Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors

Muthu Manikandan Baskaran[1] Nagavijayalakshmi Vydyanathan[1] Uday Kumar Bondhugula[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
{baskaran,vydyanat,bondhugu,rountev,saday}@cse.ohio-state.edu

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

Abstract

Recent advances in polyhedral compilation technology have made it feasible to automatically transform affine sequential loop nests for tiled parallel execution on multi-core processors. However, for multi-statement input programs with statements of different dimensionalities, such as Cholesky or LU decomposition, the parallel tiled code generated by existing automatic parallelization approaches may suffer from significant load imbalance, resulting in poor scalability on multi-core systems. In this paper, we develop a completely automatic parallelization approach for transforming input affine sequential codes into efficient parallel codes that can be executed on a multi-core system in a load-balanced manner. In our approach, we employ a compile-time technique that enables dynamic extraction of inter-tile dependences at run-time, and dynamic scheduling of the parallel tiles on the processor cores for improved scalable execution. Our approach obviates the need for programmer intervention and re-writing of existing algorithms for efficient parallel execution on multi-cores. We demonstrate the usefulness of our approach through comparisons using linear algebra computations: LU and Cholesky decomposition.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Run-time environments, Optimization

General Terms Algorithms, Performance

Keywords Compile-time optimization, Dynamic scheduling, Run-time optimization

1. Introduction

The ubiquity of multi-core 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 parallel implementations of a large number of existing sequential codes. Therefore support from compilers and run-time systems for the development of parallel applications for multi-cores will be extremely important.

The starting point of the work reported in this paper is *Pluto*, a recently developed automatic parallelization system for multi-cores [34, 8, 7, 9, 6]. The key to Pluto's approach is the use of the polyhedral model [3, 36, 29, 25, 20, 37, 4] for representing dependences and transformations. The polyhedral model provides a powerful abstraction to reason about transformations of collections of loop nests by viewing dynamic instances (iterations) of each statement as integer points in a well-defined space called the statement's *polytope*. With such a representation for each statement and a precise characterization of inter or intra-statement dependences, it is possible to reason about the correctness of complex loop transformations in a completely mathematical setting using machinery from linear algebra and linear programming. With the conventional abstractions for data dependences used in most optimizing compilers (including gcc and all vendor compilers), it is extremely difficult to perform integrated model-driven optimization using key loop transformations like permutation, skewing, tiling, unrolling, and fusion across multiple loop nests.

Given input sequential code, Pluto can automatically generate parallel OpenMP code for multi-core processors and locality-optimized tiled code for sequential execution. Even for imperfectly nested multi-statement codes such as Cholesky decomposition or LU decomposition, Pluto can automatically generate tiled parallel programs with a parallel tiled execution structure similar to that found in LAPACK routines. However, as highlighted in recent work at the University of Tennessee [18, 12, 11], the LAPACK codes for several linear algebra functions exhibit loss of efficiency on multi-core systems due to excessively constraining inter-task barrier synchronization. This problem is being addressed by Dongarra's group's PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures) project [33], by developing a run-time scheduling framework and manual rewriting of LAPACK routines to use dynamic scheduling for improved scalability. The main problem addressed in this paper is the following: Can we develop a completely automatic parallelization approach that can transform input sequential codes (with affine dependences) for asynchronous, load-balanced parallel execution?

We propose a novel technique that solves this key problem for Pluto's compile-time parallelization approach, and as a result significantly improves load-balance for execution on multi-core systems. In particular, we develop a compile-time approach to enable run-time extraction of inter-tile data dependences, and subsequent dynamic scheduling of tiles on to processor cores. To the best of our knowledge, this is the first work to develop an automatic parallelization approach with compile-time generation of code to be executed at run-time to extract inter-task dependences that are used for dynamic scheduling and load balancing. The proposed technique could potentially be applied to other parallelization approaches based on the polyhedral model, and could eliminate a fundamental weakness of these purely-compile-time approaches with respect to load imbalance and resource under-utilization.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.
Copyright © 2009 ACM 978-1-60558-397-6/09/02. . . $5.00

The rest of the paper is organized as follows. Section 2 introduces the polyhedral model for representing programs, dependences, and transformations. Section 3 presents our novel approach for generating effective parallel tiled code through dynamic scheduling of tasks in a multi-core system. The performance improvements achieved using this approach are illustrated in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. Background

There has been significant progress over the last two decades in the development of powerful compiler frameworks for dependence analysis and transformation of loop computations with affine bounds and affine array access functions [3, 36, 29, 25, 19, 37, 4]. Such program regions are typically the most computation-intensive components of scientific and engineering applications, and they appear often in important real-world code [5]. For such regular code, compile-time optimization approaches have been developed using a polyhedral abstraction of programs and dependences. Although the polyhedral model of dependence abstraction and program transformation is much more powerful than the traditional models currently used in production optimizing compilers, early polyhedral approaches were not practically efficient. Recent advances in code generation [37, 4, 46] have addressed many of these issues, resulting in polyhedral techniques being applied to codes representative of real applications such as the spec2000fp benchmarks. CLooG [4, 15] is a powerful state-of-the-art code generator that captures most of these advances. Building on these developments, we have developed the Pluto compiler framework that enables end-to-end automatic parallelization and locality optimization of affine programs for general-purpose multi-core targets [8, 7, 9, 6]. The effectiveness of this transformation system has been demonstrated on a number of non-trivial application kernels for multi-core processors, and the entire system implementation is publicly available [34].

This section provides background information on the polyhedral model together with a brief overview of Pluto.

2.1 Overview of Polyhedral Model

A hyperplane is an $n-1$ dimensional affine subspace of an n-dimensional space and can be represented by an affine equality. A halfspace consists of all points of an n-dimensional space that lie on one side of a hyperplane (including the hyperplane); it can be represented by an affine inequality. A polyhedron is the intersection of finitely many halfspaces. A polytope is a bounded polyhedron.

In the polyhedral model, a statement s surrounded by m loops is represented by an m-dimensional polytope, referred to as an iteration space polytope. The coordinates of a point in the polytope (referred to as the iteration vector $\vec{i_s}$) correspond to the values of the loop indices of the surrounding loops, starting from the outermost one. In this work we focus on regular programs where loop bounds are affine functions of outer loop indices and global parameters (e.g., problem sizes). Similarly, array access functions are also affine functions of loop indices and global parameters. Hence the iteration space polytope $\mathcal{D}_s$ can be defined by a system of affine inequalities derived from the bounds of the loops surrounding s. Each point of the polytope corresponds to an instance of statement s in program execution. Using matrix representation to express systems of affine inequalities, the iteration space polytope is defined by

$$D_s.\begin{pmatrix} \vec{i_s} \\ \vec{n} \\ 1 \end{pmatrix} \geq \vec{0}$$

where D_s is a matrix representing loop bound constraints and $\vec{n}$ is a vector of global parameters.

