Dynamic Selection of Tile Sizes

Sanket Tavarageri[1] Louis-Noël Pouchet[1] J. Ramanujam[2] Atanas Rountev[1] P. Sadayappan[1]

[1]Dept. of Computer Science and Engineering
The Ohio State University
2015 Neil Ave, Columbus, OH, USA
Email: {tavarage, pouchet, rountev, saday}@cse.ohio-state.edu

[2]Dept. of Electrical & Computer Engineering
Louisiana State University
Baton Rouge, LA, USA
Email: jxr@ece.lsu.edu

***Abstract*—Tiling is a key program transformation to achieve effective data reuse. But the performance of tiled programs can vary considerably with different tile sizes. Hence the selection of good tile sizes is crucial.**

Although there has been considerable research on analytical models for selecting tile sizes, they have not been shown to be effective in finding optimal tile sizes across a range of programs and target architectures. Auto-tuning is a viable alternative that is often used in practice, and involves the execution of different combinations of tile sizes in a systematic fashion to find the best ones. But this is sometimes infeasible — for instance when the program is to be run on unknown platforms (e.g., cloud environments).

We propose a novel approach for generating code to enable dynamic tile size selection, based on monitoring the performance of a few loop iterations. The selection operates at run time on the "production" run, without any *a priori* knowledge of the execution environment. We discuss the theory and implementation of a parametric tiled code generator that enables run-time tile size tuning and describe a search strategy to determine effective tile sizes. Experimental results demonstrate the effectiveness of the approach.

I. Introduction

Tiling [5]–[7], [17], [28], [33], [39], [41] is a key program transformation to achieve high data reuse and thereby realize high levels of performance in loop-oriented programs. The choice of tile sizes for loops has a significant impact on the execution time of such programs. Advances in *parameterized tiling*, where tile sizes are symbolic parameters [2], [14], [15], [18], [19], [21], [22], [29], [37], have enabled the generation of code that can be executed with different tile sizes in different environments. There has also been much interest in developing analytical models to determine optimal tile sizes as a function of architectural parameters (sizes and associativities of caches and TLBs, etc.) and code characteristics [8]–[10], [12], [13], [16], [30]–[32]. However, so far no analytical model has been shown to be effective in finding optimal tile sizes across a wide range of programs and target platforms. Therefore, the current state-of-practice for effective tile size selection is to use *auto-tuning* [3], [34], [36], [38], where an empirical search for the best tile sizes (often off-line) is carried out by running the program with different tile sizes, and the best performing configuration observed is retained. While ATLAS [38] is perhaps the most widely known auto-tuning framework, many other systems have demonstrated the effective use of auto-tuning for selection of tile sizes and other optimization parameters such as the degree of loop unrolling [3], [20], [23], [25], [27], [34], [35].

Although auto-tuning is effective for tile size optimization, it requires execution of a large number of tuning runs on the target platform prior to the actual "production" run of an application, and thus is more suited for the development of library packages, rather than for user code. Further, auto-tuning may not be feasible in contexts such as cloud computing [1], where users run their applications in a cost-effective manner in the "cloud," but the characteristics of the machine(s) on which the program is executed may not be known to the user beforehand. Depending on the number of other applications running on the same machine (a factor that is often beyond the control of application users), every run of the application may require different tile sizes for best performance. Furthermore, different runs of an application could be on different platforms. Auto-tuning based on empirical search of the space of possible tile sizes is thus infeasible in this context.

In this paper, we develop a novel approach for the generation of tiled code, so that different tile sizes are dynamically tried out as the actual "production" code (that is parametrically tiled) executes, and an effective set of tile sizes is determined on the fly. In other words, there is no need to rely on *a priori* empirical search and auto-tuning. Instead, the program observes and adapts its own run-time behavior, as a function of the (unknown) machine characteristics and system load. We propose new code generation techniques to enable on-the-fly variation of the tile sizes during program execution. The developed approach can handle affine multi-statement imperfectly nested loop nests. In addition, we present an algorithm for dynamic adaptation of tile sizes Our approach works by moving towards better tile sizes by monitoring the performance of a few iterations of the loop. We demonstrate that such a run-time adaptation is able to achieve near-optimal performance for a number of benchmarks.

The main contributions of the paper are as follows.

- ◇ The development of the theory and code generation techniques to allow variation of tile sizes during the execution of the program. To the best of our knowledge, this is the first solution for this code generation problem.
- ◇ The derivation of an adaptation strategy that dynamically varies the tile sizes in order to identify effective ones.

978-1-4577-1950-9/11/$26.00 ©2011 IEEE

The rest of the paper is organized as follows. In Section II, we motivate the work with an example. Section III presents the theory and code generation techniques that allow dynamic tile size variation. In Section IV, an adaptation strategy to achieve optimal performance using the tile size variation scheme is described. Experimental evaluations are presented in Section V. Related work is discussed in Section VI followed by conclusions in Section VII.

II. Motivation and Overview of Approach

We motivate the work using an example. Consider the tile selection problem for the dsyr2k kernel shown in Figure 1, which computes $C := \alpha AB^T + \alpha BA^T + \beta C$.

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    C[i][j] *= beta;
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < M; k++) {
      C[i][j] += alpha * A[i][k] * B[j][k];
      C[i][j] += alpha * B[i][k] * A[j][k];
    }
```

Fig. 1: dsyr2k Original Code

.

A parametrically tiled code variant of dsyr2k is shown in Figure 2.

