Parametric Multi-Level Tiling of Imperfectly Nested Loops

Albert Hartono, Muthu Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, P. Sadayappan

Parametric Multi-Level Tiling of Imperfectly Nested Loops

Albert Hartono
The Ohio State University

Muthu Manikandan Baskaran
The Ohio State University

Cédric Bastoul
Paris-Sud 11 University and HiPEAC Network

Albert Cohen
INRIA Saclay and HiPEAC Network

Sriram Krishnamoorthy
Pacific Northwest National Laboratory

Boyana Norris
Argonne National Laboratory

J. Ramanujam
Louisiana State University

P. Sadayappan
The Ohio State University

ABSTRACT

Tiling is a crucial loop transformation for generating high performance code on modern architectures. Efficient generation of multi-level tiled code is essential for maximizing data reuse in systems with deep memory hierarchies. Tiled loops with parametric tile sizes (not compile-time constants) facilitate runtime feedback and dynamic optimizations used in iterative compilation and automatic tuning. Previous parametric multi-level 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 only handled fixed tile sizes. In this paper, we present an approach to parametric multi-level tiling of imperfectly nested loops. The tiling technique generates loops that iterate over full rectangular tiles, making them amenable to compiler optimizations such as register tiling. Experimental results using a number of computational benchmarks demonstrate the effectiveness of the developed tiling approach.

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

General Terms: Algorithms, Design, Performance

Keywords: Parametric tiling, Imperfectly nested loops

1. INTRODUCTION

Tiling is a crucial transformation for achieving high performance, especially for systems 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 [9, 14, 20, 33, 35–37, 44]. 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. The condition for permutability of a band of loops is that all correspondingly permuted dependence vectors must have non-negative elements.

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

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 tiling imperfectly nested loops [3, 27–29] 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. There are a number of ways of embedding with significantly varying performance impact; we are not aware of any work that has developed effective heuristics for this problem. An effective approach for tiling of imperfectly nested loops was recently developed in the Pluto polyhedral transformation framework [8].

All of these works generate tiled code where the tile sizes are fixed. Since the performance of tiled code varies greatly with the choice of tile sizes, it is preferable to specify the tile sizes as runtime parameters in the code. In such cases, *parametric tiling* refers to the generation of tiled code where tile loops are runtime parameters. For example, such an approach would enable empirical search for tile sizes.

The importance of parametric tiling is exemplified by the highly successful ATLAS [41, 42] 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. But the ATLAS system can only tune BLAS kernels. Further, it 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, 11, 39, 43]. An efficient parametric tiling tool is extremely valuable for generating input tiled codes for such empirical tuning systems.

An innovative scheme for efficient parametric tiling of arbitrary polyhedral statement domains for affine computations was developed by Renganarayana et al. [34] and extended to efficiently generate multi-level parametric tiled code [25]. However, this work is only applicable to perfect loop nests. While the powerful tiling approach of Pluto [8] can handle arbitrarily nested loops for affine computations, it can only generate tiled code with fixed tile sizes.

In this paper we develop an effective approach to parametric multi-level tiling of imperfectly nested affine loops. The key to the approach described in this paper is the exploitation of the power and effectiveness of the algorithm by Quilleré et al. [6, 7, 32] for generating imperfectly nested code to scan unions of polyhedra, using input scattering functions (affine schedules for the different

statements) satisfying a generalized tiling condition, along with the development of a bounds-based approach to generation of parametric tiles by post-processing the abstract syntax tree (AST) of the loop structure generated by the CLooG code generator.

The paper is structured as follows. Section 2 discusses various previous efforts on tiling. Section 3 explains the key ideas behind our approach. A detailed discussion of the algorithm is presented in Section 4. Section 5 provides experimental results and conclusions are presented in Section 6.

2. RELATED WORK

Several previous efforts have addressed the tiling of loops involving arbitrary polyhedral statement domains. Before presenting our approach, we provide some background information about the current state of the art.

2.1 Code Generation for Polyhedral Models

There has been significant progress over the last two decades in 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, 8, 15, 17, 26, 31], 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, 7, 32] have made polyhedral approaches very powerful for transforming affine loop-based code. CLooG [6, 7, 13] is a powerful state-of-the-art code generator that is widely used. The code generation algorithm used in CLooG [6, 7] is based on the one developed by Quilleré, Rajopadhye and Wilde [32] (abbreviated hereafter as QRW). The key idea of the QRW algorithm is to recursively generate a sequence of loop nests scanning several unions of polyhedra by separating them into disjoint polyhedra and generating the corresponding loops from the outermost to the innermost levels.

2.2 Parametric Tiling of Perfect Loop Nests

Recent work from Colorado State University [19, 25, 34, 40] has addressed parametric multi-level tiling of perfect loop nests using the polyhedral model. The tiled code generator TLOG [40] 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 [34]. A similar approach to decomposing the problem of tiled code generation into the above mentioned subproblems and scanning of the tile origins using polyhedral techniques was earlier proposed by Goumas et al. [16]. However, the approach of Goumas et al. only handles fixed tile sizes. TLOG performs rectangular tiling, and hence 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 is very efficient and that the quality of the code generated is very good [34]. Kim et al. [25] generalized the TLOG algorithm to develop HiTLOG [19] that can generate multi-level parametric tiled code for perfectly nested loops. 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. [22, 23] addressed parametric tiled code generation for non-rectangular iteration spaces. The code generated using that approach can suffer from significant code expansion, but involves low overhead to scan through the full tiles in the code.

2.3 Non-parametric Tiling of Imperfect Loop Nests

Ahmed et al. [2, 3] were among the first to propose an approach for tiling imperfectly nested loops. Their 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 chosen 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. Post-processing to hoist and/or eliminate guards is needed in order to create efficient final code. Lim et al. [29] used an affine partitioning framework for tiling; the framework was built on an affine partitioning algorithm they proposed earlier for minimizing synchronization and maximizing parallelism [27, 28]. Recently, Chen et al. [12] have developed a script-based compositional framework for transformations. The system can be used to tile imperfectly nested loops. The framework requires tile sizes to be specified constants. Some specialized works [10, 38, 45] exist for tiling a restricted class of imperfectly nested loops. Parametric tiling is not considered in any of these works.