Affine array access functions can also be represented using matrices. If $a[\mathcal{F}_{ras}(\vec{i_s})]$ is the rth reference to an array a in statement s with a corresponding iteration vector $\vec{i_s}$, then

$$\mathcal{F}_{ras}(\vec{i_s}) = F_{ras}.\begin{pmatrix} \vec{i_s} \\ \vec{n} \\ 1 \end{pmatrix}$$

where F_{ras} is a matrix representing an affine mapping from the iteration space of statement s to the data space of array a. Each row in the matrix defines a mapping corresponding to a dimension of the data space.

Example. Consider the code in Figure 1(a). The iteration space polytope of statement Q is defined by $\{i, j \mid 0 \leq i \leq N-1 \ \wedge \ 0 \leq j \leq N-1\}$. In matrix representation, this polytope is given by

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 \end{pmatrix} . \begin{pmatrix} \vec{i_Q} \\ N \\ 1 \end{pmatrix} \geq \vec{0}$$

where $\vec{i_Q} = \binom{i}{j}$ is the iteration vector of statement Q. The access function of the reference to array a in statement Q is represented as

$$\mathcal{F}_{1aQ}(\vec{i_Q}) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \vec{i_Q} \\ N \\ 1 \end{pmatrix}$$

One of the key transformations for such affine code is *tiling*. When tiling is performed, in the tiled iteration space, statement instances are represented by higher dimensional statement polytopes involving *supernode* iterators and *intra-tile* iterators. The code in Figure 1(b) represents the tiled version of the code in Figure 1(a). The original iteration space and the transformed iteration space are illustrated in Figure 1(c).

Dependences. There has been a significant body of work on dependence analysis in the polyhedral model [19, 36, 47]. An instance of statement s, corresponding to iteration vector $\vec{i_s}$ within iteration domain D_s, depends on an instance of statement t (with iteration vector $\vec{i_t}$ in domain D_t), if (1) $\vec{i_s}$ and $\vec{i_t}$ are valid points in the corresponding iteration space polytopes, (2) they access the same memory location, and (3) $\vec{i_s}$ is executed before $\vec{i_t}$. Since array accesses are assumed to be affine functions of loop indices and global parameters, the constraint that defines conflicting accesses of memory locations can be represented by an affine equality (obtained by equating the array access functions in source and target statement instances). Hence all constraints to capture a data dependence can be represented as a system of affine inequalities/equalities with a corresponding polytope (referred to as a *dependence polytope*). The dependence polytope is defined by

$$\left(\begin{array}{cc} D_s & 0 \\ 0 & D_t \\ \hline -Id & H \end{array}\right) . \begin{pmatrix} \vec{i_s} \\ \vec{i_t} \\ \vec{n} \\ 1 \end{pmatrix} \left(\begin{array}{c} \geq \vec{0} \\ \hline = \vec{0} \end{array}\right)$$

where Id represents an identity matrix, and H (referred to as the *h-transformation* of the dependence) relates the target statement instance to a source statement instance that last accessed the conflicting memory location:

$$H.\begin{pmatrix} \vec{i_t} \\ \vec{n} \\ 1 \end{pmatrix} = \begin{pmatrix} \vec{i_s} \\ \vec{n} \\ 1 \end{pmatrix}$$

Schedules. Using the polyhedral model to find (affine) program transformations has been widely used for improvement of sequen-

```
for (i=0;i<N;i++) {
  P: x[i]=0;
  for (j=0;j<N;j++)
    Q: x[i]+=a[j][i]*y[j];
}
```

(a) Original code

$$D_Q^{orig}\cdot\begin{pmatrix} i\\ j\\ N\\ 1\end{pmatrix}\geq\vec{0} \qquad D_Q^{tiled}\cdot\begin{pmatrix} it\\ jt\\ i\\ j\\ N\\ 1\end{pmatrix}\geq\vec{0}$$

(c) Original and tiled iteration space

```
for (it=0; it<=floord(N−1,32);it++) {
  for (jt=0;jt<=floord(N−1,32);jt++) {
    if (jt == 0) {
        for (i=max(32*it,0);  i<=min(32*it+31,N−1); i++) {
          P: x[i]=0;
          Q: x[i]=x[i]+a[0][i]*y[0];
        }
    }
    for (i=max(32*it,0);  i<=min(32*it+31,N−1); i++) {
      for (j=max(32*jt,1); j<=min(32*jt+31,N−1);j++) {
        Q: x[i]=x[i]+a[j][i]*y[j];
      }
    }
  }
}
```

(b) Tiled code

Figure 1. Transpose matrix vector multiply (tmv) kernel

tial programs (source-to-source transformation) as well as automatic parallelization of programs [20, 29, 25, 22, 25, 8]. An affine transformation of a statement s is defined as an affine mapping that maps an instance of s in the original program to an instance in the transformed program. The affine mapping function of a statement s is given by

$$\phi_s(\vec{i_s}) = \mathcal{C}_s\cdot\begin{pmatrix}\vec{i_s}\\ \vec{n}\\ 1\end{pmatrix}$$

When $\mathcal{C}_s$ is a row vector, the affine mapping ϕ_s is a one-dimensional mapping. An m-dimensional mapping can be represented as a combination of m (linearly independent) one-dimensional mappings, in which case $\mathcal{C}_s$ is a matrix with m rows. An affine transformation is valid only if it preserves the dependences in the original program. A number of different approaches have been defined for constructing such mappings. For example, Feautrier [20, 21] defines affine time schedule, which is one-dimensional (single sequential loop in the transformed program) or multi-dimensional (nested sequential loops in the program). The schedule associates a timestamp to each statement instance. Instances are executed in increasing order of timestamps to preserve data dependences. Two statement instances that have the same timestamp can be executed in parallel.

2.2 PLUTO

Pluto [34] is a state-of-the-art automatic parallelization system that optimizes sequences of imperfectly nested loops, simultaneously for parallelism and locality, through tiling transformations. Given an input sequential code, it can automatically generate tiled parallel OpenMP code for multi-core processors. As a first step, the input program is run through a scanner and parser that constructs an abstract syntax tree. Polytopes are then extracted from the source code. After analyzing the dependences, communication-minimal and locality-optimized tiling transformations are determined through Pluto's transformation framework. Then suitable input, in the form of description of all statements, together with their iteration spaces (as polytopes) as well as the transformations (as scheduling functions) specifying the new execution order for each statement instance, is fed to the CLooG code generator [4, 15]. The union of all input iteration space polytopes is scanned by CLooG according to the specified scheduling functions, in order to generate loop nests in the target program that execute the statement instances in this new execution order. Loops that are determined by Pluto to be parallel are translated with appropriate OpenMP directives for parallelism.