```
for (it=ceil(-1+1/Ti); it<=floor(N/Ti-1/Ti);it++)
  for (jt=ceil(-1+1/Tj); jt<=floor(N/Tj-1/Tj);jt++)
    for (i=max((it*Ti),0); i<=min((it*Ti+Ti-1),N-1);i++)
      for (j=max((jt*Tj),0);j<=min((jt*Tj+Tj-1),N-1);j++)
        C[i][j]*=beta;
for (it=ceil(-1+1/Ti); it<=floor(N/Ti-1/Ti); it++)
  for (jt=ceil(-1+1/Tj); jt<=floor(N/Tj-1/Tj); jt++)
    for (kt=ceil(-1+1/Tk); kt<=floor(M/Tk-1/Tk); kt++)
      for (i=max(it*Ti, 0); i<=min(it*Ti+Ti-1, N-1); i++)
        for (j=max(jt*Tj, 0); j<=min(jt*Tj+Tj-1, N-1); j++)
          for (k=max(kt*Tk,0); k<=min(kt*Tk+Tk-1,M-1); k++)
            {
              C[i][j]+=alpha*A[i][k]*B[j][k];
              C[i][j]+=alpha*B[i][k]*A[j][k];
            }
```

Fig. 2: dsyr2k Parametrically Tiled Code

.

Table I shows the performance variation observed by empirical search over tile sizes in the range between 2 and 64 in powers of 2 $(2, 2^2, \ldots, 2^6)$ for each of the tile dimensions. Ti, Tj, Tk denote tile sizes for the three loop dimensions. The programs were compiled with Intel icc 12.0, with $N = 2500, M = 2500$, and were run on a single core of an Intel Xeon E5630 quad-core processor running at 2.53GHz with 32KB L1 cache.

The worst-case tile size tuple $(2 \times 8 \times 2)$ results in a slowdown of almost $4\times$ compared to the best tile sizes in this space, highlighting the importance of tile size selection. In scenarios where empirical auto-tuning is not feasible to determine optimal tile sizes, we face the challenging issue of determining effective tile sizes. In this paper, we present a machine-independent solution to this problem.

TABLE I: Parametric Tiling Extreme Performance

	Ti	Tj	Tk	Time (in s)
Best tile sizes	64	8	64	27.75
Worst tile sizes	2	8	2	105.93

We now illustrate the approach we develop. Figure Figure 3 shows the structure of a 2D parametrically tiled loop nest.

```
// Tile loops
for (it=lb_it; it <= ub_it; it++)
  for (jt=lb_jt; jt <= ub_jt; jt++)
    // Point loops
    for (i=f1(it, Ti);i<=f2(it, Ti);i++)
      for (j=f3(jt, Tj, i);j<=f4(jt, Tj, i);j++)
        // Task body
        { ... }
```

Fig. 3: Parametrically Tiled 2D Code Structure

Figure 4 shows the generated code structure of the tiled code using our approach to on-the-fly variation of tile sizes.