Bondhugula et al. developed Pluto [8, 30], a system for tiling arbitrary collections of imperfectly nested affine loops. Pluto finds tiling hyperplanes that are targeted at 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 that was originally presented for perfectly nested loops by Irigoin and Triolet [21]. Pluto also requires tile sizes to be fixed for code generation.

Kim and Rajopadhye [24] recently developed a non-polyhedral approach to parametric tiling of loop nests. The approach has polynomial time complexity and is practically efficient. However, the presentation of the work indicates that some manual steps have to be used to properly handle scenarios where issues of tiling validity arise.

3. OVERVIEW OF APPROACH

We 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 imperfectly nested loops can be tiled, followed by a sketch of our approach to separation of tiles for imperfectly nested loops.

3.1 Parametric Tiling for a Single Statement Domain

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. 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 shown in Figure 1(e), corresponding to the perfect loop nest in this example. Since loop i is outermost, strip mining or tiling this loop is straightforward (i.e., to partition the loop i's iteration space into smaller blocks whose size is determined by the tile size parameter T_i). Figure 1(e) shows the partitioning of the iteration space

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

(a) Original perfect loop nest

```
/* full tiles i */
for (it=lbi; it<=ubi−(Ti−sti); it+=Ti) {
  /* code tiled along dimensions i and j */
  // ... omitted ...
}
/* epilog i */
for (i=it; i<=ubi; i+=sti)
  for (j=lbj(i); j<=ubj(i); j+=stj)
    S(i,j);
```

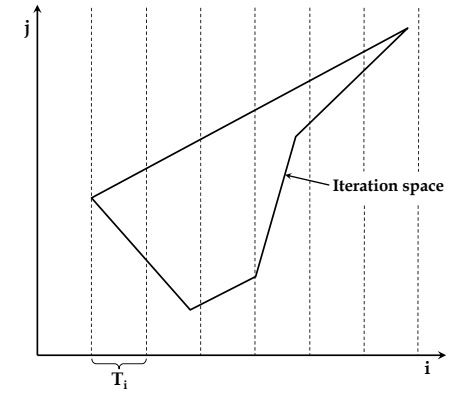
(b) After tiling loop i

```
for it {
  [compute lbv,ubv]
  if (lbv<=ubv) {
    [prolog j]
    [full tiles j]
    [epilog j]
  } else
    [untiled j]
}
[epilog i]
```

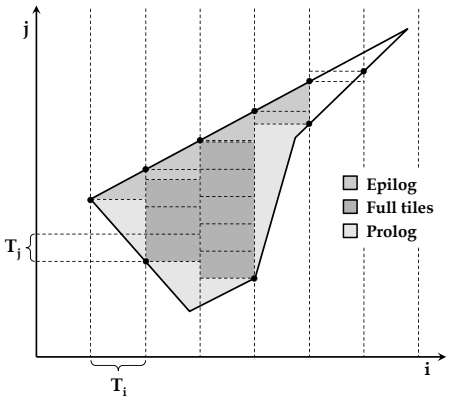
(c) After tiling loops i and j

```
/* full tiles i */
for (it=lbi; it<=ubi−(Ti−sti); it+=Ti) {
  /* compute lbv,ubv */
  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);
    /* full tiles 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);
```

(d) Detailed parametric tiled code



(e) Iteration space (tiled along dimension i)



(f) Iteration space (tiled along dimensions i and j)

Figure 1: Parametric tiling of a perfectly nested loop

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 T_i). 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 T_i.

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 and end of full tiles, but as discussed later, the actual implementation identifies tile boundaries directly from affine loop bounds by evaluating the bound functions at corner points of the outer tile extents. The approach 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 ij-tile is found. If lbv is greater than ubv, no full tiles exist over this i-tile range. In Figure 1(f), this is the case for the last full i-tile segment. 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 is equal to 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.

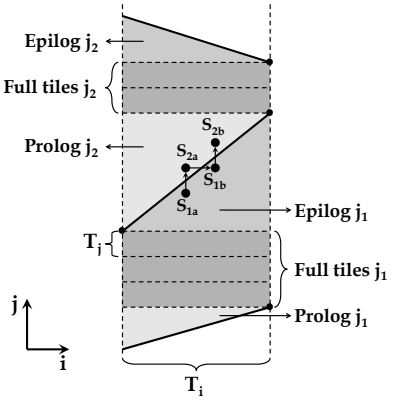
The structure of the tiled code is shown in abstracted pseudocode in Figure 1(c), and with explicit detail in Figure 1(d). At each level of nesting, for a tile range determined by the outer tiling loops, the lbv and ubv values are computed. If ubv is less than lbv, an untiled version of the code is used. If lbv is less than or equal to ubv, the executed code has three parts: a prolog for j values up to $lbv - st_j$ (where st_j is the loop stride in the j dimension), an epilog for j values greater than or equal to j_t (where j_t is the inter-tile loop iterator in the 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.

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 [21]. This condition states that all dependences are either along the tiling hyperplane or enter it from the same side. 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 $(\vec{s}, \vec{t})$ [8]. 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

```
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) Original imperfect loop nest



(b) One full tile segment along i dimension

```
for it {
  [compute lbv1,ubv1]
  if (lbv1<=ubv1) {
    [prolog j1]
    [full tiles j1]
    [compute lbv2,ubv2]
    if (lbv2<=ubv2) {
      [epilog j1 + prolog j2]
      [full tiles j2]
      [epilog j2]
    } else
      [epilog j1 + untiled j2]
  } else {
    [compute lbv2,ubv2]
    if (lbv2<=ubv2) {
      [untiled j1 + prolog j2]
      [full tiles j2]
      [epilog j2]
    } else
      [untiled j1 + untiled j2]
  }
}
[epilog i]
```

(c) Parametric tiled code