3. Approach for Compiler-Assisted Dynamic Scheduling

Recent work at the University of Tennessee [18, 12, 11] with LAPACK codes for several linear algebra functions highlights two key challenges for effective parallelization of such codes. First, effective use of modern multi-core hardware requires the introduction of *tasks* that operate on small portions of data in order to improve data locality. For affine code, fully automatic introduction of such tasks can be easily done with general polyhedral transformation tools such as Pluto, or with similar semi-automatic approaches such as [24]. The tiles generated by Pluto naturally correspond to such tasks, as they are defined through a polyhedral-based cost model with the explicit goal of reducing communication by finding profitable directions for the tiling hyperplanes. (For the rest of the paper, we will use "task" and "tile" interchangeably.) A second critical issue highlighted in [18, 12, 11] is that of *asynchronicity*: the presence of synchronization points has significant negative impact on the performance of the parallel implementations. Their PLASMA project [33] addresses this problem through a run-time scheduling framework and *manual* rewriting of LAPACK routines to use dynamic scheduling for improved scalability.

For automatic transformation frameworks such as Pluto, the generated parallel code (e.g., OpenMP parallel loops) contains barriers that can lead to excessively constrained inter-task synchronization. This problem cannot be solved by any purely-compile-time scheduling approach. Thus, the benefits of automatic, general, and effective parallelization in the polyhedral model cannot be fully realized. This fundamental weakness of these parallelization approaches presents a significant challenge, since it is imperative to effectively schedule the parallel tiles on the processor cores to avoid load imbalance and resource under-utilization.

We propose a novel fully-automatic approach for generating efficient parallel code that can be executed on a multicore system in an asynchronous, load-balanced manner. Our approach generates, at compile-time, additional program code whose role at run-time is to generate a directed acyclic graph (DAG) of tasks and their dependencies, and analyze the DAG to facilitate dynamic scheduling of the tasks on the processor cores for improved scalable execution. The key insight behind this idea is that the DAG-generating code can be generated at compile-time by constructing a dependence polytope that captures the inter-tile dependences. The DAG-generating code is generated in such a way that, at run-time, it would traverse the points in this polytope. Each such point is essentially a pair of inter-dependent tiles and thus represents an edge in the task dependence DAG.

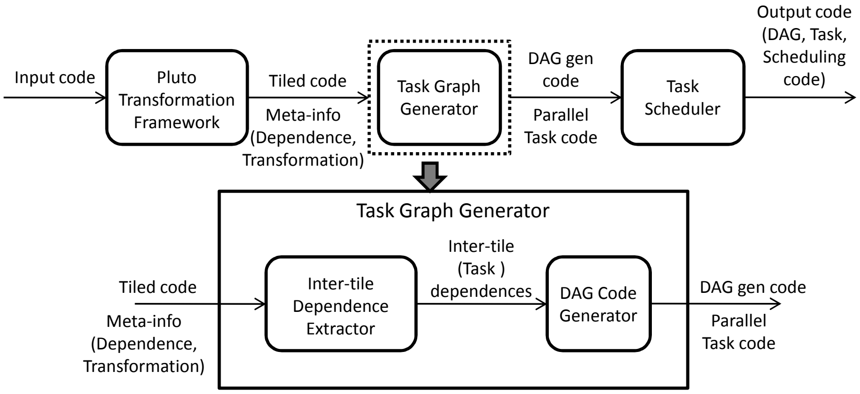

Input code
Pluto Transformation Framework
Tiled code
Meta-info (Dependence, Transformation)
Task Graph Generator
DAG gen code
Parallel Task code
Task Scheduler
Output code (DAG, Task, Scheduling code)
Task Graph Generator
Tiled code
Meta-info (Dependence, Transformation)
Inter-tile Dependence Extractor
Inter-tile (Task) dependences
DAG Code Generator
DAG gen code
Parallel Task code

Figure 2. Enabling load-balanced execution on multi-core systems

The developed system is illustrated in Figure 2. The *task graph generator* identifies the tile to be executed by a processing unit at a given time, automatically determines inter-tile dependence information, and generates code that at run-time generates a DAG representing these dependences. The *task scheduler* adds code that at run-time analyzes the task dependence DAG and infers priorities for dynamically scheduling the tasks. Thus, the input source code is transformed into code encompassing (1) a task code segment (core computation code) to be executed by a processor core, (2) a task dependence DAG generation code segment, and (3) a task scheduling code segment. The run-time execution of the transformed code generates the DAG, analyzes it to infer priorities to be used for scheduling, and executes the tiles on the processing units based on these priorities, maintaining load balance across the processor cores.

3.1 Task Graph Generator

The task graph generator component is developed on top of Pluto. As mentioned in Section 2, given an input sequential code, Pluto generates locality-optimized tiled code. The resulting tiles can be effectively scheduled on the processing units by using our dynamic scheduling approach, as opposed to using the compile-time affine scheduling currently employed by Pluto. This component has two sub-components: an *inter-tile dependence extractor* and a *DAG code generator*.

3.1.1 Inter-tile Dependence Extractor

A dependence polytope captures dependences involving pairs of statement instances accessing a common reference. It is represented as a system of inequalities and equalities capturing the domains of the statements involving the references, affine functions of the references, and ordering imposed by the dependence. In the tiled iteration space, statement instances are represented by higher dimensional statement polytopes involving supernode iterators and intra-tile iterators. Similarly, a dependence between two references in the tiled iteration space is captured by a higher dimensional dependence polytope – it represents a dependence between iterations belonging to the same tile or different tiles. The polytope that characterizes dependences between iterations in the tiled domain can be generated with the following information:

1. Inequalities describing the iteration spaces of the source and target statement in the original domain.
2. Inequalities defining a tile of the source statement and that defining a tile of the target statement, given by the affine tiling transformation from Pluto.
3. Equalities relating the source statement iterators and target statement iterators with respect to the dependence (h-transformation of the dependence).