```
global_i_min=lexicographic_min_i;
// Tile loops
for (it=lb_it; it <= ub_it; it++) {
  start_timer;
  for (jt=lb_jt; jt <= ub_jt; jt++)
    // Point loops
    for (i=f5(it, Ti, global_i_min);i<=f2(it, Ti);i++)
      for (j=f3(jt, Tj, i);j<=f4(jt, Tj, i);j++)
        // Task body
        { ... }
  if (NIterations of it)
   {
      end_timer;
      RecordPerformance(Ti, Tj, exec_time);
      Ti_prime = FetchNewTileSizes();
      Tj_prime = FetchNewTileSizes();
      // update it, lb_it, ub_it, glboal_i_min
      Ti=Ti_prime;
      Tj=Tj_prime;
      start_timer;
   }
}
```

Fig. 4: Code with Dynamically Varying Tile Sizes

The essential idea is that the execution time is monitored for the current tile sizes and decisions are made to change the tile sizes based on the observations. The main problem is to suitably traverse the iteration space as tile sizes are changed during the actual loop execution, ensuring that no loop iteration is (i) omitted or (ii) executed multiple times, and no dependence is violated. The details of the approach are presented in the next section.

Using the dynamic adaptive approach we develop in this paper (labeled *EvolveTile*), the running times obtained for the dsyr2k example are shown in Table II, for the two cases of starting the adaptation from the best and worst possible tile sizes from Table I.

TABLE II: Performance with dynamic tiling

Ti	Tj	Tk	Original (in s)	*EvolveTile* (in s)
64	8	64	27.75	28.60
2	8	2	105.93	36.76

We observe that dynamically adapting the tile sizes speeds up performance for the worst possible static tile size by $2.9\times$, while the running time for the best static tile size remains almost unaffected (3% slowdown) with the adaptive scheme. This illustrates the potential profitability of the approach as well as the low overheads of the run-time tile size adaptation mechanism.

III. Formulation of Dynamic Tile Size Variation

Our approach considers the class of affine imperfectly nested loops. which we assume have been suitably pre-processed if necessary to make rectangular tiling legal [6], [39]. We first present an algebraic characterization for the generation of loop bounds for non-adaptive parametric tiling, using a notation similar to that used in prior work of Baskaran et al. [2]. We then develop the formulation for the modified lower and upper bounds of the tile iterators as the tile sizes are changed.

A. Parametric Tiling

Let $v_1, v_2, \ldots, v_n$ represent the loop iterators of 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 iteration 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. Such a formulation is possible since the loop bounds are constrained to be affine functions of the surrounding loop iterators and program parameters.

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, by construction 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 ($\vec{p}$) and constants.

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

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.

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 a 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:

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

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:

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

Two important properties of S_T (details may be found in [2]) 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.

Henceforth, the tile coordinates t_j are interchangeably referred to as tile iterators and intra-tile coordinates u_j as point iterators.

B. Loop bounds for adaptive tile sizes

Consider a loop with iterator v_k in the original untiled code, and the corresponding tile loop with iterator t_k and the point loop with iterator u_k in the tiled code. Suppose after a certain "complete" iteration of t_k, we want to *switch* the tile size s_k of t_k to s'_k. By a "complete" iteration we mean, the body of the *for* loop with iterator t_k has completed execution. We have,

$$v_k = s_k.t_k + u_k \wedge 0 \leq u_k \leq s_k - 1$$

Hence, iteration points up to $s_k.t_k + s_k - 1$ of v_k have been scanned at the time of *switch*. For the remaining program we have,

$$v_k \geq s_k.t_k + s_k$$

The system S of inequalities representing the remainder of the iteration domain of the program is given by the original

set of inequalities for the iteration domain along with $v_k \geq s_k.t_k + s_k$:

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

$$v_k - s_k.t_k - s_k \geq 0 \quad (2)$$

The new bounds for tile coordinates can now be computed with v_k expressed as

$$v_k = s'_k.t'_k + u'_k \wedge 0 \leq u_k \leq s'_k - 1 \quad (3)$$

where t'_k is the new tile-coordinate and u'_k, the new intra-tile coordinate for the loop variable v_k.
So, by combining (1), (2) and (3) in a single system we connect s_k to the new tile size s'_k, and eliminating point loop iterators in this system leads to a valid lower bound on t'_k to execute the remainder of the iteration domain.

It has to be noted that, by virtue of S being in *row echelon* form, only the bounds of t_k and the bounds of tile loops inner to t_k ($t_i, i \in [k..n]$) may be modified by the new S. The bounds of loops outer to k ($t_j, j \in [1..k-1]$) will not change. As an immediate consequence, at the time of *switch*, any of the tile sizes of the tile iterators inner to k may be changed but tile sizes of iterators outer to k *cannot* be changed.

C. Code Generation

At the time of *switching* the tile size s_k of tile iterator t_k to s'_k, recalculation of bounds for tile iterators and point iterators are handled in the following way. In the *switch code*, t_k is assigned the new lower bound and s_k is assigned the new tile size s'_k. Optionally, tile sizes of the loops inner to k may also be changed.

Doing only this however may result in rescanning of some of the scanned iteration points due to the relaxed projection performed during the elimination of intra-tile coordinates (described in §III-A). To avoid this, in the original system of inequalities S, for each of the loop variables v_k whose tile sizes may potentially be changed, a new inequality is introduced:

$$v_k \geq global\_v_k\_min$$

along with the original set of inequalities,

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

The $global\_v_k\_min$ is a symbolic constant and is initially set to lexicographic minimum for the loop variable v_k; this can be computed using a parametric integer linear programming solver such as PIP [11]. Tiled code is generated from this extended set of inequalities and at the time of *switch*, $global\_v_k\_min$ is set to $(s_k.t_k + s_k)$. Thus, since the check that the point iterator $u_k \geq global\_v_k\_min$ had already existed in the point loops, no points will be rescanned. Thus, the generated code is guaranteed to scan *all* the iteration points *only once*.

The tile size for the loop corresponding to v_k may be subsequently changed again when the condition $u_k \geq global\_v_k\_min$ becomes true always. This will be the case when the iteration points up to $(s_k.t_k + s_k - 1)$ which were scanned with the old tile size are visited with the new tile size also. In other words, when the inequality $(s'_k.t'_k + t'_k - 1) \geq (global\_v_k\_min - 1)$ holds, the tile size may be further adapted.

D. Example

We now illustrate the above steps using an example. Consider the input loop nest in Figure 5, and its tiled version as produced by the PTile algorithm in Figure 6. The result of the application of the above mentioned algorithms is shown in Figure 8, along with the necessary *switch code* to adapt the tile size at run-time.

Finally, to further illustrate the tiled execution with run-time tile size adaptation, Figure 7 shows an illustrative tiling of the iteration space that could result from dynamically varying the tile sizes.

IV. Tile Size Adaptation

We now discuss the use of previously described tile size variation technique to optimize performance. For dense iteration spaces, one observes that there are approximately the same number of iteration points between any two equal intervals in terms of iterator values of a loop. This fact can be exploited to gauge the performance of a given tile size by measuring the time it takes to execute a certain number of iterations of a tile loop instead of having to wait for the loop nest to completely finish running. Using the above parametric code generation scheme, one can switch to more promising tile sizes as the loop is continuing the execution.

A. Search Heuristic

Consider the loop nest shown in Figure 9 (I). The performance of the loop nest is clocked and the tile sizes are adjusted $NEvolvePoints$ number of times. Let such points in the program be called *EvolvePoints*. It may be noted that if this number is too low then there will be less room for adaptation. On the other hand, if this number is set too high then the overhead due to numerous adaptations of the tile sizes may over-compensate the benefits accrued due to maneuvering towards effective tile sizes. Section IV-B provides an estimation on the number of *EvolvePoints* to be executed in a program. At an *EvolvePoint*, the performance is recorded and a heuristic (one such is Figure 9 - II) is invoked to get the new tile sizes to switch to.

The performance of a loop is monitored every $NIterations$ iterations where $NIterations$ is given by $\frac{loop-lengh}{NEvolvePoints}$. Thus, the more the *EvolvePoints* there are, the intervals in which the performance is clocked is smaller and similarly, the lesser the *EvolvePoints*, the performance is metered at larger intervals.

In order to search rectangular tile spaces (i.e. the tile spaces where tile sizes of different dimensions may be different),

```
for (i=0;i<=N;i++)
  for (j=i;j<=N;j++) {
    // Tile body
    S1(i,j);
  }
```

Fig. 5: Initial Code

```
// Tile loops
for (it=ceil((-Ti+1)/Ti); it<=floor(N/Ti); it++)
  for (jt=ceil((it*Ti-Tj+1)/Tj); jt<=floor(N/Tj); jt++)
    // Point loops
    for (i=max(it*Ti, 0); i<=min(it*Ti+Ti-1, N); i++)
      for (j=max(jt*Tj, i); j<=min(jt*Tj+Tj-1, N); j++) {
        // Tile body
        S1(i,j);
      }
```

Fig. 6: Tiled Code

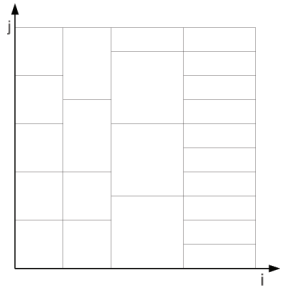

Fig. 7: Run-time Tile Sizes

