeTile(n)	81	4.3K	0	0	3	0	0	249.5K
	PrimeTile(f)	425	25.1K	0	0	14	0	0	465.2K
	HiTLOG	57	1.99K	3	3	3	1.0G	490.0M	197.1M
DSYRK	Pluto	29	843	8	8	0	1.7G	903.6M	0
	PrimeTile(n)	61	1.9K	0	0	1	0	0	187
	PrimeTile(f)	236	9.0K	0	0	4	0	0	210
	HiTLOG	53	1.4K	3	3	1	261.4M	39.0M	3.6M
DTRMM	Pluto	31	899	8	8	1	951.2M	909.5M	1
	PrimeTile(n)	61	1.98K	0	0	2	0	0	140.2K
	PrimeTile(f)	241	9.4K	0	0	6	0	0	562.2K
	HiTLOG	57	1.6K	3	3	3	388.1M	179.9M	32.6M

Table 7: Evaluation of code quality using static and dynamic metrics on all benchmarks for different tiling methods

Our approach to efficient tiling of imperfect loop nests also assumes a similar separation between the transformation phase and code generation phase. The transformation phase passes on a schedule to the code generator, along with a specification of a contiguous band of loops in the target code that is to be tiled. Just as CLooG receives only the schedule functions and not the actual data dependences, in our implemented system, the tiling code generator does not have information about the data dependences. For our experiments, we used affine transforms from the Pluto system as the input to out tiling code generator. However, any schedule could be used, as long as the generalized tiling condition (dependences are lexicographically non-negative in all tiled dimensions) is satisfied by the schedule.

Our choice of modularization and separation of the transformation phase from the code generation phase does mean that the tiling code generator might be forced to be conservative in some circumstances. We illustrate this using some examples.

The key idea behind our tiling approach is that of geometric separation of tiles corresponding to instances of multiple statements. Since no actual dependence information is available, the only assumption that can be made is that the input schedule satisfies the generalized tiling condition, i.e. that all dependences must be lexicographically non-negative in the common embedded space, with respect to all the tiling dimensions. The embedded iteration space may be the original iteration space of the program or an iteration space transformed from the original iteration space according to suitable affine transformations to ensure that the generalized tiling condition is satisfied. The instances of different statements of the program are embedded according to an affine-by-statement scheduling function.

Consider the example described in Figure 9. There are two imperfect loop nests — Code A and Code B. Both the codes have two statements, S1 and S2, S1 being a one-dimensional statement and S2 a two-dimensional statement. The individual statement polyhedra for S1 and S2 are exactly the same for both codes, but the dependences are different. In Code A, there is a data dependence from an instance of S1 to all instances of S2 that have the same outer loop iterator value. In Code B, in addition to the dependence that exists in Code A, there is a dependence from S2 to S1 as shown in Figure 9(f).

It can be seen that Code A is fully tileable along both the i and j dimensions. A tiled version of Code A would first execute a tile of S1 (spanning a range of i values), followed by a set of S2 tiles for the same range of i values), and then another tile of S1 followed by a set of S2 tiles, and so on.

Figure 9(b) shows valid scheduling functions for S1 and S2 in Code A. We use an identity mapping for S2 (map S2(i,j) to point [i,j] in the embedded target space). Two (of many possible) valid scheduling functions are shown for S1. Schedule S1a maps S1(i) to embedded-space point [i,0], while schedule S1b maps S1(i) to [i,i] in the 2D target embedded space. The mapping onto the embedded space is shown in Figure 9(d) for use of schedule S1a with S1. It can be seen that geometric separation of the tiles for S1 and S2 is feasible. In contrast, with schedule S1b (shown in Figure 9(e)), geometric separation of tiles for S1 is not feasible. This is because of the possibility of lexicographically non-negative dependences (such as between S2(i,j) to S1($i+1$)) that would disallow tiling of S1. Given just the schedule (which implies the mapping onto the embedded target space) but no information about the actual dependences, it is impossible to rule out such a possibility. Indeed, Code B provides such an example. As seen in 9(f), S1b is a valid schedule for Code B. In this case, S1a is not a schedule that satisfies the generalized tiling condition since the dependences from S2 to S1 would have a negative component. With Code B, tiling of S1 is in fact not feasible.