Let D_s and $\vec{x_s}$ represent the iteration space matrix and iteration vector, respectively, of a statement s in original domain. Let DT_s represent the iteration space matrix of the statement in the tiled domain, derived from the tiling transformation generated by Pluto. DT_s embeds information that defines a tile of the statement (T_s) and also that defines the original domain of the statement (D_s). Let $x\vec{T}_s$ represent the iteration vector of supernode iterators. Then the domain of the statement in the tiled iteration space is given by

$$DT_s \cdot \begin{pmatrix} x\vec{T}_s \\ \vec{x_s} \\ \vec{n} \\ 1 \end{pmatrix} \geq \vec{0}, \quad \text{where} \quad DT_s = \begin{pmatrix} T_s & 0 \\ 0 & D_s \end{pmatrix}$$

If there exists a dependence between two statements s and t, and if H represents the h-transformation of the dependence, then the dependence polytope in the tiled domain is given by

$$\left(\begin{array}{cccc} T_s & 0 & 0 & 0 \\ 0 & D_s & 0 & 0 \\ 0 & 0 & T_t & 0 \\ 0 & 0 & 0 & D_t \\ \hline 0 & -Id & 0 & H \end{array}\right) \cdot \begin{pmatrix} x\vec{T}_s \\ \vec{x_s} \\ x\vec{T}_t \\ \vec{x_t} \\ \vec{n} \\ 1 \end{pmatrix} \left(\begin{array}{c} \geq \vec{0} \\ \hline = \vec{0} \end{array}\right)$$

In our approach for dynamic scheduling of tiles on multi-core parallel systems, we are interested in dependences between tiles, i.e. dependences between iterations belonging to different tiles, to define a dependence preserving schedule of tiles across processor cores. The basic idea to derive inter-tile dependence from a dependence polytope in the tiled domain is to project out the dimensions belonging to intra-tile iterators from the dependence polytope to derive a system of inequalities/equalities involving only inter-tile or supernode iterators. The projection of intra-tile dimensions is done using Fourier-Motzkin elimination. This system is further projected to eliminate tiling dimensions that do not involve in the distribution of tiles across processor cores. The projection procedure is repeated for all dependence polytopes in the tiled domain. A projected dependence polytope has the form

$$\begin{pmatrix} D'_s & 0 \\ 0 & D'_t \end{pmatrix} \cdot \begin{pmatrix} x\vec{T}'_s \\ x\vec{T}'_t \\ \vec{n} \\ 1 \end{pmatrix} \geq \vec{0}$$

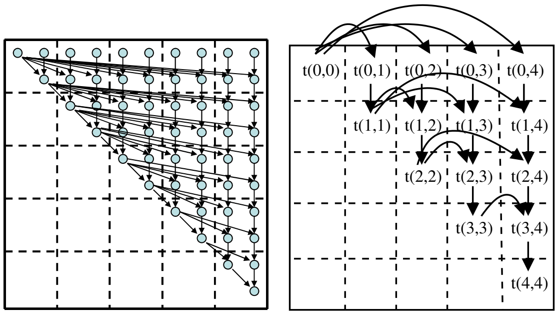
```
for (k=0; k<N; k++)
  for (j=k+1; j<N; j++)
   S1: a[k][j] = a[k][j]/a[k][k];

  for(i=k+1; i<N; i++)
    for (j=k+1; j<N; j++)
      S2: a[i][j] = a[i][j] − a[i][k]*a[k][j];
```

(a) Original LU code

```
for (c1=0; c1<=floord(N−2,32); c1++)
  for (c2=max(ceild(16*c1−15,16),0);
       c2<=floord(N−1,32); c2++)
    for (c3=max(ceild(16*c1−465,496),
          ceild(16*c1−15,16));
         c3<=floord(N−1,32); c3++)
      for (c4 =...)
        for (c5 =...)
          S1(c1,c2,c4,c5)
          for (c6 =...)
            S2(c1,c3,c2,c4,c6,c5)
```

(b) Tiled LU code

Figure 3. Example with LU decomposition



(a) Original dependences (b) Inter-tile dependences

Figure 4. Dependences in the tiled domain for LU code

Example. Figure 3(a) shows sequential code for LU decomposition and Figure 3(b) shows the corresponding tiled code. The tiles represent the computational tasks, with each task uniquely identified by its tile number. In the LU code in Figure 3(b), the pair of values for the outer tile variables (c1,c2) uniquely defines a task. The dependences in the tiled domain for the LU code are illustrated in Figure 4(a). The dependences across tiles after the projection of intra-tile dependences are shown in Figure 4(b).

3.1.2 DAG Code Generator

At the beginning of the run-time execution, the inter-tile dependences are captured and represented in the form of a task dependence DAG. The tiles are vertices of the DAG and the inter-tile dependences are edges between the corresponding vertices. We could associate a weight with each vertex based on the expected execution time of the tile and similarly, a weight with each edge based on the time to communicate data between the incident tasks. However, in our current implementation, we associate unit weights with the vertices and zero weights with the edges.

Recall that our goal is an approach to generate code at compile-time; this code, at run-time, generates the DAG of inter-tile dependences. Since the tiles represent the vertices, the following technique is used to generate code that creates the vertices in the DAG. The iteration space polytopes of all statements in the tiled domain, projected to contain only the supernode iterators, are provided to CLooG. CLooG scans the union of all polytopes and generates a loop nest that enumerates all tiles, and hence all vertices of the DAG. Figure 5(a) shows the compile-time-generated code that (at run-time) creates the vertices of the inter-tile dependence graph for the LU decomposition example. Note that this is a fully automatic and general approach: given any automatically (or manually) constructed tiling, as defined by a set of valid tiling hyperplanes, this technique can directly generate code that enumerates the tiles at run-time in order to construct the DAG vertices.

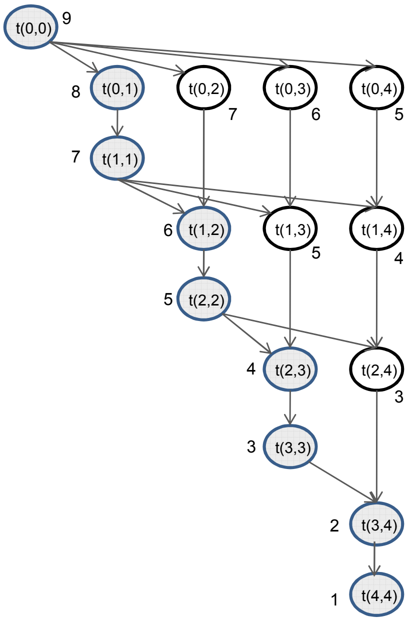


Figure 6. Inter-tile dependence DAG for LU decomposition

The following technique is used to generate code that creates the edges in the DAG. The inter-tile dependence extractor outputs a set of projected dependence polytopes (one corresponding to each dependence) that capture the inter-tile dependences. Each dependence polytope contains (1) supernode iterators of the tile containing the source statement instance, and (2) supernode iterators of the tile containing the target statement instance. Each dependence polytope is scanned using CLooG to create a loop nest with source tile iterators as the outer loops and target tile iterators as the inner loops. In this manner, all pairs of inter-dependent tiles are enumerated, and thus all edges of the DAG can be constructed. Figure 5(b) shows the code that generates edges of the inter-tile dependence graph corresponding to one of the dependences in the LU decomposition code. The actual DAG generated at run-time is shown in Figure 6. A key advantage of this approach is that it is automatic and general. It takes full advantage of all advances in polyhedral dependence analysis and code generation, while at the same time enables parallelizing compilers to go beyond the limitations of purely-compile-time affine scheduling.

```
for (c1=0; c1<=floord(N-2,32); c1++)
  for (c2=max(ceild(16*c1-15,16),0);
       c2<=floord(N-1,32); c2++)
    dag_add_vertex (c1, c2, 1.0);
```