```
// Tile loops
global_i_min = 0;
for (it=ceil((-Ti+1)/Ti); it<=floor(N/Ti);it++) {
  global_j_min = 0;
  for (jt=ceil((it*Ti-Tj+1)/Tj); jt<=floor(N/Tj);jt++) {
    // Point loops
    for (i=max(max(it*Ti, 0), global_i_min);
         i<=min(it*Ti+Ti-1, N);i++) {
      for (j=max(max(jt*Tj, i), global_j_min);
           j<=min(jt*Tj+Tj-1, N);j++) {
        // Tile body
        S1(i,j);
      }
    }
    /* Switch code for Tj begins */
    // Change Tj at the end of an iteration of jt
    if (switch_condition_for_jt == true) {
      global_j_min= jt*Tj+Tj;
      jt = ceil((jt*Tj+Tj+1-Tj_prime)/Tj_prime);
      jt = jt - 1; // 'for' loop increments iterator
      Tj = Tj_prime;// Tj_prime - new tile size for jt
    }
    /* 'Switch code for Tj ends */
  }
  /* Switch code for Ti begins */
  // Change Ti at the end of an iteration of it
  if (switch_condition_for_it == true) {
    global_i_min= it*Ti+Ti;
    it = ceil((it*Ti+Ti+1-Ti_prime)/Ti_prime);
    it = it - 1;// 'for' loop increments iterator
    Ti = Ti_prime; // Ti_prime - new tile size for it
    Tj = Tj_prime; // Tj_prime - new tile size for jt
    }
  /* Switch code for Ti ends */
}
```

Fig. 8: Tiled Adaptive Code

tile sizes of the loops are tuned from outermost to innermost loop and again tile size of the outermost loop is tuned and so on. This, as opposed to turning a particular loop completely and moving onto the next, is necessary as there may exist an interplay between tile sizes of different loop dimensions which is due to the fact that tiles with the same amount of data footprint may be realized with different shapes and one leads to better locality than the other [5]. Hence, the strategy to tune a loop $NTunings$ times and going on to tuning the next loop is adopted.

Algorithm shown in Figure 9 (II)- *Fetch_New_Tile_sizes* takes as input the current tile size and the performance queue which holds performance of the tile sizes examined so far. The search for effective tile sizes is bootstrapped by checking the performance of the current tile size, half of that and double that size. Notice that when the size of the performance queue is 1, double the input tile size is returned and when the size of the performance queue is 2, $\frac{1}{4}^{th}$ of the input tile size $CurrentT$ is returned because the original tile size was doubled in the previous iteration and thus it needs to be divided by 4 to get half of the original tile size.

Once the bootstrapping is done, the tile size *best tile size* which has performed by far the best is retrieved. The tile size just smaller than this is called the *left neighbor* and just greater than this is called the *right neighbor*. If the *best tile size* has no *left neighbor* then decreasing the tile size appears to lead to better performance. Hence the new tile size to be examined is half of the best found yet. On the other hand, if there is no *right neighbor* then increasing the tile size seems to be the promising way to go. Thus, the quest continues with doubling the tile size. If the performance is decreasing in either direction then the supposition is, the effective tile size lies between the current *best tile size* and one of its *neighbors*. The search moves towards that neighbor whose performance is closer to the *best tile size* performance. If the most effective tile size is the *best tile size* itself then after a very few *EvolvePoints*, the *neighbors* will come very close to the *best tile size* and the algorithm will start returning the *best tile size* in the subsequent calls.

B. Determining the number of EvolvePoints

In order to just be able to reach the most effective tile size (not known *a priori*) for a loop, all the tile sizes in the allowable range need to be searched, which is between the lower bound and upper bound of that loop in the original, untiled program. Intuitively, for larger ranges, more *Evolve-Points* are needed than for smaller ranges. Following the search strategy described in *Fetch_New_Tile_sizes* algorithm, we now derive the minimum number of *EvolvePoints* $NEvolvePoints$ that are required to reach the effective tile size starting from anywhere in the allowable range. This is, with the assumption that there is only one effective tile size and is constant. If there are more than one effective tile sizes then lesser *EvolvePoints* will be sufficient than given by the derivation. $NEvolvePoints$ is input to the loop nest depicted in Figure 9 (I).

(I) Loop nest employing run-time tile size variation

Input: Number of EvolvePoints: $NEvolvePoints$
Number of tunings at each level: $NTunings$
