Parametric Tiling of Affine Loop Nests

Sanket Tavarageri,[1] Albert Hartono,[1,2] Muthu Baskaran,[1,2] Louis-Noël Pouchet,[1]
J. Ramanujam[3] and P. Sadayappan[1]

[1] The Ohio State University
[2] Reservoir Labs, Inc. (work done while at The Ohio State University)
[3] Louisiana State University

Abstract. Tiling, a key transformation for optimizing programs, has been widely studied in the literature. Parameterized tiled code is important for auto-tuning systems since they often execute a large number of runs with dynamically varied tile sizes. In this paper, we present a comparative study of three recently developed approaches to parametric tiling of imperfectly nested loops.

1 Introduction

The ubiquity of multicore processors has brought parallel computing squarely into the mainstream. Unlike the past, when the development of parallel programs was primarily a task undertaken by a small cadre of expert programmers, it is now essential to develop efficient parallel implementations of a large number of sequential codes. Current trends in micro architecture are increasingly towards larger number of processing elements on a single chip. The difficulty of programming these architectures to effectively tap the potential of multiple on-chip processing units is a significant challenge. With the increasing number of cores in multicore processors, the aggregate bandwidth between memory and cache is often a critical bottleneck that limits the effective exploitation of parallelism.

Tiling is a key transformation in optimizing for parallelism and data locality. Tiling for locality involves grouping points in an iteration space into smaller blocks (tiles) allowing reuse in multiple directions when the block fits in a faster level of the memory hierarchy (registers, L1, or L2 cache). Tiling for coarse-grained parallelism partitions the iteration space into tiles that may be executed concurrently on different processors with a reduced frequency and volume of inter-processor communication: a tile is atomically executed on a processor with communication required only before and after execution. Tiling has received a lot of attention in the compiler community [16, 29, 25, 5, 34, 15, 1, 21], but until recently there was no robust algorithm and software implementation that had been demonstrated to be effective on a number of benchmarks. The first effective approach for tiling of imperfectly nested loops was developed in the Pluto polyhedral transformation framework [4, 23]. However, Pluto can only generate tiled code where the tile sizes are fixed at compile-time. Since the performance of tiled code can vary greatly with the choice of tile sizes, it is highly desirable to specify the tile sizes as runtime parameters in the code. The generation of tiled code where tile sizes of loops are runtime parameters is called *parametric tiling;* such an approach would enable empirical search for tile sizes in auto-tuning systems.

Automatic tuning approaches perform empirical parameter searches on the target platform. For example, ATLAS [33] 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 performance results of ATLAS are comparable to those of vendor-provided BLAS libraries. But the ATLAS system can only tune BLAS kernels and 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 [7, 31, 2]. An efficient parametric tiling tool is extremely valuable for generating input tiled codes for such empirical tuning systems.

In this paper, we present a comparative study of three recently developed approaches for parametric tiling of affine loop nests:

- **PrimeTile**: This was the first system to generate parametrically tiled code for affine imperfectly nested loops. As explained in the next section, it uses a level by level approach to generate tiled code, with a prolog, epilog, and a full-tiles loop nest corresponding to each nesting level of the original code. The approach was limited to sequential output tiled code.
- **DynTile**: This was the first implementation of an approach for parallel execution of parametrically tiled affine code. It utilizes wavefront parallelism in the tiled iteration space corresponding to the convex hull of all the statement domains of the input untiled code. Wavefront parallelism in the tiled iteration space is exploited through use of an inspector/executor approach. A statically generated inspector code scans the tiles and places them in bins corresponding to the different wavefronts. Dynamic scheduling of the tiles is then performed in order of increasing wavefronts.
- **PTile**: This was the first approach to compile-time generation of code for wavefront-parallel tiled execution. Instead of the dynamic runtime scanning and binning of tiles as done by the DynTile approaches, transformed code for wavefront parallel tiled execution is automatically generated.

The rest of the paper is organized as follows. Section 2 presents an overview of the three approaches that are compared. Experimental results on a number of benchmarks are presented in Section 3. Related work is discussed in Section 4. We conclude the paper with a discussion in Section 5.

2 Parametric Tiling

In this section, we provide a brief overview of the three approaches to parametric tiling that we experimentally evaluate in this paper.

2.1 PrimeTile

We provide a short overview of PrimeTile by discussing the approach to generation of parametric full tiles in the context of perfectly nested loops. Details on how imperfectly nested loops are handled may be found in [12].