```
/* full tiles i */
for (it=lbi; it<=ubi-(Ti-sti); it+=Ti) {
  // ... omitted ...
  if (lbv1<=ubv1) {
    // ... omitted ...
    if (lbv2<=ubv2) {
      /* [epilog j1 + prolog j2] */
      for (i=it; i<=it+(Ti-sti); i+=sti) {
        for (j1=j1t; j1<=ubj1(i); j1+=stj1)
          S1(i,j1);
        for (j2=lbj2(i); j2<=lbv2-stj2; j2+=stj2)
          S2(i,j2);
      }
      // ... omitted ...
    } else
      // ... omitted ...
  } else {
    // ... omitted ...
    if (lbv2<=ubv2) {
      // ... omitted ...
    } else
      /* [untiled j1 + untiled j2] */
      for (i=it; i<=it+(Ti-sti); 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);
      }
  }
}
/* epilog i */
// ... omitted ...
```

(d) Partially detailed parametric tiled code

Figure 2: Parametric tiling of an imperfectly nested loop

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 development of the QRW algorithm. Its refinement and implementation in CLooG is now widely used for generating code for multi-statement domains. The Pluto system uses CLooG for generating tiled code for imperfectly nested loop programs. However, Pluto cannot generate parametric tiled code. The key idea behind our new approach to parametric tiling of imperfect loop-nests is to combine the power of the QRW algorithm (for sorting and separating polyhedra corresponding to multiple-statement domains) with a bounds-based approach to separating tiles, using the AST structure generated by the QRW 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 QRW algorithm is scanned to generate the tiled code structure as described in the next sub-section.

3.3 Separation of Tiles for Overlapping Statement Domains

We use a simple imperfectly nested loop example shown in Figure 2(a) to illustrate the approach to 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 QRW 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 QRW 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 S_1 and S_2 have been separated out in the point-wise code by the QRW algorithm, under what conditions can we also separate out tiles corresponding to these two statements? Our approach to addressing this question is to use lower and upper bound values for the two statements (computed in a similar manner to the perfect-nest example shown earlier), and exploit the fact that all dependences are lexicographically non-negative in all the tiling dimensions (due to satisfaction of the generalized tiling condition).

Consider the example shown in Figure 2(a) as an imperfectly nested loop structure generated by the QRW algorithm. Figure 2(b) depicts the corresponding statement domains of S_1 and S_2 within one full tile segment along the i dimension. Since lbv_1 is less than or equal to ubv_1, we have a separable set of tiles for S_1, and since lbv_2 is less than or equal to ubv_2, there are also separable tiles for S_2. The epilog of S_1 and prolog of S_2 need to be combined and interleaved to ensure satisfaction of any dependences between S_1 to S_2 or vice versa. The staircase-like dependences between statement instances S_{1a}, S_{1b}, S_{2a} and S_{2b} shown in Figure 2(b) exemplify the need for combining and interleaving the epilog of S_1 and the prolog of S_2. The pseudocode in Figure 2(c) shows the different possible cases to be considered and the code correspond-

```
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++) {
  B[1]=(A[1+1]+A[1]+A[1−1])/3;
  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

Figure 3: Example of pre-processing: 1D Jacobi code

ing to the four combinations. Details of some of the combined and interleaved loops can be seen in Figure 2(d).

4. PARAMETRIC TILING ALGORITHM

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

1. Pre-processing: Extraction of statement polyhedra, and generation of a valid affine schedule where a band of the scheduling functions satisfies the generalized tiling condition;
2. Use of the QRW algorithm to scan the statement polyhedra and to generate tileable loop code, with preservation of complete embedding information;
3. Parsing of the loop code to build rectangularly tileable loop ASTs;
4. Recursive traversal of the ASTs for parametric tiling;
5. Conversion of the parametrically tiled loop ASTs into target code.

4.1 Pre-processing and AST Generation

The tiling algorithm takes as input a sequence of arbitrarily nested loops with loop bounds and array access expressions that are affine functions of outer loop variables and program parameters. First, the imperfect loop nest is made rectangularly tileable, using suitable transformations such as skewing. The required transformations are captured in the form of scattering functions (or affine scheduling functions) provided to CLooG. While any suitable scheduling functions that satisfy the generalized tiling condition (described in Section 3.2) may be used, for our experiments described later, we have used the scattering functions generated by Pluto [8].

The second step in the tiling procedure is the generation of a loop code for the imperfectly nested loop structure by application of the QRW polyhedral scanning algorithm. We use an adaptation of the implementation of the QRW algorithm in CLooG to ensure that all embedding information for all statements is explicitly preserved in the generated loop code. 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 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 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.

Figure 3 illustrates the pre-processing steps and the effect of the adaptation of CLooG to create redundant loops for explicit representation of complete embedding information for all statement domains. Figure 3(a) shows the code for 1D Jacobi stencil computation. It is not tileable due to data dependences with negative components. Skewing can be used to eliminate these backward dependences and make rectangular tiling valid. Figure 3(b) shows the affine transformation generated by Pluto, used to enable rectangular tiling. Figure 3(c) shows the output code after the transformation is processed using CLooG. We observe 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).

The output code of the adapted CLooG is then parsed into loop ASTs, which become the input of the transformation module that implements the tiling algorithm described in the following subsection.

4.2 Tiled Loop Generation Algorithm

We first present the tiling algorithm for single-level tiling. The algorithm (Figure 4) has a tail-recursive structure. It takes as input a sequence of arbitrarily nested loops and returns a sequence of transformed loops that scan the tiled iteration space.

As broadly discussed earlier in Section 3, the algorithm performs a *depth-first traversal* of the input ASTs (i.e., dimension by dimension), to decompose each loop AST node into a sequence of loops: a prolog loop, a full-tile loop, and an epilog loop.

The algorithm for generating inter-tile and intra-tile loops is provided in Figure 5. In the pre-processed 1D Jacobi code shown in Figure 3(d) for example, the inter-tile loop of the outermost loop t is for (tt=0; tt<=T-1-(Tt-1); tt+=Tt), and its intra-tile loop is for (t=tt; t<=tt+(Tt-1); t+=1), where t_t and T_t are the inter-tile loop iterator and the tile size parameter corresponding to loop t, respectively.