The above example illustrates the possibility of conservative tiling decisions because of using a geometric separation of tiles for multiple statements. A monolithic system that both performs polyhedral transformations as well as code generation could possibly be developed that avoids this, but is beyond the scope of this work. We believe that instead it would be preferable to maintain the CLooG-like separation between transformation modules and the code generator, and seek to generate schedules like S1a and not S1b in the first place. It is interesting that the Pluto system generates schedule S1a and not S1b for this example.

```
for  (i=0;i<N;i++) {
  S1: a[i] = a[i] + 5.0;
  for  (j=i; j<N; j++)  {
    S2: b[i][j] = b[i][j] + a[i];
  }
}
```
(a) Code A

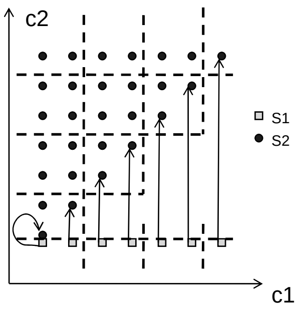
$$S1a \begin{bmatrix} c1 \\ c2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ N \end{bmatrix}$$

$$S1b \begin{bmatrix} c1 \\ c2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ N \end{bmatrix}$$

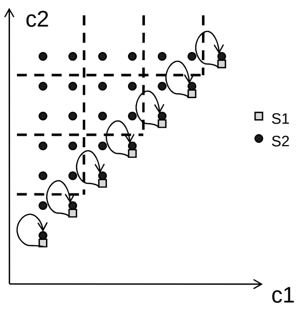
$$S2 \begin{bmatrix} c1 \\ c2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ N \end{bmatrix}$$

(b) Schedules for S1 and S2

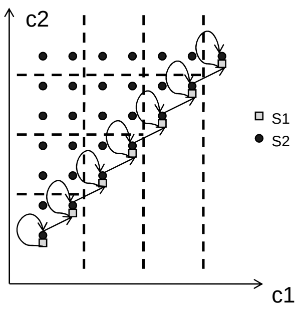
```
for  ( it=0; it<N;i+=Ti) {
 for  (i=it; i<min(it+Ti,N); i++) {
  S1: a[i] = b[i−1][i−1] + 5.0;
 }
  for  (j=i; j<N; j++)  {
    S2: b[i][j] = b[i][j] + a[i];
  }
}
```
(c) Code B



(d) Embedded iteration spaces of S1 and S2 for code A (using schedule S1a for S1)



(e) Embedded iteration spaces of S1 and S2 for code A (using schedule S1b for S1)



(f) Embedded iteration spaces of S1 and S2 for code B (using schedule S1b for S1)

Figure 9: Tiling in embedded transformed space

7 Conclusions

Tiled loops with parameterized tile sizes (not compile time constants) facilitate runtime feedback and dynamic optimizations used in iterative compilation and automatic tuning. Previous parametric multilevel tiling approaches are restricted to perfectly nested loops, where all assignment statements are contained inside the innermost loop of a loop nest. Previous solutions to tiling for imperfect loop nests are limited to the case where tile sizes are fixed. In this paper we have developed an effective approach to parametric multi-level tiling of imperfectly nested affine loops. The key idea behind our tiling algorithm is the use of a geometric approach to multi-statement tile separation by analysis of the AST generated by Quillere's polyhedra scanning algorithm used with schedules that satisfy the generalized tiling condition. The approach is effective in generating loops that traverse over full rectangular tiles that are suited for potential compiler optimizations such as register tiling. We have demonstrated the effectiveness of the developed tiling approach with an extensive experimental evaluation using a number of computational benchmarks.