Consider the 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

```
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);
```

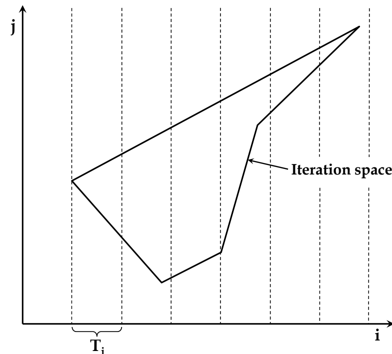
(b) After tiling loop *i*

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

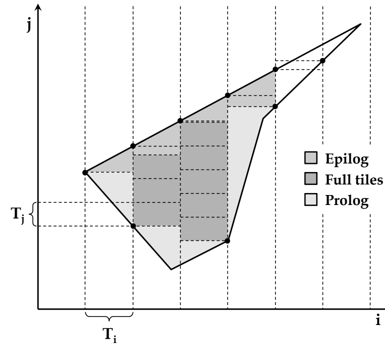
(c) After tiling loops *i* and *j*

```
/* 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*)

Fig. 1. Parametric tiling of a perfectly nested loop in PrimeTile

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 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 in [12], 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 pseudo-code 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.

2.2 DynTile

When the tile sizes are parametric, it is problematic to generate parallel code using the polyhedral framework since nonlinear expressions arise in the specification of constraints and objective functions. Hence for an arbitrary parametric tiled code, it is non-trivial to extract parallelism. In [13], we developed an approach that circumvents the problems by employing a dynamic scheduling approach to schedule tiles for parallel execution.

In the case of a program with single statement, the loop structure is a perfect loop nest. Generating aligned tiled code involves syntactic processing of the loop bounds in addition to generating the tile loops. The tile loops are generated as perfectly nested loops

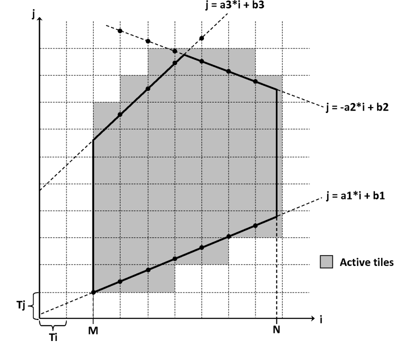
that enumerate the tiles as tile numbers in the tile space. Fig. 2 illustrates an example for generating aligned tiled code for a single statement program.

```
for (i=M;i<=N;i++)
  for (j=b1+a1*i;j<=min(b2-a2*i,b3+a3*i);j++)
    S(i,j);
```

(a) Original loop structure