It is possible that a tiled statement polyhedron has no full tiles along a dimension. Hence, at each dimension of the iteration space, we need to find the start and end points of possible full tiles for the tile ranges determined by the outer tiling loops. In programs containing affine loop nests, the determination of the start and end of full tiles can be done statically as illustrated in Figure 6. We utilize the fact that the boundary values of the full tiles of the outer dimensions are already known and that the loop bounds corresponding to

TILE: Generate parametric tiled loops (main procedure).

Input: Input ASTs: $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$
Output: Transformed ASTs

Return **AUXTILE**$((\mathcal{T}_1, \ldots, \mathcal{T}_n), (), (), ())$

AUXTILE: Generate parametric tiled loops (auxiliary tail-recursive procedure).

Input: Input ASTs: $(\mathcal{T}_{I_1}, \ldots, \mathcal{T}_{I_p})$; Outer loops: $(\mathcal{L}_1, \ldots, \mathcal{L}_q)$; Transformed ASTs: $(\mathcal{T}_{T_1}, \ldots, \mathcal{T}_{T_r})$; Partially transformed ASTs: $(\mathcal{T}_{P_1}, \ldots, \mathcal{T}_{P_s})$
Output: Transformed ASTs

1. *Termination condition (base case)*:
   If the input AST list is empty, construct $\mathcal{L} \leftarrow (\mathcal{T}_{P_1}, \ldots, \mathcal{T}_{P_s})$ enclosed with intra-tile loops $\mathcal{L}_{intratile_1}, \ldots, \mathcal{L}_{intratile_q}$ that were obtained from **GETINTRA**$(\mathcal{L}_i)$ for $1 \leqslant i \leqslant q$. Then return $(\mathcal{T}_{T_1}, \ldots, \mathcal{T}_{T_r}, \mathcal{L})$.
2. *Handling of non-loop statement*:
   If $\mathcal{T}_{I_1}$ is a not a loop, then return **AUXTILE**$((\mathcal{T}_{I_2}, \ldots, \mathcal{T}_{I_p}), (\mathcal{L}_1, \ldots, \mathcal{L}_q), (\mathcal{T}_{T_1}, \ldots, \mathcal{T}_{T_r}), (\mathcal{T}_{P_1}, \ldots, \mathcal{T}_{P_s}, \mathcal{T}_{I_1}))$.
3. *Generation of full-tile-start and full-tile-end statements*:
   Generate an assignment statement $\mathcal{S}_{lbv} \leftarrow lbv = \mathcal{E}_{lb}$; to determine the start of full tiles, and an assignment statement $\mathcal{S}_{ubv} \leftarrow ubv = \mathcal{E}_{ub}$; to determine the end of full tiles, where $(\mathcal{E}_{lb}, \mathcal{E}_{ub}) \leftarrow$ **FINDLBVUBV**$(\mathcal{T}_{I_1}, (\mathcal{L}_1, \ldots, \mathcal{L}_q))$.
4. *Generation of prolog loop*:
   Construct $\mathcal{L} \leftarrow$ a copy of loop $\mathcal{T}_{I_1}$ with its upper bound expression being replaced with $lbv - st$, where st is the loop stride of $\mathcal{T}_{I_1}$. Then generate $\mathcal{L}_{prolog} \leftarrow (\mathcal{T}_{P_1}, \ldots, \mathcal{T}_{P_s}, \mathcal{L})$ enclosed with intra-tile loops $\mathcal{L}_{intratile_1}, \ldots, \mathcal{L}_{intratile_q}$ that were obtained from **GETINTRA**$(\mathcal{L}_i)$ for $1 \leqslant i \leqslant q$.
5. *Generation of full-tile loop and its children*:
   Recurse to the next dimension to get the transformed child ASTs $(\mathcal{T}'_{C_1}, \ldots, \mathcal{T}'_{C_m}) \leftarrow$ **AUXTILE**$((\mathcal{T}_{C_1}, \ldots, \mathcal{T}_{C_n}), (\mathcal{L}_1, \ldots, \mathcal{L}_q, \mathcal{T}_{I_1}), (), ())$, where $\mathcal{T}_{C_1}, \ldots, \mathcal{T}_{C_n}$ are the child ASTs of $\mathcal{T}_{I_1}$. Construct $\mathcal{L} \leftarrow$ a copy of loop $\mathcal{T}_{I_1}$ with lbv and ubv as its lower and upper bounds (respectively). Then generate $\mathcal{L}_{fulltile} \leftarrow (\mathcal{T}'_{C_1}, \ldots, \mathcal{T}'_{C_m})$ enclosed with $\mathcal{L}_{intertile}$ that were obtained from **GETINTER**$(\mathcal{L})$.
6. *Generation of epilog loop*:
   Generate $\mathcal{L}_{epilog} \leftarrow$ a copy of loop $\mathcal{T}_{I_1}$ with its lower bound expression being replaced with i_t where i_t is the iterator of $\mathcal{L}_{intertile}$. Note that $\mathcal{L}_{epilog}$ is only *partially transformed* since at this point it has not been enclosed with the intra-tile loops of $\mathcal{L}_1, \ldots, \mathcal{L}_q$.
7. *Optimization for no opening-boundary tiles*:
   If the lower bound expression of $\mathcal{T}_{I_1}$ is free of any loop iterators of $\mathcal{L}_1, \ldots, \mathcal{L}_q$, then return
   AUXTILE$((\mathcal{T}_{I_2}, \ldots, \mathcal{T}_{I_p}), (\mathcal{L}_1, \ldots, \mathcal{L}_q), (\mathcal{T}_{T_1}, \ldots, \mathcal{T}_{T_r}, \mathcal{L}_{prolog}, \mathcal{L}_{fulltile}), (\mathcal{L}_{epilog}))$.
8. *Generation of if-there-are-full-tiles condition and statement blocks for the true/false branches*:
   Generate an if-statement $\mathcal{S}_{if} \leftarrow$ if $(lbv \leqslant ubv)$ { $\mathcal{T}_{t_1}; \ldots; \mathcal{T}_{t_v}$; } else { $\mathcal{T}_{f_1}; \ldots; \mathcal{T}_{f_w}$; }, where:
   $(\mathcal{T}_{t_1}, \ldots, \mathcal{T}_{t_v}) \leftarrow$ **AUXTILE**$((\mathcal{T}_{I_2}, \ldots, \mathcal{T}_{I_p}), (\mathcal{L}_1, \ldots, \mathcal{L}_q), (\mathcal{L}_{prolog}, \mathcal{L}_{fulltile}), (\mathcal{L}_{epilog}))$ and
   $(\mathcal{T}_{f_1}, \ldots, \mathcal{T}_{f_w}) \leftarrow$ **AUXTILE**$((\mathcal{T}_{I_2}, \ldots, \mathcal{T}_{I_p}), (\mathcal{L}_1, \ldots, \mathcal{L}_q), (), (\mathcal{T}_{P_1}, \ldots, \mathcal{T}_{P_s}, \mathcal{T}_{I_1}))$.
9. *Ordering of the generated statements*:
   Return $(\mathcal{T}_{T_1}, \ldots, \mathcal{T}_{T_r}, \mathcal{S}_{lbv}, \mathcal{S}_{ubv}, \mathcal{S}_{if})$.