(a) Creating all DAG vertices

```
for (s1=0;s1<=floord(N-3,32);s1++)
  for (s2=max(0,ceild(16*s1-15,16));
       s2<=min(floord(N-2,32),s1+1);s2++)
    for (t1=max(max(ceild(16*s1-15,16),
                    ceild (32*s2 -31,32)),0);
         t1<=min(min(floord(32*s2+31,32),s1+1),
                     floord (N-2,32));t1++)
      for (t2=max(max(max(0,ceild(32*s1-29,32)),
                  ceild (16*s2-15,16)), ceild (16*t1 -15,16));
           t2<=floord(N-1,32);t2++)
        dag_add_edge(s1, s2, t1, t2, 0.0);
```

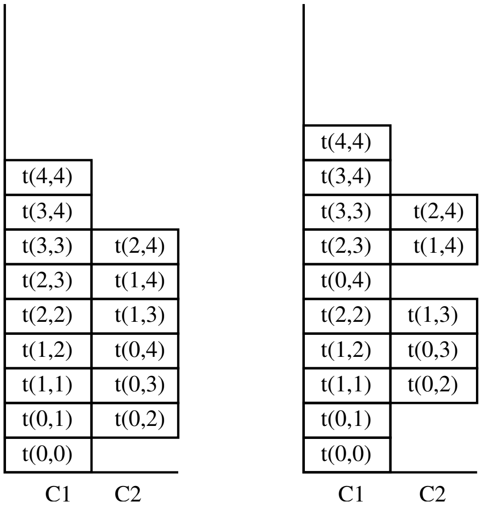
(b) Creating all DAG edges for one inter-tile dependence

Figure 5. Code for DAG generation

3.2 Task Scheduler

The task scheduler component adds code that, at run-time, analyzes the task dependence graph to assign priorities to the tasks and dynamically schedule them on cores/processors. The scheduling strategy used in our approach is as follows. Two metrics are associated with each vertex in the DAG (say G): *top level* and *bottom level*. The top level of a vertex v in G, denoted by $topL(v)$, is defined as the length of the longest path from the source vertex (i.e., the vertex with no predecessors) in G to v, excluding the vertex weight of v. The length of a path in G is the sum of the weights of the vertices and edges along that path. In our current implementation, since we have associated unit weights with the vertices and zero weight with the edges, the length of a path is the number of tasks that need to be executed along that path. The bottom level of a vertex v in G, denoted by $bottomL(v)$, is defined as the length of the longest path from v to the sink (vertex with no children), including the vertex weight of v. Any vertex v with maximum value of the sum of $topL(v)$ and $bottomL(v)$ belongs to a critical path in G.

The tasks are prioritized based on the sum of their top and bottom levels or just the bottom level, and a priority queue of ready-to-run tasks is maintained. A task is ready to run if all its predecessors have completed. Upon completion, each task sets a flag to denote its completion, computes amongst its children the set of tasks that are ready to run, and adds them to the priority queue. Tasks from the priority queue are executed in priority order on processors/cores as and when they become idle.



(a) Dynamic schedule (b) Affine schedule

Figure 7. Dynamic schedule vs. affine schedule (time steps)

Example. In the inter-tile dependence DAG for LU decomposition shown in Figure 6, the vertices are marked with $bottomL(v)$. The figure illustrates the dynamic scheduling strategy based on critical path analysis that prioritizes tasks based on $bottomL(v)$. The tasks are scheduled for execution based on the (1) completion of predecessor tasks, (2) $bottomL(v)$ priority, and (3) availability of processor core. Figure 7(a) shows the scheduling of the tasks represented in the DAG in Figure 6 for a dual-core system, using our dynamic scheduling approach. As evident from the figure, the ready-to-run tasks (tasks whose predecessor tasks are completed) with higher $bottomL(v)$ are given priority and the two cores are fully utilized as long as there are enough ready tasks to be scheduled. When two tasks with equal priority are ready for scheduling, the one at the top of the priority queue is chosen.

Figure 7(b) shows the affine polyhedral schedule produced by using the default approach in Pluto. The affine schedule for the case shown in Figure 7(b) is derived using a time schedule $\theta(i,j) = i + j$, where (i,j) is the 2-tuple denoting the task number. Tasks are executed in a strictly increasing order of $i + j$ values. Tasks with the same $i + j$ value are executed in parallel based on the availability of processor cores. Comparing the two schedules in Figure 7, it is easy to see the benefits of effective dynamic scheduling over compile-time affine scheduling. Furthermore, this result is not specific to Pluto or to this particular affine schedule — in general, such benefits will be observed when compared with any compile-time affine scheduling approach.

3.3 Run-time Execution

As explained earlier, at compile-time our approach generates code that has three segments: 1) the core computation or task code segment to be executed by a processor core, 2) the dependence DAG generation code segment, and 3) the task scheduling code segment. Algorithm 1 lists the steps that are performed at run-time while executing the code generated by our approach. The steps 5-10 are performed in parallel asynchronously by threads executing on different cores. The DAG generation code is executed first to create the DAG. Then $topL(v)$ and $bottomL(v)$ are calculated based on critical path analysis to prioritize the tasks/vertices. A priority queue is maintained to insert tasks based on priority and extract them for execution. Each parallel process waits for a task to be ready for execution and executes it by calling the task code. On completion of the task, all the tasks dependent on it have their wait-count decremented to indicate the completion of one of the parent tasks. A successor task is inserted into the priority queue if its wait-count is zeroed.

4. Experimental Results

This section assesses the effectiveness of the developed automatic dynamic scheduling approach using two linear algebra computa-

Algorithm 1 Run-time Execution

```
Execute DAG generation code to create a DAG $G$
Calculate $topL(v)$ and $bottomL(v)$ for each vertex $v \in G$, to prioritize the vertices
Create a Priority Queue $PQ$
$PQ.insert$ ( vertices with no parents in $G$)
while not all vertices in $G$ are processed do
    $taskid = PQ.extract(\,)$
    Execute $taskid$ // Compute code
    Remove all outgoing edges of $taskid$ from $G$
    $PQ.insert$ ( vertices with no parents in $G$)
end while
```

tions: LU and Cholesky decomposition. The sequential code for these computations is first transformed using the Pluto framework to generate locality-optimized tiled code (code that is tiled for data locality optimization at the levels of L1 cache and L2 cache), followed by processing through our task generator component to identify the computational tiles and inter-tile dependences, and generate code to create task dependence DAG, then followed by processing through our task scheduler component to add code that performs dynamic scheduling. The DAG is created at run-time and the computation is dynamically scheduled as described in Section 3.2. The reported performance for the dynamically scheduled versions of LU and Cholesky include all the overheads due to dynamic DAG generation as well as dynamic scheduling.