```
/* Intertile loops it, jt */
for (it=floor(M/Ti); it<=floor(N/Ti); it++)
  for (jt=floor((a1*(it*Ti)+b1)/Tj);
       jt<=floor((min(b2-a2*(it*Ti),
                      a3*(it*Ti+Ti-1)+b3))/Tj);
       jt++)
   /* Intratile loops i, j */
    for (i=max(M,it*Ti);i<=min(N,it*Ti+Ti-1);i++)
      for (j=max(a1*i+b1,jt*Tj);
           j<=min(min(b2-a2*i,b3+a3*i),jt*Tj+Tj-1);
           j++)
        S(i,j);
```

(b) Tiled loop structure



(c) Tiled iteration space (shown with region of active tiles)

Fig. 2. Parametric tiling of a single statement domain in DynTile

In the case of a program with multiple statements, the loop structure is an imperfectly nested loop. Generating aligned tiled code in this case involves additional processing to generate perfectly nested tile loops. The convex hull of the union of the domains of all statements is found and used to generate the loop structure of the tile loops.

With DynTile, a runtime scheduling approach is used to schedule tiles in a wavefront for parallel execution. The approach involves generating an *inspector code* at compile-time that at runtime creates *bins* of tiles where each bin represents a wavefront. The tiles in a bin, henceforth, are scheduled for parallel execution.

2.3 PTile

In contrast to the DynTile approach described previously, the PTile approach uses a fully polyhedral technique for compile-time generation of parallel parametric tiled code.

Let $v_1, v_2, \ldots, v_n$ represent the loop variables a loop nest of depth n (v_1 representing the outermost loop and v_n representing the innermost loop). Let $p_1, p_2, \ldots, p_k$ represent symbolic parameters (such as problem sizes). The system S (of m inequalities) representing the domain of the program is given by

$$S : \sum_{j=1}^{n} B_{ij}.v_j + \sum_{j=1}^{k} P_{ij}.p_j + c_i \geq 0, \qquad i \in [1..m]$$

where each B_{ij} and P_{ij} represent the coefficients of the corresponding loop variable and parameter, respectively, and c_i represents a constant in an inequality.

The m inequalities represent the lower and upper bounds of all loop variables. Hence the system S is in a *row echelon* form where the inequalities expressing the loop bounds of a variable v_i have coefficient 0 for all variables $v_j : i < j \leq n$. In other words, the bounds of a loop variable v_i are expressed as a function of its outer loop variables (v_j : $1 \leq j < i$), parameters and constants. The loop bounds would look like:

$$max(f_{11}(p,c),\dots,f_{1k}(p,c)) \leq v_1 \leq$$
$$min(g_{11}(p,c),\dots g_{1l}(p,c))$$
$$max(f_{21}(v_1,p,c),\dots,f_{2q}(v_1,p,c)) \leq v_2$$
$$\leq min(g_{21}(v_1,p,c),\dots g_{2r}(v_1,p,c))$$
$$\ddots$$
$$max(f_{n1}(v_1,\dots,v_{n-1},p,c),\dots,f_{ny}(v_1,\dots,v_{n-1},p,c)) \leq$$
$$v_n \leq min(g_{n1}(v_1,\dots,v_{n-1},p,c),\dots g_{nz}(v_1,\dots,v_{n-1},p,c))$$

Conversely, given a system of inequalities in row echelon form, loops generated with bounds for each loop variable derived directly from the system (in row echelon form) scan all valid integer points represented by the system (details may be found in [3]).

The PTile approach to parameterized tiling relies on the above property for generating code from a system of inequalities in row echelon form and the fact that a system with tiling transformation (equivalent to the original system) can be derived. Each variable v_j in the domain (which in turn represents each dimension in the domain) can be expressed in terms of tile coordinates t_j, tile sizes s_j, and intra-tile coordinates u_j as: $v_j = s_j.t_j + u_j \wedge 0 \leq u_j \leq s_j - 1$. The system S can now be (equivalently) represented as:

$$S' : \sum_{j=1}^{n} B_{ij}.s_j.t_j + \sum_{j=1}^{n} B_{ij}.u_j + \sum_{j=1}^{k} P_{ij}.p_j + c_i \geq 0,$$
$$i \in [1..m] \quad \wedge \quad 0 \leq u_j \leq s_j - 1, \;\; j \in [1..n]$$

A new system S_T is derived from S' such that the solutions to S' satisfy S_T. In the new system S_T, the intra-tile coordinates are eliminated through a relaxed projection. S_T is as follows:

$$S_T : \sum_{j=1}^{n} B_{ij}.s_j.t_j + \sum_{j=1}^{n} B^{+}_{ij}.(s_j - 1) + \sum_{j=1}^{k} P_{ij}.p_j + c_i \geq 0, \quad i \in [1..m]$$

Two important properties of S_T (details may be found in [3]) are: (1) the solutions to S' also satisfy S_T and (2) S_T is in row echelon form. Hence scanning S_T will generate the tile loops (loops with tile coordinates).

The constraints expressed by S_T together with that expressed by S' represent the complete set of inequalities characterizing the loop structure of sequential tiled code. Scanning S_T generates the tile loops as discussed above. Scanning S' generates the intra-tile loops in terms of tile coordinates, tile sizes, and intra-tile coordinates.

Fig. 3(a) shows an example of a nested loop, whose statement domain can be expressed as a set of inequalities shown in Fig. 3(b). After the original loop iterators are