Tile sizes: $T_0, T_1 \ldots T_{n-1}$

$LoopLength \leftarrow (ub_0 - lb_0 + 1)$
$NIterations \leftarrow \frac{LoopLength}{NEvolvePoints}$
$CurrentTuningLevel \leftarrow 0$
$TuningsAtCurrentLevel \leftarrow 0$
$start\_time \leftarrow clock()$
for $it_0 = lb_0 \rightarrow ub_0$ **do**
  for $it_1 = lb_1 \rightarrow ub_1$ **do**
    $\ddots$
    for $it_{n-1} = lb_{n-1} \rightarrow ub_{n-1}$ **do**
      $task\_body$
    end for
  end for

  if $((it_0 - lb_0 + 1) \bmod NIterations = 0)$ **then**
    $end\_time \leftarrow clock()$
    $time \leftarrow end\_time - start\_time$
    $i \leftarrow CurrentTuningLevel$
    $RecordPerformance(PerfQueue, T_i, time)$
    $T_i' \leftarrow Fetch\_New\_Tile\_sizes(T_i, PerfQueue)$
    if $T_i' \neq T_i$ **then**
      $Update\ it_0, lb_0, ub_0, NIterations$
      $Increment\ TuningsAtCurrentLevel$
    else
      $TuningsAtCurrentLevel \leftarrow NTunings$
    end if
    if $TuningsAtCurrentLevel = NTunings$ **then**
      $Increment\ CurrentTuningLevel\ modulo\ n$
      $PerfQueue.clear()$
      $TuningsAtCurrentLevel \leftarrow 0$
    end if
    $start\_time \leftarrow clock()$
  end if
end for

(II) Fetch_New_Tile_sizes algorithm

Input: Current tile size: $CurrentT$, Performance data queue: $PerfQueue$
Output: New tile size: $NewT$
if $PerfQueue.size() = 1$ **then**
  $NewT \leftarrow CurrentT \times 2$
else if $PerfQueue.size() = 2$ **then**
  $NewT \leftarrow CurrentT/4$
else
  $BestTile \leftarrow FindBestTileSize(PerfQueue)$
  $LeftNeighbor \leftarrow FindLeftNeighbor(PerfQueue)$
  $RightNeighbor \leftarrow FindRightNeighbor(PerfQueue)$
  if $LeftNeighbor = nil$ **then**
    $NewT \leftarrow BestTile/2$
  else if $RightNeighbor = nil$ **then**
    $NewT \leftarrow BestTile \times 2$
  else
    if $Perf(LeftNeighbor) > Perf(RightNeighbor)$ **then**
      $NewT \leftarrow (BestTile + LeftNeighbor)/2$
    else if $Perf(LeftNeighbor) < Perf(RightNeighbor)$ **then**
      $NewT \leftarrow (BestTile + RightNeighbor)/2$
    else
      $NewT \leftarrow CurrentT$
    end if
  end if
end if
return $NewT$

Fig. 9: (I) The loop nest employing run-time tile size variation. (II) Fetch_New_Tile_sizes algorithm

Let the lexicographic minimum and maximum for a loop iterator v_i in the original iteration space be N_{i_min} and N_{i_max} respectively. The range of values the tile size for tile iterator t_i corresponding to iterator v_i may take are in the interval $[N_{i\_min}, N_{i\_max}]$ and length of the interval is,

$$N_{i\_range} = N_{i\_max} - N_{i\_min} + 1$$

In *Fetch_New_Tile_sizes* algorithm, the tile sizes are first varied in geometric progression and once the interval in which the effective tile size lies is found, binary search is used to approach the effective tile size. The bound on the number of steps it takes for each of these phases is $\lceil \log_2(N_{i\_range}) \rceil$.

Thus, the upper bound on the number of tiles to be explored for a given dimension is:

$$2 \times \lceil \log_2(N_{i\_range}) \rceil$$

For a loop nest with n loops, the search is carried out in an iterative manner and in each iteration, every loop is tuned $NTunings$ times. At the beginning of a tuning step, the performance of 3 tile sizes – i.e., the current tile size, half of that, double of that – are examined before making a decision on moving one way or the other. The two tile sizes explored – i.e., half of the initial tile size and double the initial size – constitute as necessary but not necessarily useful explorations in that these tile sizes may have already been explored in the previous iteration of tuning of the loop. Hence, the number of useful tile size variations in an iteration of tuning of a loop is $NTunings - 2$. Accounting for this bootstrapping, the minimum number of *EvolvePoints* is

$$NEvolvePoints = \sum_{i=1}^{n} \left\lceil \frac{2 \times \lceil \log_2(N_{i\_range}) \rceil}{NTunings - 2} \right\rceil \times NTunings \quad (4)$$

V. Experimental results

To evaluate the effectiveness of *EvolveTile*, we characterize the behavior of non-adaptive vs. adaptive tiling schemes for a large space of possible input tile sizes. Specifically, we

TABLE III: Benchmarks Used in the Experiments

	Description	Problem size
dsyrk	Symmetric rank k update	N=3000
dsyr2k	Symmetric rank $2k$ update	N=2500
fdtd-2d	2D Finite difference time domain method	TSTEPS=2000, N=2000
adi	Finite difference method for PDEs	TSTEPS=1500, N=1500
jacobi-2d	2D Jacobi method	TSTEPS=2500, N=2500

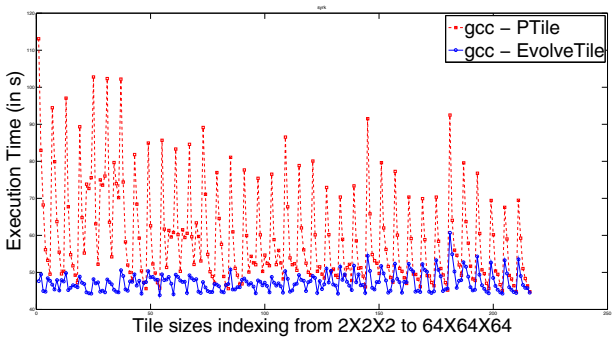

Fig. 10: Running times of dsyrk using GNU gcc

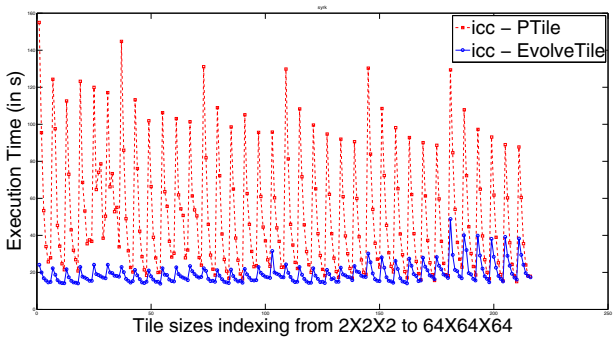

Fig. 11: Running times of dsyrk using Intel icc