The experiments were conducted on two systems: a) a quad-core Intel Core 2 Quad Q6600 CPU clocked at 2.4 GHz (1066 MHz FSB) with a 32 KB L1 D cache, 8MB of L2 cache (4MB shared per core pair), and 2 GB of DDR2-667 RAM, running Linux kernel version 2.6.22 (x86-64), and b) a dual quad core Intel Xeon(R) E5345 CPU clocked at 2.33 GHz with each chip having a 8MB L2 cache (4MB shared per core pair) and 6 GB RAM, running Linux kernel version 2.6.18. The performance of the parallel code generated by our approach (which enables effective dynamic scheduling at run-time) was compared against that of the parallel code generated by Pluto (which is executed based on static affine polyhedral schedule). ICC 10.x was the primary compiler used to compile the Pluto generated code as well as the code generated by our approach; it was run with -fast -funroll-loops (-openmp for parallelized code); the -fast option turns on -O3, -ipo, -static, -no-prec-div on x86-64 processors; these options also enable auto-vectorization in icc.

An empirical study was carried out on the influence of L2 tile sizes on the statically scheduled (Pluto generated) LU code and our dynamically scheduled LU code, since L2 tiles are the ones that are scheduled for execution on different processor cores. We fixed the problem size as 8K and L1 tile size as $16 \times 300 \times 16$ (kji) and varied the L2 tile sizes. We found that dynamically scheduled parallel code always yielded better performance than statically scheduled parallel code. When the L2 tile sizes were very small ($16 \times 300 \times 16$), the performance of both statically and dynamically scheduled LU was poor, due to high synchronization overheads. The performance improved as the L2 tile sizes were increased, up till a point, after which (from tile sizes larger than or equal to $256 \times 600 \times 256$) the performance of both statically and dynamically scheduled LU saturated with increasing number of cores, due to contention for the shared L2 cache. We found $64 \times 300 \times 64$, $128 \times 300 \times 128$ and $256 \times 300 \times 256$ to be good L2 tile sizes for both statically and dynamically scheduled LU.

Figures 8 and 9 show the performance of LU in GFLOPS and the parallel speedup achieved on the two experimental systems for problem size N=8K. The L1 tile size was fixed as $16 \times 300 \times 16$ and the L2 tile size was fixed as $64 \times 300 \times 64$. As the L2 cache is shared between a core-pair and threads are typically scheduled first to cores that do not share the L2 cache, running an application on up to 2 cores in the quad core system and up to 4 cores in the dual quad core system, will not result in sharing of the L2 cache. We see that for these cases, the dynamically scheduled LU is able to achieve near perfect scaling. Thus dynamic scheduling is very effective in balancing the load on the cores. Even beyond 2 cores in the quad core system and 4 cores in the dual quad core system, dynamically scheduled LU is able to achieve significant performance improvement over statically scheduled LU.

We evaluated the usefulness of dynamic scheduling with another linear algebra computation: Cholesky decomposition. After an empirical evaluation of tile sizes, we fixed the L1 tile size as $8 \times 16 \times 8$ (kij) and the L2 tile size as $64 \times 64 \times 64$. We observed similar trends as for LU decomposition. Figure 10 shows the parallel speedup achieved for both statically and dynamically scheduled Cholesky (for a problem size of 8K) on the two experimental systems. We see that dynamic scheduling enables the parallel application to scale very well, achieving close to linear speedups.

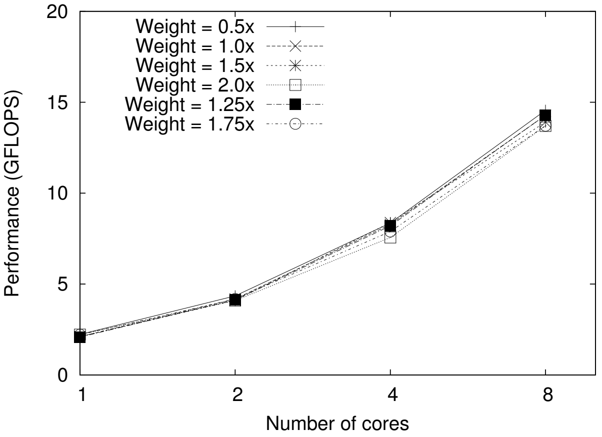
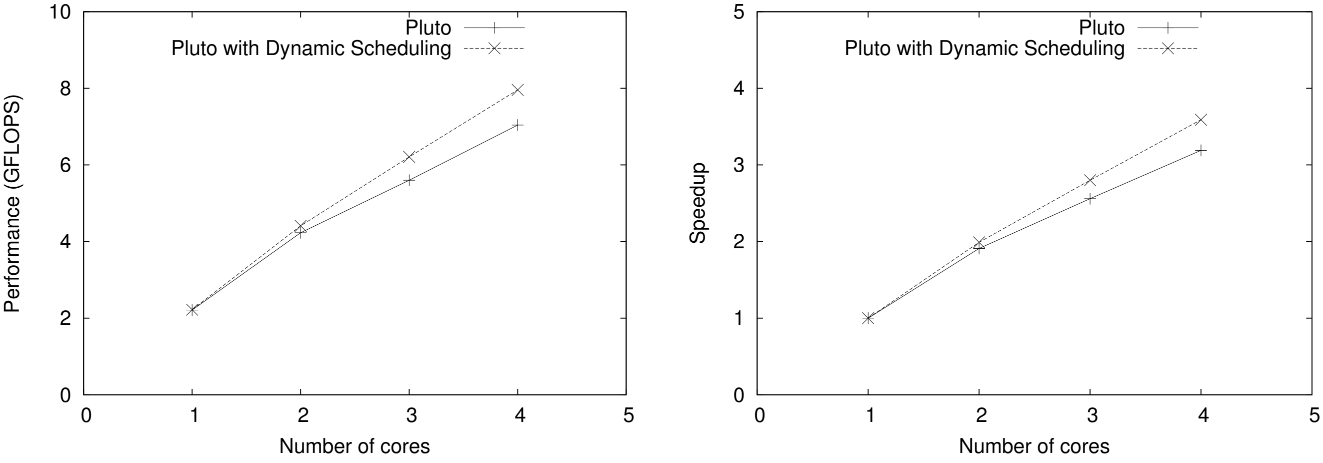