```
for  (v1=0; v1<=T−1;v1++)
 for  (v2=2∗v1+2;v2<=2∗v1+N−1;v2++)
  for  (v3=max(2∗v1+2,v2−N+4);
   v3<=min(2∗v1+N−1,v2+N−4);v3++)
     S1(v1,v2,v3);
```

(a) Loop nest

$$\begin{aligned} v_1 &\geq 0 \\ -v_1+T-1 &\geq 0 \\ -2v_1+v_2-2 &\geq 0 \\ 2v_1-v_2+N-1 &\geq 0 \\ -2v_1+v_3-2 &\geq 0 \\ 2v_1-v_3+N-1 &\geq 0 \\ -v_2+v_3+N-4 &\geq 0 \\ v_2-v_3+N-4 &\geq 0 \end{aligned}$$

(b) Inequalities for statement domain

$$\begin{aligned} s_1.t_1+u_1 &\geq 0 \\ -s_1.t_1-u_1+T-1 &\geq 0 \\ -2s_1.t_1-2u_1+s_2.t_2+u_2-2 &\geq 0 \\ 2s_1.t_1+2u_1-s_2.t_2-u_2+N-1 &\geq 0 \\ -2s_1.t_1-2u_1+s_3.t_3+u_3-2 &\geq 0 \\ 2s_1.t_1+2u_1-s_3.t_3-u_3+N-1 &\geq 0 \\ -s_2.t_2-u_2+s_3.t_3+u_3+N-4 &\geq 0 \\ s_2.t_2+u_2-s_3.t_3-u_3+N-4 &\geq 0 \end{aligned}$$

(c) Inequalities for tiling iterators

$$\begin{aligned} s_1.t_1+s_1-1 &\geq 0 \\ -s_1.t_1+T-1 &\geq 0 \\ -2s_1.t_1+s_2.t_2+s_2-1-2 &\geq 0 \\ 2s_1.t_1-s_2.t_2+2s_1-2+N-1 &\geq 0 \\ -2s_1.t_1+s_3.t_3+s_3-1-2 &\geq 0 \\ 2s_1.t_1-s_3.t_3+2s_1-2+N-1 &\geq 0 \\ -s_2.t_2+s_3.t_3+s_3-1+N-4 &\geq 0 \\ s_2.t_2-s_3.t_3+s_2-1+N-4 &\geq 0 \end{aligned}$$

(d) Row-echelon tiling iterators

$$\begin{aligned} &\texttt{for } (t_1=\lceil\tfrac{-s_1+1}{s1}\rceil; t_1 <= \lfloor\tfrac{T-1}{s_1}\rfloor; t_1\texttt{++}) \\ &\quad\texttt{for } (t_2=\lceil\tfrac{2*s_1*t_1-s_2+3}{s_2}\rceil; t_2 <= \lfloor\tfrac{2*s_1*t_1+2*s_1+N-3}{s_2}\rfloor; t_2\texttt{++}) \\ &\qquad\texttt{for } (t_3=\texttt{max}(\lceil\tfrac{2*s_1*t_1-s_3+3}{s_3}\rceil, \lceil\tfrac{s_2*t_2-s_3-N+5}{s_3}\rceil); \\ &\qquad\quad t_3 <= \texttt{min}(\lfloor\tfrac{2*s_1*t_1+2*s_1+N-3}{s_3}\rfloor, \lfloor\tfrac{s_2*t_2+s_2+N-5}{s_3}\rfloor); t_3\texttt{++}) \end{aligned}$$

(e) Parametrically tiled loop nest

Fig. 3. Illustration of sequential parametric tiling in PTile

rewritten in terms of tiling loops and intra-tile loops, the system of inequalities can be rewritten as shown in Fig. 3(c). The intra-tile variables in the system can be eliminated by using upper/lower bounds to generate inequalities for the tile iterators in row-echelon form, as shown in Fig. 3(d). The parametrically tiled code is shown in Fig. 3(e)

After tiling as described above, if any of the tiling loops is parallel (i.e. has no loop carried dependences), coarse-grained parallel tiled execution is directly possible. However, even if none of the tiling loops is parallel, wavefront parallelism is always feasible among the tiles. But instead of viewing wavefront-parallel tile execution as involving a unimodular transformation from one n-dimensional space (nesting order $t_1, t_2, \ldots, t_n$ of sequential tiled execution) to another n-dimensional space (nesting order

$w, t_1, t_2, \ldots, t_{n-1}$), it is viewed in terms of a sparse $n+1$ dimensional space with nesting order $w, t_1, t_2, \ldots, t_n$. While this might seem very wasteful, by optimizing the scanning of this higher dimensional space, parameterized parallel tiled execution is achieved with negligible overhead of scanning empty tiles. The primary problem of generating loop bounds for the outermost w loop via Symbolic Fourier Motzkin elimination is eliminated by generating the lowest and highest numbered wavefronts in the untiled form of the loops (which can be generated as a parametric expression in the problem parameters by use of an integer linear programming solver such as PIP [10, 22]) and then generating bounds for the lowest and highest numbered tiled wavefront loop. No explicit "skewing" of the tile space is done; the $t_1, t_2, \ldots, t_n$ loops are executed in original lexicographic order but constrained to include only those tiles that actually belong in the current tile wavefront w. The $n+1$ dimensional loop nest $w, t_1, t_2, \ldots, t_n$ is optimized by addition of constraints derived from the wavefront inequalities.

Starting with the system S_T that would generate the tile loops for sequential tiled execution, the equality $w = \sum_{j=1}^{n} t_j$ is added, representing the relation between the tile loop variables and the wavefront number. This equality is introduced using the two inequalities:

$$w - \sum_{j=1}^{n} t_j \geq 0 \qquad \text{and} \qquad -w + \sum_{j=1}^{n} t_j \geq 0.$$

These new inequalities are combined with the existing inequalities representing the loop bounds for sequential tiled execution through a symbolic Fourier Motzkin procedure. The main challenge is that with symbolic parameters, certain terms may have ambiguous sign. If such ambiguity occurs in the symbolic FM process, a relaxation of the inequality is performed to generate a weaker inequality, i.e., one that is guaranteed to be true for any values that satisfy the original inequality. The relaxation is performed in a manner that removes any critical ambiguity in the sign of variable coefficients in the inequality. The approach is illustrated using an example below.