Figure 4: Parametric tiling algorithm

the dimension are affine functions of the loop iterators of the outer dimensions. As an example, the start and end of full tiles in the second inner loop of the pre-processed 1D Jacobi code (Figure 3(d)) are 2*(tt+(Tt-1))+2 and 2*(tt)+N-2 respectively, where t_t is the inter-tile loop iterator of the outer loop t and T_t is the corresponding tile size variable. A lower bound can have multiple affine expressions contained in a max function (or a min function in case of upper bound). In such a case, we process each expression independently.

The tail-recursive procedure begins with the following input: (1) a list of input ASTs at the same nesting level that need to be tiled, (2) a sequence of outer loops ordered from outermost to innermost, (3) all preceding ASTs (at the same nesting depth) that have previously been transformed, and (4) all preceding ASTs that have not been fully transformed (described in details later). The output of the recursive procedure is a sequence of transformed loops that scan a set of tiles whose union covers all points in the iteration space corresponding to all given loop nests, for the ranges of the outer tiles. In the algorithm, for a given loop at one dimension, the start and end of possible full tiles (lbv and ubv respectively) are statically computed in Step 3. The two bounds are subsequently used to split the given loop into prolog, full-tile loop, and epilog (in Steps 4, 5 and 6 respectively). The prolog spans over the loop iteration points up to $lbv - st$ (where st is the stride of the loop). The full-tile loop covers the points inside all full tiles between lbv and ubv. The epilog enumerates the remaining iteration points.

The runtime check generated in Step 8 detects the presence of full tiles. If lbv is less than or equal to ubv, then full tiles can possibly exist and therefore a full-tile loop (preceded by a prolog and succeeded by an epilog) will be executed; otherwise, an untiled version of the loop is used. Two tail-recursive calls are made to produce a tiled version and an untiled version of the given loop. Each generated loop version then becomes a separate statement block of the constructed if-statement. Applying the tiling algorithm to a simple 2D loop nest that contains n inner loops will yield a tiled code with a total of 2^n possible tile cases. For instance, the 2D imperfect loop nest shown in Figure 2 has two inner loops and its tiled code contains four different tile cases. The last step of the tiling algorithm (Step 9) places the generated if-statement after the two assignment statements used to compute lbv and ubv.

When the lower bound expression of a loop has no references to the outer loop iterators, the value of lbv will always be constant and identical to the original lower bound. So, lbv computation becomes unnecessary. This also implies that the tiled iteration space has no opening boundary tiles (i.e., no prolog). Consequently, the if-conditional statement need not be produced since lbv is always less than or equal to ubv and the loop iteration space can always

GETINTER: Build an inter-tile loop.
GETINTRA: Build an intra-tile loop.

Input: Input loop AST: $\mathcal{T}$
Output: Inter-tile loop of $\mathcal{T}$ (for **GETINTER**); Intra-tile loop of $\mathcal{T}$ (for **GETINTRA**)

1. Generate an inter-tile loop iterator i_t and a tile size parameter T_i.
2. $\mathcal{L}_{intertile} \leftarrow$ for (i_t=lb; $i_t \leqslant ub$-(T_i-st); i_t+=T_i)
   $\mathcal{L}_{intratile} \leftarrow$ for (i=i_t; $i \leqslant i_t$+T_i-st; i+=st)
   where i, lb, ub, st are the iterator, the lower bound, the upper bound and the stride of loop $\mathcal{T}$ (respectively).
3. Return $\mathcal{L}_{intertile}$ for **GETINTER**.
   Return $\mathcal{L}_{intratile}$ for **GETINTRA**.

Figure 5: Generation of inter-tile loop and intra-tile loop

FINDLBVUBV: Find the lower and upper bounds of full-tile loop.

Input: Input loop AST: $\mathcal{T}$; Outer loops: $(\mathcal{L}_1, \ldots, \mathcal{L}_n)$
Output: Start and end points of possible full tiles in loop $\mathcal{T}$

1. $\mathcal{E}_{lb} \leftarrow$ a copy of the lower bound expression of $\mathcal{T}$
   For each outer loop iterator i in the affine expression $\mathcal{E}_{lb}$:

   Let $\mathcal{L}$ be the outer loop in $\mathcal{L}_1, \ldots, \mathcal{L}_n$ that has an iterator i. If the coefficient of i is positive, then substitute i with the upper bound of the inter-tile loop of $\mathcal{L}$. Otherwise, substitute i with the lower bound of the inter-tile loop of $\mathcal{L}$.

2. $\mathcal{E}_{ub} \leftarrow$ a copy of the upper bound expression of $\mathcal{T}$
   For each outer loop iterator i in the affine expression $\mathcal{E}_{ub}$:

   Let $\mathcal{L}$ be the outer loop in $\mathcal{L}_1, \ldots, \mathcal{L}_n$ that has an iterator i. If the coefficient of i is positive, then substitute i with the lower bound of the inter-tile loop of $\mathcal{L}$. Otherwise, substitute i with the upper bound of the inter-tile loop of $\mathcal{L}$.