Figure 11. Performance of LU for various task weights

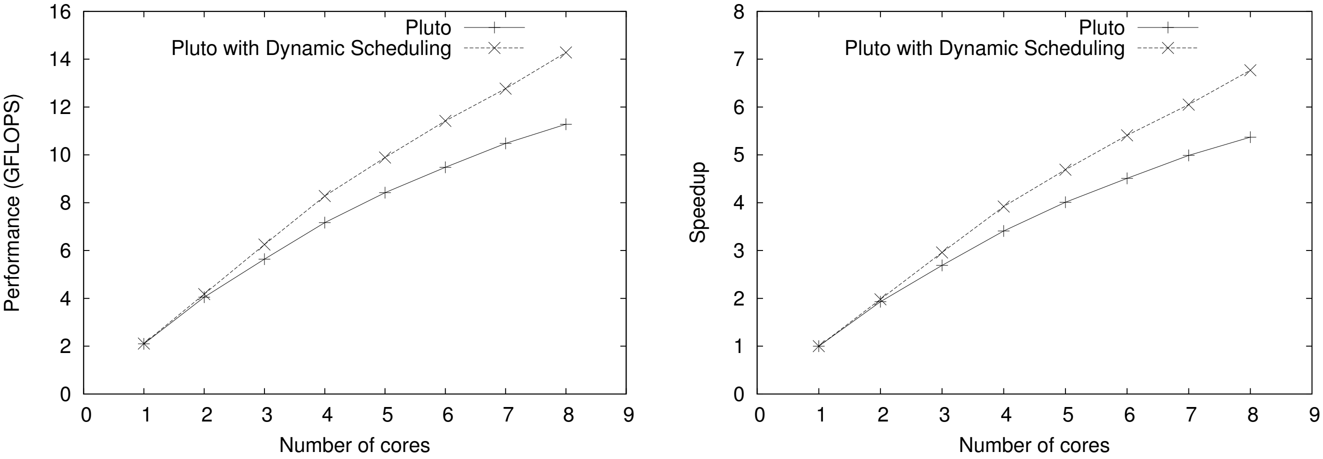
As mentioned earlier, the performance measurements of the dynamically scheduled versions of LU and Cholesky include the inter-tile dependence DAG generation overhead and the dynamic scheduling overhead (due to task priority calculation and priority queue maintenance). The overheads introduced by our approach are quite insignificant and do not affect performance. We measured separately the various overheads involved in our approach using the LU benchmark. The run-time DAG generation takes only around 0.001%, 0.003%, and 0.005% of the total execution time on 2, 4, and 8 processors, respectively. The task priority calculation and priority queue maintenance overhead account for only around 0.013%, 0.023%, and 0.036% of the total execution time on 2, 4, and 8 processors, respectively.

We also conducted experiments to assess the robustness of the dynamic scheduling strategy. Although we do not use empirically measured performance data to model task/vertex weights, assigning unit weights could still capture the priorities effectively. This is because the critical path analysis through inter-tile dependences could effectively capture the task priorities in spite of the less accurate estimate of task weight. Figure 11 shows the performance in GFLOPS for LU decomposition (for a problem size of 8K) for various higher/lower weights assigned to the tasks that perform more computation. As before, the L1 tile size was fixed as $16 \times 300 \times 16$ and the L2 tile size was fixed as $64 \times 300 \times 64$. The performance remains almost the same for various weights assigned to tasks per-



(a) Absolute performance (b) Parallel speedup

Figure 8. Performance of LU on 4 cores



(a) Absolute performance (b) Parallel speedup

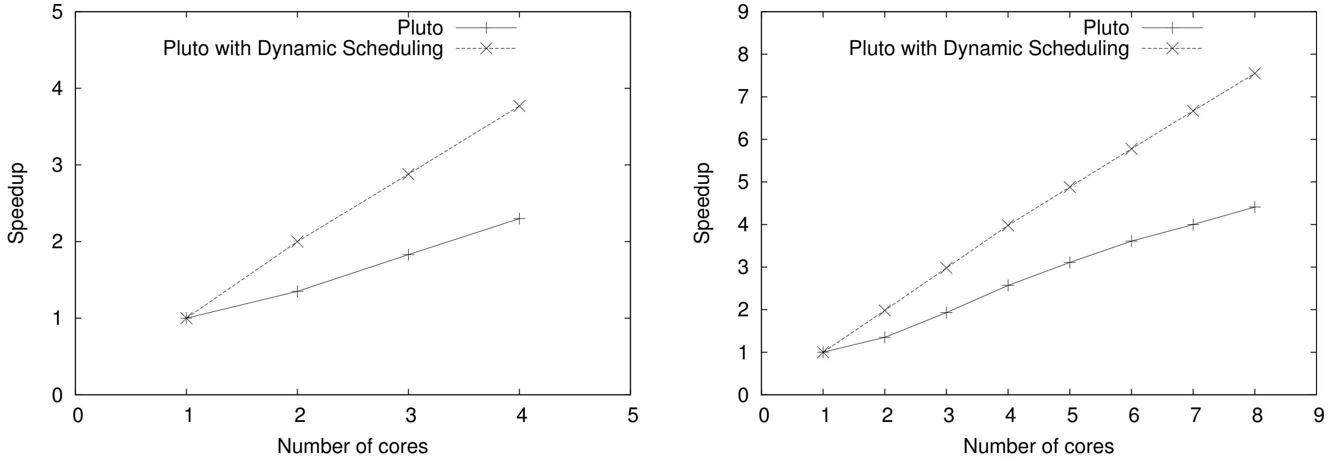
Figure 9. Performance of LU on 8 cores

forming more computation, clearly indicating that the scheduling strategy is robust enough even with unit weights assigned to tasks.

We conclude this section with a discussion on the absolute performance achieved relative to machine peak. Although the results presented above demonstrate excellent scalability, the absolute achieved GFLOPS performance is currently lower than the machine peak by over a factor of 2. The single-node performance of the generated tiled code is only about half of the machine peak because vectorization is sub-optimal. The Pluto system currently does not incorporate much sophistication in the approach to vectorization, relying primarily on the vectorization capability of the icc compiler. Work is in progress to implement a much more effective vectorization strategy using vector intrinsics. Another approach that we plan to pursue is that of using tuned kernels such as BLAS routines. The key idea is that of automatically recognizing when the tiled code generated by Pluto can be replaced by pre-optimized kernels. The dominant operation for Cholesky and LU decomposition is the multiply-add, and the core of the tiled code generated by Pluto is essentially a DGEMM. The use of DGEMM to replace the tiled code generated by Pluto requires the automatic separation and extraction of full rectangular tiles from the general polyhedral tiles and the identification of suitable pre-optimized kernels to substitute for the full tiles.