Consider the system S_T for generating sequential tile loops (as discussed earlier) and the wavefront inequalities representing $w = \Sigma_{j=1}^{n} t_j$. The RSFME procedure starts by combining the lower bound inequalities of t_n from S_T with the wavefront inequality $w - \Sigma_{j=1}^{n-1} t_j - t_n \geq 0$ and upper bound inequalities of t_n from S_T with the wavefront inequality $-w + \Sigma_{j=1}^{n-1} t_j + t_n \geq 0$, to eliminate t_n and hence derive new lower and upper bound inequalities for t_{n-1}. However while combining the bounds, there is a possibility that we might need to add symbolic coefficients of two terms with opposite signs and the resulting sign may be indeterminate at compile-time. At this point, we replace the tile loop variables with their parametric bounded values (t_j^{min} or t_j^{max}) and use relaxed bound inequalities. At any level k of the RSFME procedure, we combine new wavefront inequalities added at level k with the existing lower bound and upper bound inequalities at level k to eliminate the tile loop variable at level k and derive new lower and upper bound inequalities for level $k-1$.

3 Qualitative Analysis of Parametric Tiling

We now present a comparative evaluation of the three parametric tiling techniques discussed: PrimeTile, DynTile and PTile. Our objective is to measure the impact on perfor-

	Lower-bound constraint	Upper-bound constraint
Outermost wavefront loop:	(1a): wmin ≤ w	(1b): w ≤ wmax
Loop it:	(2a): (1-Ti+1)/Ti ≤ it	(2b): it ≤ N/Ti
Loop jt:	(3a): (1-Tj+1)/Tj ≤ jt	(3b): jt ≤ (N-it.Ti)/Tj
Loop kt:	(4a): (it.Ti-Tk+1)/Tk ≤ kt	(4b): kt ≤ N/Tk
Wavefront constraints:	(5a): w-it-jt ≤ kt	(5b): kt ≤ w-it-jt
Eliminate variable kt to derive new wavefront constraints for loop jt:	(6a): Combine (5a) and (4b) w-it-jt ≤ N/Tk w-it-N/Tk ≤ jt (w.Tk-it.Tk-N)/Tk ≤ jt	(6b): Combine (5b) and (4a) (it.Ti-Tk+1)/Tk ≤ w-it-jt jt ≤ w-it-it.Ti/Tk+1-1/Tk jt ≤ (w.Tk-it.Tk-it.Ti+Tk-1)/Tk
Eliminate variable jt to derive new wavefront constraints for loop it:	(7a): Combine (6a) and (3b) w-it-N/Tk ≤ N/Tj-it.Ti/Tj w-N/Tj-N/Tk ≤ it-it.Ti/Tj *(ambiguous sign encountered)* → (i) Relaxation to obtain a new lower-bound constraint: w-N/Tj-N/Tk+itmin.Ti/Tj ≤ it $\frac{(w.Tj.Tk-N.Tk-N.Tj+itmin.Ti.Tk)}{(Tj.Tk)} \leq it$ → (ii) Relaxation to obtain a new upper-bound constraint: it.Ti/Tj ≤ itmax-w+N/Tj+N/Tk $it \leq \frac{(itmax.Tj.Tk-w.Tj.Tk+N.Tj+N.Tk)}{(Ti.Tk)}$	(7b): Combine (6b) and (3a) 2/Tj-1 ≤ w-it-it.Ti/Tk+1-1/Tk it+it.Ti/Tk ≤ w+2-2/Tj-1/Tk $it \leq \frac{(w.Tj.Tk^2+2.Tj.Tk^2-Tj.Tk-2.Tk^2)}{(Ti.Tj.Tk+Tj.Tk^2)}$

Fig. 4. Example of application of Relaxed Symbolic Fourier Motzkin Elimination (RSFME) on a non-rectangular tiled loop nest

mance of the different alternatives to generate tiled code. We first study the impact of the relaxation step in the RSFME algorithm, then measure the loop control overhead for the three techniques, and finally compare the performance of the parametrically tiled code when run sequentially and run using coarse-grain parallelism.

3.1 Experimental Setup

Target machine The experiments were run on a dual-socket quad-Core AMD Opteron Shangai 8218 processors running at 2.5GHz, with 64+64kB L1 cache/core, 512kB L2 cache/core and 6MB L3 cache. This processor is a node of the Ohio Supercomputer Center clusters. We used two vendor compilers: GNU GCC 4.4.0 and Intel ICC 11.0. For each we experimented with and without vectorization support: gcc-novec uses option `-O3 -fno-tree-vectorize`, gcc-vec uses option `-O3`, icc-novec uses option `-fast`

-no-vec and icc-vec uses option -fast. Finally, for parallel execution experiments, we marked the outer-most parallel loop with an OpenMP pragma and turned on the -fopenmp flag for GCC, and -openmp flag for ICC.

Benchmarks We focus our study on four numerical kernels. They are listed in Table 1. All these benchmarks use double-precision floating point arithmetic. Note that although we do use a specific problem size for the experiments, the parametric tiling software is presented with an arbitrarily-nested input code with parametric loop bounds, which forms the most general setting of affine-control programs. Hence the code does not need to be generated again for a different problem size.

Table 1. Benchmarks used in the experiments

Name	Description	Max loop depth	Problem size
2d-fdtd	2D Finite Difference Time Domain method	3	T=1500, N=1500
cholesky	Cholesky factorization	3	N=2500
dtrmm	Triangular matrix multiplication	3	N=2000
lu	LU factorization	3	N=2000

For each benchmark, a sequence of affine loop transformations was applied to make the loop nest(s) fully permutable, and hence tilable [4, 23]. Note that for stencils, such as 2d-fdtd for instance, it is necessary to skew the original iteration domains to enable tiling for all loops; this skewed and tiled version is generated seamlessly using the affine framework. For all experiments, the tile size was set to 16 for all tiled dimensions.