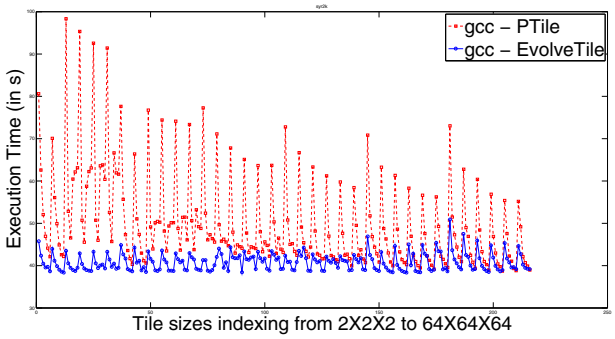

Fig. 12: Running times of dsyr2k using GNU gcc

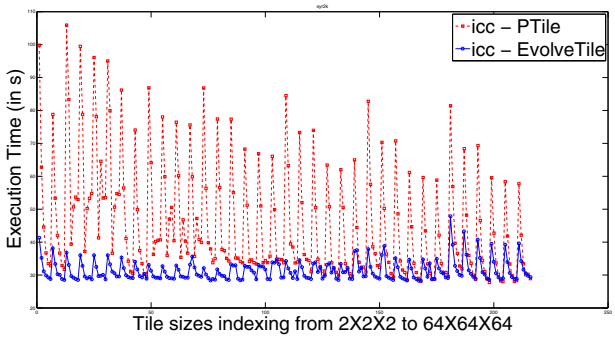

Fig. 13: Running times of dsyr2k using Intel icc

perform single-level tiling and compare the performance of the *EvolveTile* technique for dynamic tile size adaptation with the performance of a non-adaptive approach, implemented by the *PTile* system [2]. We note that all our experiments are done using a sequential execution of the programs.

The experiments were run using one core of an Intel Xeon E5630 quad-core processor, running at 2.53GHz with 32KB L1 cache. The tested configuration has 12GB of RAM and runs Linux kernel version 2.6.18-164.el5. Programs were compiled with GNU gcc 4.5.1, with `-O3` flag, and Intel icc 12.0.0, with `-fast` flag.

We used five benchmarks extracted from the PolyBench [26] suite. They are listed in Table III, with the specific problem sizes we have used. All the tested benchmarks have nested loops of depth 3, and *rectangular* tile sizes which are powers of 2 in the range $(2, 2, 2)$ to $(64, 64, 64)$, that is, a 3D tile size tuple is $(2^i, 2^j, 2^k)$ with i, j, k ranging from 2 to 6 are explored.

Figure 10 to 19 show the running time for each benchmark for different tile size tuples. Each point on the x axis corresponds to a tile size tuple, and the tile size tuples are ordered lexicographically. The y axis shows the execution time for the corresponding tile size combination. For *EvolveTile* the tile size tuple is the initial seed tile sizes it begins the execution with, and the tile sizes are adapted as the computation proceeds.

We note that the periodic spikes observed in the *PTile* running times correspond to large changes in the tile sizes. This is an effect of the lexicographic ordering of the tile size tuples on the x-axis; for example, $(4, 64, 64)$ is immediately followed by $(8, 2, 2)$.

The overall trend of the *EvolveTile* performance curve closely follows the *PTile* performance curve, but with a much smaller variability, and very often with a much lower execution time. This trend may be clearly spotted in the execution time characterization of fdtd-2d benchmark for compiler gcc,

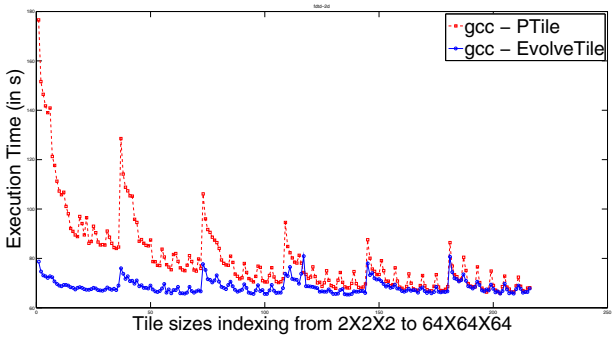

Fig. 14: Running times of fdtd-2d using GNU gcc

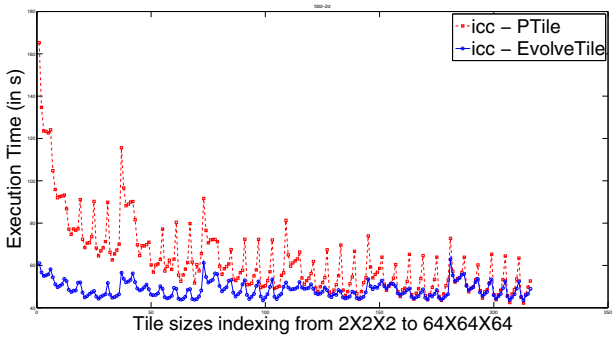

Fig. 15: Running times of fdtd-2d using Intel icc

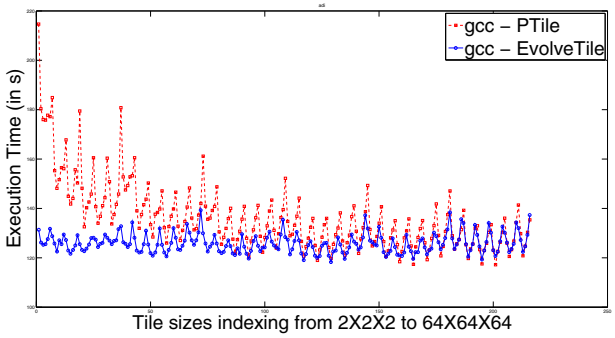

Fig. 16: Running times of adi using GNU gcc

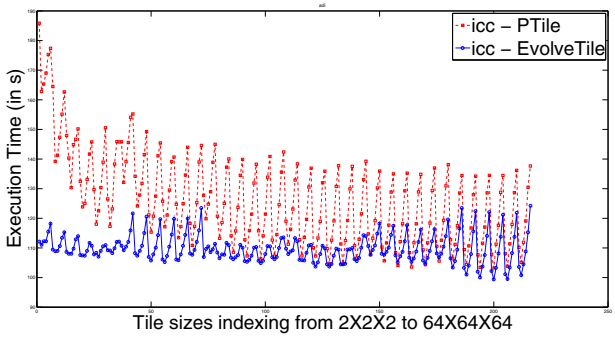

Fig. 17: Running times of adi using Intel icc

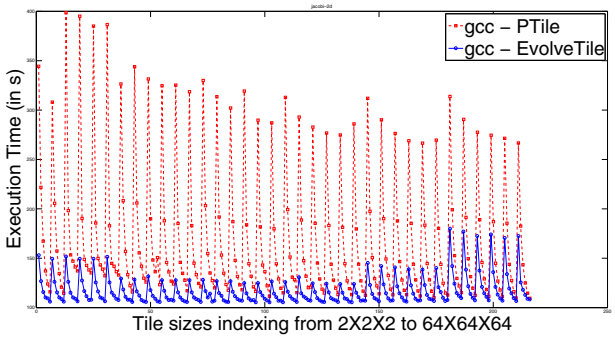

Fig. 18: Running times of jacobi-2d using GNU gcc

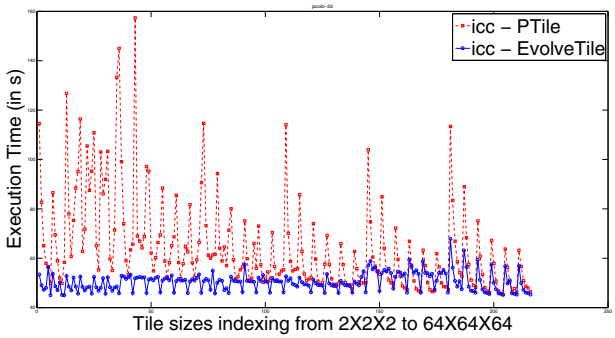

Fig. 19: Running times of jacobi-2d using Intel icc

shown in Figure 14. More specifically, the tile size combination which leads to the execution time spike in *PTile* also leads to a spike (although a much smaller one) in *EvolveTile*. This is because, during the search for effective tile sizes, a certain fraction of the total computation is performed using the seed tile size. Therefore, if the computation is started off with poorly performing tile sizes, then the performance of the first few iterations of the computation is greatly affected until the adaptation starts changing the tile sizes.

However, when the $NEvolvePoints$ is set to a value greater or equal to the lower bound given by Equation (4), most of the computation will be performed using adapted tile sizes found through the search algorithm proposed (*Fetch_New_Tile_sizes* algorithm). In our experiments, we have set $NEvolvePoints$ according to this rule.

We summarize our experimental results in Table IV and Table V, for the gcc and icc compilers respectively. We report the best and worst execution time found in the space analyzed (rectangular tile sizes from 2 to 64), as well as the average execution time of all the tested tile sizes.

We observe from Table IV and Table V the robustness of our approach in particular through the Average column. For

TABLE IV: Execution Time with GNU gcc

	Scheme	Best (in s)	Average (in s)	Worst (in s)
dsyrk	*PTile*	44.79	59.46	113.09
	EvolveTile	43.82	47.34	60.66
dsyr2k	*PTile*	38.72	49.99	98.33
	EvolveTile	38.39	40.82	50.91
fdtd-2d	*PTile*	65.90	80.30	176.59
	EvolveTile	65.31	68.62	80.91
adi	*PTile*	117.19	136.11	214.67
	EvolveTile	118.32	125.87	139.27
jacobi-2d	*PTile*	104.22	170.27	398.57
	EvolveTile	105.34	116.87	179.86

TABLE V: Execution Time with Intel icc

	Scheme	Best (in s)	Average (in s)	Worst (in s)
dsyrk	*PTile*	15.02	50.46	155.00
	EvolveTile	13.94	19.21	48.87