8 Availability of the Tiling Tool

The PrimeTile software and the modified version of CLooG are available for public download at [1].

Acknowledgment

The authors would like to thank Uday Bondhugula of IBM Research for many productive discussions and his valuable help with Pluto and the writing of this paper. This work was supported in part by the National Science Foundation through awards 0403342, 0509467, and 0811781, and a State of Ohio Development Fund.

References

[1] PrimeTile: A Parametric Multi-Level Tiler for Imperfect Loop Nests. http://primetile.sourceforge.net.

[2] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Proc. Supercomputing (SC 2000)*, 2000.

[3] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *IJPP*, 29(5), Oct. 2001.

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *PPoPP'91*, pages 39–50, 1991.

[5] Workshop on Automatic Tuning for Petascale Systems. http://cscads.rice.edu/workshops/summer08/autotuning.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE PACT*, pages 7–16, Sept. 2004.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[8] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, 1994.

[9] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 114–124, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[10] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization (CGO'05)*, 2005.

[11] CLooG: The Chunky Loop Generator. http://www.cloog.org.

[12] S. Coleman and K. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proc. SIGPLAN '95 Conference on Programming Languages Design and Implementation*, pages 279–290, 1995.

[13] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 20(1):23–53, 1991.

[14] G. Goumas, M. Athanasaki, and N. Koziris. An Efficient Code Generation Technique for Tiled Iteration Spaces. *IEEE Trans. Parallel Distrib. Syst.*, 14(10):1021–1034, 2003.

[15] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau, 2004. Habilitation Thesis.

[16] HiTLoG: Hierarchical Tiled Loop Generator. Available at http://www.cs.colostate.edu/MMAlpha/tiling/.

[17] K. Hogstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *SPAA*, pages 201–211, 1999.

[18] F. Irigoin and R. Triolet. Supernode partitioning. In *PoPL*, pages 319–329, 1988.

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

[20] M. Jiménez, J. M. Llabería, and A. Fernández. A cost-effective implementation of multilevel tiling. *IEEE Trans. Parallel Distrib. Syst.*, 14(10):1006–1020, 2003.

[21] D. Kim, L. Renganarayanan, M. Strout, and S. Rajopadhye. Multi-level tiling: 'm' for the price of one. In *SC*, 2007.

[22] A. Lim. *Improving Parallelism And Data Locality With Affine Partitioning*. PhD thesis, Stanford University, Stanford, CA, USA, 2001.

[23] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN PPoPP*, pages 103–112, 2001.

[24] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ACM ICS*, pages 228–237, 1999.

[25] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998. Extended version of PoPL'97 paper.

[26] Pluto: A polyhedral automatic parallelizer and locality optimizer for multicores. Available at http://pluto-compiler.sourceforge.net.

[27] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, Aug. 1992.

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

[29] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *JPDC*, 16(2):108–230, 1992.

[30] L. Renganarayana, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. In *PLDI'07*, pages 405–414, 2007.

[31] L. Renganarayana and S. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *SC*, 2004.

[32] G. Rivera and C.-W. Tseng. Locality optimizations for multi-level caches. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 2, New York, NY, USA, 1999. ACM.

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

[34] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI*, pages 215–228, 1999.

[35] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. Scalable autotuning framework for compiler optimization. In *23rd IEEE International Parallel & Distributed Processing Symposium Rome, Italy, Italy*, May 2009.

[36] TLoG: A Parametrized Tiled Loop Generator. Available at http://www.cs.colostate.edu/MMAlpha/tiling/.

[37] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. Supercomputing '98*, 1998.

[38] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing Journal*, 2000.

[39] R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *ICPP*, pages 89–98, 2005.

[40] J. Xue. *Loop tiling for parallelism.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[41] Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *J. Supercomput.*, 27(3):219–264, 2004.