We experimented with three approaches to parametric tiling code generation, namely PrimeTile [12, 24], Dyntile [13], and PTile [3].

3.2 Tightness of Relaxed Symboliv Fourier-Motzkin Elimination

The extension of the Fourier-Motzkin elimination algorithm to symbolic affine constraints raises the problem of possibly combining two inequalities containing parametric constants of opposite signs. For instance, one may need to combine the following inequalities

$$T_i/T_j * it - jt \geq 0$$
$$-T_i * it + T_k * jt \geq 0$$

in order to eliminate the jt variable. As there is no assumption made at compile-time on the actual value of the parameters T_i, T_j and T_k, combining the above constraints results in:

$$(T_i/T_j - T_i/T_k) * it \geq 0$$

where the sign of $(T_i/T_j - T_i/T_k)$ could be positive or negative, and thus represent either a lower bound or an upper bound for it. Because the input code has affine bounds, the only

source of such ambiguity cases comes from the relative values of tile size parameters [3]. The tile iterators (e.g., it, jt) are bounded by parametric constants since we are tiling a bounded iteration domain, so that we have: $it^{min} \leq it \leq it^{min}$ and $jt^{min} \leq jt \leq jt^{min}$. Therefore, a sufficient approximation for correctness is to relax the constraint by substituting jt with jt^{min} and jt^{max}, thus leading to the constraints:

$$T_i/T_j * it - jt^{min} \geq 0$$
$$-T_i * it + T_k * jt^{max} \geq 0,$$

which define a bounding box around the it variable. The drawback is the scanning of empty points, as the relaxation introduces over-approximation of the bounds for it. This may lead to control overhead and in the worst case to scanning empty tiles.

To determine the potential penalty introduced by such a relaxation, we analyzed the execution of the RSFME algorithm on our benchmark suite. Interestingly, in our experiments we found that the relaxation step of RSFME was never needed by the algorithm: no parametric expressions of opposite signs were combined during the resolution. We extended this analysis to five additional benchmarks: 2d-jacobi, adi, gemver, trisolve and 3d-stencil, and also observed that *the relaxation step was never required to successfully compute the tile loop bounds*. Hence, it is expected that for most cases the bounds for the tile loops as generated by PTile will match the convex hull of the iteration domain to be tiled.

3.3 Control Overhead Comparison

There is a trade-off between the code size and the control complexity between the code generated by the three considered techniques for parametric tiling. PrimeTile generates simple loop bounds at the expense of increasing the code size by explicitly expanding the various possible cases of tiles into distinct code blocks. On the other hand, DynTile and PTile produce significantly smaller sized codes, because a single loop nest has its bounds parameterized to deal with all tile cases. The drawback is much more complex loop bounds for DynTile and PTile (presence of multiplications, divisions, min and max expressions) that may be detrimental to performance.

The purpose of this experiment is to measure the control overhead for the three techniques. For this experiment, we substituted all statements in the program with a dummy scalar increment (which is executed about 3×10^9 times for all benchmarks). So any effect from the floating point operations or memory accesses are removed and the only remaining timed operations are either control-related, or the dummy scalar increment. We report in Table 2 the execution time, in seconds, for each benchmark using dummy statements. For all experiments, the tiled code implements separation of partial and full tiles.

We observe that for cholesky, dtrmm and lu benchmarks, the execution time difference between the three techniques is small enough to be attributable to experimental noise. For these benchmarks, there is no penalty in using more complex controls such as in PTile or DynTile to generate the loop bounds. For 2d-fdtd, we observe a control overhead difference of 20–40% between the three techniques. After manual inspection of the generated codes, we have been unable to identify a clear source for the perfor-

Table 2. Control Overhead Comparison for PrimeTile, DynTile and PTile

Benchmark	Compiler	PrimeTile	DynTile	PTile
2d-fdtd	gcc-novec	4.16s	4.23s	5.91s
2d-fdtd	gcc-vec	4.15s	4.23s	5.89s
2d-fdtd	icc-novec	4.04s	4.12s	4.63s
2d-fdtd	icc-vec	2.97s	3.25s	3.72s
cholesky	gcc-novec	3.11s	3.05s	3.24s
cholesky	gcc-vec	3.19s	3.05s	5.07s
cholesky	icc-novec	4.06s	3.04s	3.23s
cholesky	icc-vec	2.26s	2.30s	2.46s
dtrmm	gcc-novec	4.97s	4.96s	5.07s
dtrmm	gcc-vec	4.97s	4.96s	6.51s
dtrmm	icc-novec	6.44s	6.44s	4.94s
dtrmm	icc-vec	3.50s	3.75s	3.64s
lu	gcc-novec	3.30s	3.23s	3.38s
lu	gcc-vec	4.20s	3.23s	4.35s