dsyr2k	*PTile*	27.75	45.65	105.93
	EvolveTile	28.31	31.37	47.83
fdtd-2d	*PTile*	42.21	62.62	165.24
	EvolveTile	43.50	48.65	62.86
adi	*PTile*	103.21	126.17	185.87
	EvolveTile	99.41	109.67	124.26
jacobi-2d	*PTile*	45.15	64.19	157.35
	EvolveTile	45.01	50.49	67.85

instance with gcc, the difference between the best tile sizes and the average across the space using *PTile* ranges from 16% to 55%, while using *EvolveTile* this difference narrows down to less than 10%, and often below 5%. This in conjunction to the fact that performance of the best tile sizes is similar for both schemes shows that tile-size adaptation approach can compensate with great success ineffective seed tile sizes, and correctly adapt at run time the tile sizes towards a very good solution. Similar observations are done when using icc, as shown in Table V: for all but dsyrk the difference between the best and average numbers for *EvolveTile* is around 10%, a number which is from 2 to 8 times smaller than with *PTile*, while again observing a minimal difference between the best execution time for *PTile* and *EvolveTile*.

Finally, we also report the overall best running times in Table VI when all the tile sizes which are powers of 2 in the range from 2 to the benchmark problem size are considered, for all benchmarks. This space is much larger than the previous tile size space considered, and spans the entire possibilities of power-of-two tile sizes. We observe only a marginal benefit in increasing the space size, at most a 5% performance improvement. Nevertheless our observations on the low overhead of the adaptive scheme are confirmed, since *EvolveTile* performs at worst 5% slower than *PTile* when using directly the best tile size. Interestingly, for adi with the icc compiler we observe that *EvolveTile* performs even better than *PTile*. This is because the search algorithm in *EvolveTile* is not limited to using only the tile sizes that are power of two, as shown in Figure 9(II), and for this specific case the best working tile size that *EvolveTile* computed during the execution was better than any power-of-two tile sizes tested with *PTile*.

TABLE VI: Best Execution Time Obtained

	Tested Tile sizes	Scheme	gcc (in s)	icc (in s)
dsyrk	(2, 2, 2) to	*PTile*	43.35	12.97
	(2048, 2048, 2048)	*EvolveTile*	43.41	13.51
dsyr2k	(2, 2, 2) to	*PTile*	37.77	27.04
	(2048, 2048, 2048)	*EvolveTile*	37.81	27.84
fdtd-2d	(2, 2, 2) to	*PTile*	64.12	42.21
	(1024, 1024, 1024)	*EvolveTile*	64.08	43.29
adi	(2, 2, 2) to	*PTile*	117.19	103.21
	(1024, 1024, 1024)	*EvolveTile*	117.99	98.54
jacobi-2d	(2, 2, 2) to	*PTile*	104.22	43.82
	(2048, 2048, 2048)	*EvolveTile*	104.28	44.43

VI. Related Work

Exploiting data locality is a key issue in achieving high performance and tiling has been widely used to improve data locality in loop nests. Since, the choice of tile sizes can greatly influence the realized performance, a number of researchers have addressed the problem of selecting good tile sizes based on cache-based performance models [4], [8]–[10], [12], [13], [24], [31], [40]. Hsu and Kremer [16] provide a comparative study of tile size selection algorithms. Recently, Yuki et al. [42] have explored the automatic creation of cubic tile size models.

Several works have used search-based techniques for finding tile sizes in an auto-tuning environment for known target machines [3], [23], [34], [35], [38]. The ATLAS system employs extensive empirical tuning to find the best tile sizes for different problem sizes in the BLAS library; tuning is done once at installation. Kisuki et al. [23] have used different techniques such as genetic algorithms and simulated annealing to manage the size of the search space. Tiwari et al. [35] note: "a key challenge that faces auto-tuners, especially as we expand the scope of their capabilities, involves scalable search among alternative implementations." The Active Harmony project [34], [35] uses several different algorithms to reduce the size of the search space such as the Nelder-Mead simplex algorithm. However, auto-tuning is not suitable for scenarios like cloud computing, where the characteristics of the target platforms and the load due to other jobs cannot be known before hand.

VII. Conclusion

In this paper, we have developed an approach to dynamically varying tile sizes as a tiled loop nest executes, to find effective tile sizes for the execution environment. We presented the theory behind the code generation techniques required to enable such a dynamic variation of tile sizes. A tile-size optimizing algorithm was also proposed that uses the tile size adaptation capability to move towards effective tile sizes starting from an initial set of seed tile sizes. Experimental results on several benchmarks demonstrate that the approach is robust and effective across a wide range of seed tile sizes.