3. Return $(\mathcal{E}_{lb}, \mathcal{E}_{ub})$.

Figure 6: Static determination of the start and end of full tiles

be partitioned into full and partial tiles using a full-tile loop and an epilog (Step 7). Similarly in the case of a loop with an upper bound that is free of outer loop iterators, ubv computation is not needed; but, the epilog still needs to be generated for enumerating the iteration points that have not been executed by the full-tile loop.

The tail-recursive tiling procedure takes two auxiliary parameters: transformed ASTs and partially transformed ASTs, both used to accumulate the tiling result. A transformed AST is one that is already enclosed with intra-tile loops that correspond to the outer loops. A partially transformed AST means that its enclosure with the intra-tile loops of the outer loops is deferred to the next few recursions when there are no more ASTs at the current nesting level (Step 1), or when the prolog of the succeeding loop gets formed (Step 4). Such a deferred enclosure enables the combining and interleaving of all contiguous prolog, epilog and untiled loop, to ensure that any data dependences across distinct statement domains are not violated.

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

A multi-level tiled loop is a loop nest where tiling is applied multiple times to create different levels of hierarchical tiles, with each lower-level tile nested in the one above. Multi-level tiling is important for exploiting data locality in deep memory hierarchies.

GETINTER: Build an outermost-level inter-tile loop.
GETINNERINTERINTRA: Build inner-level inter-tile loops and an intra-tile loop.
GETINTRA: Build an intra-tile loop corresponding to the outermost-level inter-tile loop.

Input: Input loop AST: $\mathcal{T}$; Number of levels of tiling: n.
Output: Outermost-level inter-tile loop of $\mathcal{T}$ (for **GETINTER**); All inner-level inter-tile loops and intra-tile loop of $\mathcal{T}$ (for **GETINNERINTERINTRA**); Intra-tile loop that corresponds to the outermost-level inter-tile loop of $\mathcal{T}$ (for **GETINTRA**)

1. Generate inter-tile loop iterators $i_{t_1}, \ldots, i_{t_n}$ and tile size parameters $T_{1_i}, \ldots, T_{n_i}$.
2. $\mathcal{L}_{intertile_n} \leftarrow$ for (i_{t_n}=lb; $i_{t_n} \leqslant ub$-(T_{n_i}-st); i_{t_n}+=T_{n_i})
   $\mathcal{L}_{intertile_{n-1}} \leftarrow$ for ($i_{t_{n-1}}$=i_{t_n}; $i_{t_{n-1}} \leqslant i_{t_n}$+$T_{n_i}$-$T_{(n-1)_i}$; $i_{t_{n-1}}$+=$T_{(n-1)_i}$)
   ...
   $\mathcal{L}_{intertile_1} \leftarrow$ for (i_{t_1}=i_{t_2}; $i_{t_1} \leqslant i_{t_2}$+$T_{2_i}$-$T_{1_i}$; i_{t_1}+=T_{1_i})
   $\mathcal{L}_{intratile} \leftarrow$ for (i=i_{t_1}; $i \leqslant i_{t_1}$+T_{1_i}-st; i+=st)
   $\mathcal{L}'_{intratile} \leftarrow$ for (i=i_{t_n}; $i \leqslant i_{t_n}$+T_{n_i}-st; i+=st)
   where i, lb, ub, st are the iterator, the lower bound, the upper bound and the stride of loop $\mathcal{T}$ (respectively).
3. Return $\mathcal{L}_{intertile_n}$ for **GETINTER**.
   Return $(\mathcal{L}_{intertile_{n-1}}, \ldots, \mathcal{L}_{intertile_1}, \mathcal{L}_{intratile})$ for **GETINNERINTERINTRA**.
   Return $\mathcal{L}'_{intratile}$ for **GETINTRA**.

Figure 7: Generation of multi-level inter-tile loops and intra-tile loops

We use the algorithm depicted in Figure 7 to generate multi-level inter-tile loops and intra-tile loop for a given loop AST. Tiling a loop nest using n levels of tiling requires generation of n groups of inter-tile loops and one group of intra-tile loops. The group of inter-tile loops at the outermost level (level n) enumerate the origins of the largest tiles, and the successive groups of inter-tile loops (at levels $n-1$ to 1) traverse the origins of the smaller nested tiles. The innermost intra-tile loops visit all points inside each smallest tile. The multi-level loop tiling assumes that the outer-level tile sizes are a multiple of the inner-level tile sizes.

The core algorithm described in Figure 4 generates efficient parametric single-level tiled code by subdividing each statement polyhedron into partial and full tiles. Since the tiling loops that iterate over full tiles can be known at compile time, the main tiling algorithm can be extended to generate multi-level tiled code. From the implementation perspective, the multi-level tiling extension involves adding a new input parameter n to the recursive tiling procedure that describes the number of tiling levels, and modifying the procedure calls to GETINTER and GETINTRA to include the number of tiling levels. A modification on the terminating step (Step 1) of the core algorithm is also required. This step generates the intra-tile loops of the outer loops, and then uses the loops to enclose the partially transformed ASTs. We modify this step using the following condition. If the partially transformed ASTs contain no loops, then we currently are at the deepest nesting level, meaning that the enclosing tiling loops iterate over full tiles. Hence, instead of generating intra-tile loops, we use GETINNERINTERINTRA to generate $n-1$ sets of inter-tile loops starting from level $n-1$ to level one, and another set of intra-tile loops at the innermost level. On the other hand, if the outer tiling loops do not enumerate full tiles, we only generate intra-tile loops that correspond to the outermost-level inter-tile loops (using GETINTRA).