lu	icc-novec	3.20s	3.23s	4.32s
lu	icc-vec	2.28s	2.43s	2.60s

mance drop, and one can only suspect that the control structure became too complex for the compiler to efficiently optimize.

3.4 Performance of Tiled Code

Next, we present the actual execution times of the program generated with the three techniques. For a fair comparison, all versions implemented separation of full tiles from partial tiles, but no register tiling was done. The execution times are shown in Table 3.

We observe that for sequential execution, PrimeTile performs best in our experiments, with a performance often comparable with DynTile, in particular when considering ICC. We also observe that GCC has more trouble in optimizing the code generated by DynTile than the one generated by PrimeTile. The slowdown of PTile is explained by the order in which tiles are executed: PTile generates a wavefront outer-most tile loop, that is needed to ensure correctness of parallel tile execution. However, for sequential execution, this wavefront order is detrimental to locality, and thus to performance.

We also report the execution time of the programs generated with DynTile and PTile for parallel tiled execution. PrimeTile is not used for this comparison as it cannot natively generate parallel parametric tiled code. For the experiments shown in Table 3, tiles were executed in pipeline-parallel fashion using a sequential outer-loop to capture wavefronts. The next outer-most loop was marked with `#pragma omp for` and the program run using 8 H/W threads, from within a single OSC cluster node. We observe that DynTile slightly outperforms PTile for parallel execution. The explanation lies in the amount of parallelism that is exploited by the two approaches. For DynTile, all tiles that can be executed in parallel for a given wavefront are indeed marked as parallel, with a `#pragma`

Table 3. Execution Time with PrimeTile, DynTile and PTile Generated Code

		Sequential			Parallel	
Benchmark	Compiler	PrimeTile	DynTile	PTile	DynTile	PTile
2d-fdtd	gcc-novec	43.84s	49.19s	56.78s	9.32s	10.98s
2d-fdtd	gcc-vec	43.82s	49.22s	56.85s	9.37s	10.98s
2d-fdtd	icc-novec	40.27s	48.12s	54.29s	13.30s	12.96s
2d-fdtd	icc-vec	40.52s	49.61s	54.63s	13.03s	13.18s
cholesky	gcc-novec	6.13s	10.50s	13.43s	1.91s	2.81s
cholesky	gcc-vec	6.08s	10.46s	13.45s	1.89s	2.82s
cholesky	icc-novec	5.63s	5.86s	8.19s	1.21s	2.40s
cholesky	icc-vec	5.36s	5.74s	8.22s	1.27s	2.61s
dtrmm	gcc-novec	9.29s	14.34s	18.99s	2.55s	4.50s
dtrmm	gcc-vec	9.25s	14.57s	18.99s	2.54s	3.69s
dtrmm	icc-novec	9.84s	9.19s	13.27s	2.17s	3.22s
dtrmm	icc-vec	9.91s	9.12s	13.44s	2.33s	3.27s
lu	gcc-novec	8.30s	9.15s	10.98s	2.56s	2.94s
lu	gcc-vec	8.29s	9.15s	10.98s	2.98s	2.43s
lu	icc-novec	6.30s	5.63s	7.49s	6.18s	1.60s
lu	icc-vec	6.36s	5.58s	6.52s	6.36s	1.62s

`omp for` around the loop which iterates over all tiles belonging to the wavefront. However, for PTile, only the next outer-most loop is marked as parallel, the outer-most loop being the wavefront loop. So even though the remaining tile loops are also parallel, this parallelism is not exploited. It is expected that coalescing the parallel tile loops into a single loop would improve load balancing, and exploit all the available parallelism. For lu with ICC, we remark that the execution time is excessively high for DynTile. Although there is no clear explanation after scrutiny of the generated code, we suspect that effective ICC optimization is somehow being inhibited by this specific input code.

We also observed that for all considered benchmarks, neither ICC nor GCC was able to vectorize loops in the computational kernel. For most cases, the compiler was unable to analyze the loops because of complex controls and/or access patterns. The increased complexity of controls generated by polyhedral parametric tiling challenges the compiler's loop analyzer, and it has been shown that for such cases it is beneficial to expose and explicitly mark vectorizable loops from within the polyhedral transformation framework [4].

4 Related Work

Much work exists in the area of tiling (with fixed tile sizes) of perfectly nested loops for both the sequential and parallel case [16, 9, 28, 29, 25, 5, 34, 15, 27]. Goumas et al. [11] presented a code generation technique for fixed-size parallel tiling of perfectly nested loops.

Parameterized sequential tiling has received much attention recently. Renganarayana et al. [26, 32] developed an effective code generation technique for parameterized tiling of perfectly nested loops , based on enumerating tile origins using a method called the "outset method." Later Kim et al. [20, 14] extended the approach to multi-level parameterized tiling of perfectly nested loops. Both these systems generate sequential tile code. Kim and Rajopadhye [19] recently developed a non-polyhedral approach to sequential parametric tiling of loop nests. Jimenez et al. developed techniques for register tiling of non-rectangular iteration spaces [17] and also a code generation technique for parameterized multi-level tiling of perfectly nested loops [18].