ACKNOWLEDGMENT

We thank the reviewers for their feedback and suggestions that have helped us improve the presentation of the paper. This work was supported in part by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915, and the U.S. National Science Foundation through awards 0811457, 0811781, 0926687 and 0926688.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.

[2] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *CGO*, April 2010.

[3] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC. In *Proc. ACM International Conference on Supercomputing*, pages 340–347, 1997.

[4] F. Bodin, W. Jalby, D. Windheiser, and C. Eisenbeis. A quantitative algorithm for data locality optimization. In *Code Generation*, pages 119–145, 1991.

[5] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International conference on Compiler Construction (ETAPS CC)*, Apr. 2008.

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

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

[8] J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 492–499, New York, NY, USA, 1999. ACM Press.

[9] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290, New York, NY, USA, 1995. ACM Press.

[10] K. Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, Sep 1993.

[11] P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.

[12] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Languages and Compilers for Parallel Computing*, pages 328–343, 1991.

[13] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, 1999.

[14] 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.

[15] HiTLoG: Hierarchical Tiled Loop Generator. www.cs.colostate.edu/MMAlpha/tiling.

[16] C. Hsu and U. Kremer. A quantitative analysis of tile size selection algorithms. *J. Supercomput.*, 27(3):279–294, 2004.

[17] F. Irigoin and R. Triolet. Supernode partitioning. In *PLDI*, 1988.

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

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

[20] N. Kalinnik, M. Korch, and T. Rauber. An efficient time-step-based self-adaptive algorithm for predictor-corrector methods of runge-kutta type. *Journal of Computational and Applied Mathematics*, In Press, Corrected Proof:–, 2011.

[21] D. Kim and S. Rajopadhye. Parameterized tiling for imperfectly nested loops. Technical Report CS-09-101, Colorado State U., Dept. Computer Science, February 2009.

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

[23] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[24] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[25] J. Li, G. Tan, and M. Chen. Automatically tuned dynamic programming with an algorithm-by-blocks. *Parallel and Distributed Systems, International Conference on*, 0:452–459, 2010.

[26] PolyBench: Polyhedral Benchmark suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

[27] A. Qasem, J. Guo, F. Rahman, and Q. Yi. Exposing tunable parameters in multi-threaded numerical code. In *Proceedings of the 2010 IFIP international conference on Network and parallel computing*, NPC'10, pages 46–60, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.

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

[30] L. Renganarayana and S. Rajopadhye. Positivity, posynomials and tile size selection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 55:1–55:12, Piscataway, NJ, USA, 2008. IEEE Press.

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

[32] V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 146–153, Washington, DC, USA, 2000. IEEE Computer Society.

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

[34] C. Tapus, I. Chung, and J. K. Hollingsworth. Active harmony: towards automated performance tuning. In *SC*, pages 1–11, 2002.

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

[36] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable autotuning framework for computer optimization. In *IPDPS'09*, Rome, May 2009.

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

[38] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[39] M. Wolf. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, 1989.

[40] M. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91*, pages 30–44, 1991.

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

[42] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. Eichenberger, and K. O'Brien. Automatic creation of tile size selection models. In *CGO*, pages 190–199, 2010.