5. Related work

A number of works that use dependence abstractions weaker than those in polyhedral models have addressed loop parallelization [10, 2, 17, 16, 48]. In the context of loop parallelization in polyhedral models, several scheduling-driven works have developed techniques for finding minimum latency schedules or schedules with maximum fine-grained parallelism [20, 21, 17, 25]; these approaches are not aimed at coarse-grain parallelization or locality enhancement. Some works have used fine-grain schedules to determine loop structures which are then tiled to create coarse-grain tasks [20, 21, 25]. In contrast to these, partitioning-driven parallelization is addressed in works of Lim et al. [32, 31, 30] and our work on Pluto [34, 8, 7, 9, 6]. Note that due to synchronization/communication costs on most modern parallel architectures, at



(a) on 4 cores

(b) on 8 cores

Figure 10. Performance of Cholesky

least one level of coarse-grained parallelism is desirable, in addition to enhanced locality. The Pluto approach is the first to explicitly model tiling in a polyhedral transformation framework, which allows us to address two key issues: (i) effective extraction of coarse-grained parallelism, and (ii) data locality optimization.

Nevertheless, depending on the structure of the loops and their parallelization, the tiled output code from any of the above approaches may still suffer from load-imbalance; therefore, dynamic scheduling of tiles is key to improving performance. The approach we pursue here has some similarities to the inspector/executor approach used in *runtime compilation* [42, 43, 35] in that an analyzer is created at compile-time for execution at run-time to facilitate optimized execution. However, a fundamental difference is that run-time compilation approaches typically use inspectors to obtain essential information (e.g., dependence information) that can only be known at run-time. In contrast, in our context, all dependence information is completely known at compile-time for the affine computations that we address. The problem is that the affine schedule generated by the Pluto framework (or any other existing automatic parallelization framework) is overly constraining due to the use of a static parallel loop structure with implicit barrier synchronization. The same problem exists with the parallel implementations in LAPACK routines, as highlighted by the recent research from the University of Tennessee [18, 12, 11]. The solution approach we pursue has been inspired by that work, with the main difference being that we seek to generate the dynamically self-scheduling code completely automatically by compiler transformations from sequential code for the computation.

Several efforts have targeted dynamic run-time parallelization [13, 28, 39, 41] as well as speculative parallelization [14, 38, 40]. The basic difference between these approaches and our work is that we use dynamic scheduling to improve performance of loop computations that are amenable to compile-time characterization of dependences.

A plethora of work has been published on the topic of DAG scheduling [1, 45, 44, 23, 27, 26]. Although more sophisticated DAG scheduling algorithms could have been used in our work, we found that a straightforward bottom-level based critical-path dynamic DAG scheduling algorithm was very effective. The focus of our work has not been on exploring alternative scheduling algorithms, but on developing an approach to automatic compile-time generation of DAG generation code to be executed at run-time to facilitate dynamic load balancing of tiled parallel code.

6. Conclusions

The parallel code generated by automatic parallelization approaches for multi-statement input programs with statements of different dimensionalities suffers from excessive synchronization in the form of barriers, leading to poor scalability on multi-core systems due to load imbalance. In this paper, we have developed a fully-automatic parallelization approach that can transform input sequential codes with affine dependences for asynchronous, load-balanced parallel execution. We have described an approach that generates, at compile-time, additional program code whose role at run-time is to dynamically extract inter-tile data dependences, and dynamically schedule the parallel tiles on the processor cores to improve load balance for effective parallel execution on multi-core systems. The effectiveness of the approach has been demonstrated through two linear algebra computations: LU and Cholesky decomposition.

Acknowledgments This work is supported in part by the U.S. National Science Foundation through awards 0403342, 0508245, 0509442, 0509467, 0541409, 0811457 and 0811781. We would like to thank Cedric Bastoul from Paris-Sud XI University, Orsay, France, for CLooG. We would also like to thank Louis-Noel Pouchet from INRIA, France, for providing us with the code for Fourier Motzkin elimination.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974.

[2] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, 1987.

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

[4] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'04*, pages 7–16, 2004.

[5] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 23–30, 2003.

[6] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformations for

communication minimal parallelization and locality optimization of arbitrarily nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, Ohio State University, May 2007.

[7] 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.

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, 2008.

[9] U. Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, Oct. 2007.

[10] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3–4):421–444, 1998.

[11] A. Buttari, J. Dongarra, P. Husbands, J. Kurzak, and K. Yelick. Multithreading for synchronization tolerance in matrix factorization. In *Proceedings of the SciDAC 2007 Conference*. Journal of Physics: Conference Series, 2007.

[12] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600, Innovative Computing Laboratory, University of Tennessee Knoxville, September 2007. Submitted to *Parallel Computing*. LAPACK Working Note 191.

[13] D.-K. Chen, J. Torrellas, and P.-C. Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 518–527, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[14] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 13–24, New York, NY, USA, 2003. ACM.

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

[16] A. Darte, G.-A. Silber, and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[17] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *IJPP*, 25(6):447–496, Dec. 1997.

[18] J. Dongarra. Four important concepts that will effect math software. In *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA'08)*, 2008.

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

[20] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I: one-dimensional time. *IJPP*, 21(5):313–348, 1992.

[21] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *IJPP*, 21(6):389–420, 1992.

[22] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.

[23] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):686–701, 1993.

[24] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations. *IJPP*, 34(3):261–317, June 2006.

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

[26] H. Kasahara, H. Honda, and S. Narita. Parallel processing of near fine grain tasks using static scheduling on oscar (optimally scheduled advanced multiprocessor). In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 856–864, Washington, DC, USA, 1990. IEEE Computer Society.

[27] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[28] S.-T. Leung and J. Zahorjan. Improving the performance of runtime parallelization. *SIGPLAN Not.*, 28(7):83–91, 1993.

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

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

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

[32] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998.

[33] Parallel linear algebra for scalable multi-core architectures (PLASMA) project. http://icl.cs.utk.edu/plasma.

[34] PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[35] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 361–370, New York, NY, USA, 1993. ACM.

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

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

[38] C. G. Quinones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *PLDI 05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, 2005.

[39] L. Rauchwerger and D. Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. *SIGPLAN Not.*, 30(6):218–232, 1995.

[40] P. Rundberg and P. S. Om. Low-cost thread-level data dependence speculation on multiprocessors. In *In Fourth Workshop on Multi-threaded Execution, Architecture and Compilation*, pages 1–9, 2000.

[41] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 263–273, New York, NY, USA, 2007. ACM.

[42] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distrib. Comput.*, 8(4):303–312, 1990.

[43] J. H. Salz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5):603–612, 1991.

[44] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[45] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 17–26, New York, NY, USA, 1986. ACM.

[46] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *International Conference on Compiler Construction (ETAPS CC'06)*, pages 185–201, Mar. 2006.

[47] N. Vasilache, C. Bastoul, S. Girbal, and A. Cohen. Violated dependence analysis. In *ACM ICS*, June 2006.

[48] M. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.