A script-based compositional transformation framework has been developed by Chen et al. [8]; their framework can be used for fixed-size tiling of imperfectly nested loops. Specialized frameworks [6, 30, 35] have been developed for fixed-size tiling of particular classes of imperfectly nested loops. Parallel tiling of general imperfectly nested loops was developed by Bondhugula et al. [4] in the Pluto [23] system, but tile sizes are restricted to be compile-time constants.

5 Conclusion

Tiling is a key loop transformation for coarse-grained parallelization as well as data locality optimization. Parametrically tiled code with symbolic tile size variables is of great interest since it enables empirical tuning and optimization on different target platforms. In this paper, a comparative experimental assessment was performed on three recently developed parametric tiling approaches.

Acknowledgment This work was funded in part by the U.S. National Science Foundation through awards 0541409, 0811457, 0811781, 0926687 and 0926688, by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915, and by the Army through contract W911NF-10-1-0004. We thank for the Ohio Supercomputer Center for access to their parallel systems.

References

1. N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *IJPP*, 29(5), Oct. 2001.
2. Workshop on Automatic Tuning for Petascale Systems. `http://cscads.rice.edu/workshops/summer08/autotuning`.
3. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *CGO*, April 2010.
4. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *PLDI*, 2008.
5. P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, 1994.
6. S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proc. Supercomputing '92*, pages 114–124, 1992.
7. 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.
8. 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.

9. S. Coleman and K. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *PLDI'95*, pages 279–290, 1995.
10. P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.
11. G. I. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Automatic parallel code generation for tiled nested loops. In *Symposium on Applied Computing*, pages 1412–1419, 2004.
12. A. Hartono, M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *ICS*, 2009.
13. A. Hartono, M. Baskaran, J. Ramanujam, and P. Sadayappan. Parametric tiled loop generation for effective parallel execution on multicore processors. In *IPDPS*, 2010.
14. HiTLoG: Hierarchical Tiled Loop Generator. `www.cs.colostate.edu/MMAlpha/tiling`.
15. K. Hogstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *SPAA*, pages 201–211, 1999.
16. F. Irigoin and R. Triolet. Supernode partitioning. In *PLDI*, 1988.
17. M. Jiménez, J. Llabería, and A. Fernández. Register tiling in nonrectangular iteration spaces. *ACM Trans. Program. Lang. Syst.*, 24(4):409–453, 2002.
18. M. Jiménez, J. Llabería, and A. Fernández. A cost-effective implementation of multilevel tiling. *IEEE Trans. Parallel Distrib. Syst.*, 14(10):1006–1020, 2003.
19. D. Kim and S. Rajopadhye. Parameterized tiling for imperfectly nested loops. Technical Report CS-09-101, Colorado State U., Dept. Computer Science, February 2009.
20. D. Kim, L. Renganarayanan, M. Strout, and S. Rajopadhye. Multi-level tiling: 'm' for the price of one. In *SC*, 2007.
21. 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.
22. PIP: The Parametric Integer Programming Library. `http://www.piplib.org`.
23. The Pluto automatic parallelizer. `sourceforge.net/projects/pluto-compiler`, 2010.
24. PrimeTile: A Parametric Multi-Level Tiler for Imperfect Loop Nests. `http://www.cse.ohio-state.edu/~hartonoa/primetile/`.
25. J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.
26. L. Renganarayana, D. Kim, S. Rajopadhye, and M. Strout. Parameterized tiled loops for free. In *PLDI'07*, pages 405–414, 2007.
27. L. Renganarayana and S. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *SC*, 2004.
28. G. Rivera and C. Tseng. Locality optimizations for multi-level caches. In *SC*, 1999.
29. R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Tech. Report 90.38, RIACS, NASA Ames Research Center, 1990.
30. Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI*, 1999.
31. A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. Scalable autotuning framework for compiler optimization. In *IPDPS '09*, May 2009.
32. TLoG: A Parametrized Tiled Loop Generator. `http://www.cs.colostate.edu/MMAlpha/tiling/`.
33. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *SC*, 1998.
34. J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
35. Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *J. Supercomput.*, 27(3):219–264, 2004.