Figure 8 presents an example of generation of a multi-level tiled code. The generated two-level tiled code corresponds to the simple 2D perfect loop nest example shown previously in Figure 1(a). In

```
/* full tiles i */
for (it2=lbi; it2<=ubi-(T2i-sti); it2+=T2i) {
  /* compute lbv,ubv */
  // ... omitted ...
  if (lbv<=ubv) {
    /* prolog j */
    for (i=it2; i<=it2+(T2i-sti); i+=sti)
      for (j=lbj(i); j<=lbv-stj; j+=stj)
        S(i,j);
    /* full tiles j */
    for (jt2=lbv; jt2<=ubv-(T2j-stj); jt2+=T2j)
      for (it1=it2; it1<=it2+(T2i-T1i); it1+=T1i)
        for (jt1=jt2; jt1<=jt2+(T2j-T1j); jt1+=T1j)
          for (i=it1; i<=it1+(T1i-sti); i+=sti)
            for (j=jt1; j<=jt1+(T1j-stj); j+=stj)
              S(i,j);
    /* epilog j */
    for (i=it2; i<=it2+(T2i-sti); i+=sti)
      for (j=jt2; j<=ubj(i); j+=stj)
        S(i,j);
  } else
    /* untiled j */
    for (i=it2; i<=it2+(T2i-sti); i+=sti)
      for (j=lbj(i); j<=ubj(i); j+=stj)
        S(i,j);
}
/* epilog i */
// ... omitted ...
```

Figure 8: Parametric two-level tiled code for the perfectly nested loop from Figure 1(a)

the figure, we observe that the outermost inter-tile loops now use level-two loop iterators (i_{t_2} and j_{t_2}) and level-two tile size variables (T_{2_i} and T_{2_j}). The full-tile loop of j is tiled along both i and j dimensions for two levels of tiling; whereas the prolog, the epilog and the untiled loop of j are strip-mined along i dimension for only one level of tiling (i.e., there is no inter-tile loop i_{t_1}).

4.3.2 Optimizing Boundary Tiles

The multi-level tiling approach discussed so far does not optimize the partial/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. After loops that enumerate points inside boundary tiles are completely generated in Steps 1 and 4 of the core algorithm, we can further tile the boundary tiles by recursively calling the multi-level tiling procedure with a tile size that is used at the next lower level of tiling (i.e., $n-1$). In this way, all newly formed boundary tiles can be recursively tiled and refined into smaller full and partial tiles. To identify at compile time if the loop formed in the terminating step (Step 1) is a boundary-tile loop (not enumerating full tiles), the same checking condition used in the multi-level tiling extension is applied. Figure 9 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 non-tiled area, between using and not using boundary tile optimization.

4.3.3 Optimizing One-Time Loops

As mentioned earlier, loops running exactly once are explicitly inserted into the input ASTs for the tiled loop generation. Generating tiled code for loop structures that contain one-time loops using the tiling algorithm will of course generate correct results. However, we can substantially reduce the code generation time by always keeping one-time loops completely untiled. The true case of the if-condition used to check the presence of full tiles need not be evaluated and generated because the start and end of full tiles are always the same for one-time loops. Furthermore, the iterator variable of one-time loop is never actually used inside the scope of the loop body since its value has already been propagated properly by CLooG. This is exemplified in the pre-processed 1D Jacobi code shown previously in Figure 3(d). Consequently, later removal of one-time loops is always safe. As a post-processing after tiling, these dummy one-time loops get removed from the final tiled code to make it efficient.

5. EXPERIMENTAL EVALUATION

In this section, we discuss experiments carried out to assess the effectiveness of the developed tiling approach, which has been implemented in a code generation tool, PrimeTile. More details about PrimeTile can be found in [1, 18]. Comparisons are made with two state-of-the-art tiled-code generators, Pluto [30] and HiTLOG [19] (through one of the reviews for this paper, we became aware of another tiled-code generator [24] that has very recently been developed, but that system was unavailable to perform a comparison). PrimeTile is more general and powerful than both of 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 autotuning systems such as ATLAS; 2) HiTLOG can generate parametric tiled code only for perfectly nested affine loops, while PrimeTile generates parametrically tiled code for arbitrary imperfectly nested affine codes. The primary comparisons are with Pluto (version 0.4.1), since it can generate tiled code for imperfectly nested loops. For the special case of perfectly nested loops, PrimeTile was compared with both Pluto and HiTLOG.

Eight benchmarks were used, including linear algebra kernels and stencil computations, as listed in Table 1. Three of the benchmarks have perfectly nested loops, and the other five have imperfectly nested loops. As pointed out earlier, due to data dependences, skewing and other transformations may be needed to make rectangular tiling legal; the need for such skewing transformation is indicated in the fourth column. The fifth column displays the maximum loop-nest depth for 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 – by running Pluto without the `--tile` option, 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 transformation (such as skewing) needed for legality of rectangular tiling of the loops is performed. Intermediate CLooG files generated by Pluto were used as inputs for HiTLOG and PrimeTile, ensuring that all three systems performed tiling on an identically pre-processed version of the input code.

All experiments were run on a multicore Intel Xeon workstation with dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB), 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), 16 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64). We used version 4.2.4 of g++ (GCC) with -O3 optimization flag to compile the generated tiled codes.

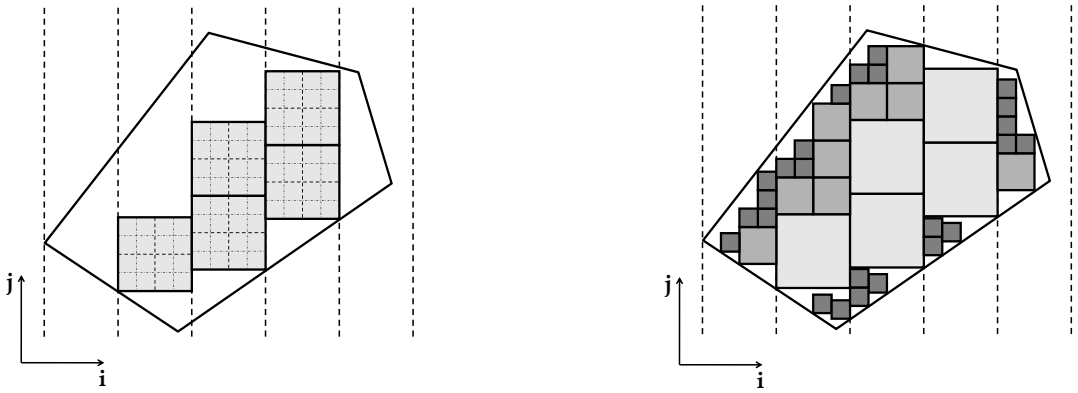


Figure 9: Iteration space tiled for three levels of tiling; w/o boundary tile optimization (left) and w/ boundary tile optimization (right)

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

	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): imperfectly nested loops

	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	0.32	0.09	0.06	0.85	0.33	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): perfectly nested loops

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 a tiled code in which boundary tiles are also fully recursively tiled (using smaller tile sizes). The other version (labeled "PrimeTile(n)") represents a tiled code in which boundary tiles are not tiled at all. The time taken for code generation for the five imperfectly nested benchmarks and the three perfectly nested benchmarks are given in Table 2 and Table 3, respectively. With PrimeTile and Pluto, 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 performance tradeoff 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, versions can be generated that vary in the level of tiling performed for the boundary tiles, in between the extremes corresponding to PrimeTile(n) and PrimeTile(f).

The Pluto system only performs multi-level tiling for up to two levels. In our experiments, we used 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 of Ancourt and Irigoin [4]) and duplicated scattering functions for the tile space. We manually analyzed the CLooG input file generated by Pluto for each benchmark to create a script to parse the CLooG input file and modify the extracted domains and scattering functions with additional tiling dimensions using the same method that Pluto uses.

	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)

Pluto fails to generate tiled code for more than four levels of tiling (because of overflow errors in calls to a polyhedral library within CLooG). PrimeTile successfully generates tiled code for any number of levels of tiling (we tried up to 8 levels). Except for the case of single-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.

5.2 Performance of Generated Tiled Code

In this section, we assess the efficiency of the tiled code generated by PrimeTile. We generated tiled code using one level of tiling and two levels of tiling for various tile sizes. For one-level tiling, the considered tile sizes are 2^n for n ranging from 1 to 10. For two-level tiling, the outer tile sizes (T_2) are 128, 256, 512, 768, and 1024, and the inner tile sizes (T_1) range from 2 to $T_2/2$. We used square tiles just for ease of experimentation. We generated tiled code using the three tiled code generators and measured the execution time for all combinations of tile sizes, to select the best-performing tile sizes. Table 4 lists the best execution times of the tiled codes generated by the three systems.

We further optimized the best-performing tiled code to assess the benefits of separating full tiles. For tiled code generated by PrimeTile and HiTLOG, we performed an additional 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 tried 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 the tiled codes enhanced using unrolling and scalar replacement for PrimeTile and HiTLOG (rows corresponding to tiling methods with suffix "(regtile)"). Pluto provides a loop unroll-jam facility that is controlled using the `--ufactor=<factor>` optimization flag. We used unroll factors ranging from 1 to 10 to generate unroll-jammed code from Pluto. However, Pluto does not apply scalar replacement optimization. Hence, for a fair comparison with Pluto, we also present the performance results of register tiled code without scalar replacement for PrimeTile and HiTLOG (rows corresponding to tiling methods with suffix "(unroll)"). Additional details, such as the best tile sizes and best unroll factors corresponding to the optimized cases reported in Table 4, are available in another report [18].

For imperfectly nested loops, the performance of the tiled code generated by Pluto is comparable to that generated by PrimeTile. For perfectly nested loops, the performance results demonstrate that the parametric multi-level tiled codes generated by PrimeTile and HiTLOG consistently outperform the fixed multi-level tiled code from Pluto. The results also indicate that the execution times of the tiled codes generated by PrimeTile and HiTLOG are very comparable.

Due to the loop unrolling and scalar replacement optimizations, the register tiled code performs significantly better than the code that is tiled only for different levels of caches. The best unroll-jammed code from Pluto consistently performs worse compared to the unrolled code from PrimeTile or HiTLOG. The unrolled and register tiled versions from PrimeTile and HiTLOG exhibit comparable performance.

6. 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 multi-level tiling approaches were restricted to perfectly nested loops, while previous solutions to tiling for imperfect loop nests only handled fixed tile sizes. This paper describes an effective approach to parametric multi-level tiling of imperfectly nested affine loops. The key idea behind the algorithm is the use of a bounds-based approach to multi-statement tile separation by analysis of the AST generated by Quilleré et al.'s polyhedra scanning algorithm, in conjunction with schedules that satisfy a generalized tiling condition for multi-statement domains. Separation of partial tiles from full tiles is also achieved, thereby enabling optimizations such as register tiling. The effectiveness of the tiling approach has been demonstrated through experimental evaluation using a number of computational benchmarks.

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

Acknowledgments

We thank Uday Bondhugula for many productive discussions and his invaluable help with Pluto, and Bhargavi Rajaraman for help on the experimental evaluation. We thank Lakshminarayanan Renganarayanan for valuable feedback that helped improve the presentation of the paper. This work was supported in part by the U.S. National Science Foundation through awards 0403342, 0508245, 0509442, 0509467, 0541409, 0811457 and 0811781, and by a grant from the State of Ohio Development Fund.

7. 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. Efficient code generation for automatic parallelization and optimization. In *ISPDC*, page 23, 2003.

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

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *PLDI'08*, 2008.

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

[10] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proc. Supercomputing '92*, pages 114–124, 1992.

[11] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO'05*, 2005.

[12] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. Technical Report 08-897, USC Computer Science Technical Report, June 2008.

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

[14] S. Coleman and K. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *PLDI'95*, pages 279–290, 1995.

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

[16] 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.

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

[18] A. Hartono, M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Primetile: A parametric multi-level tiler for imperfect loop nests. Technical Report OSU-CISRC-2/09-TR04, The Ohio State University, Feb. 2009.

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

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

[21] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

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

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

[24] D. Kim and S. Rajopadhye. Parameterized tiling for imperfectly nested loops. Technical Report CS-09-101, Colorado State University, Department of Computer Science, February 2009.

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

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

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

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

[29] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *PPoPP'01*, 2001.

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

[31] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *CACM*, 8:102–114, Aug. 1992.

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

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

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

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

[36] G. Rivera and C. Tseng. Locality optimizations for multi-level caches. In *Supercomputing '99*, page 2, 1999.

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

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

[39] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. Scalable autotuning framework for compiler optimization. In *IPDPS '09*, May 2009.

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

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

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

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

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

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