Transforming Loops to Recursion for Multi-Level Memory Hierarchies *

Qing Yi
Rice University
Houston, TX 77005.
qingyi@cs.rice.edu

Vikram Adve
University of Illinois at Urbana-Champaign
Urbana, IL 61801.
vadve@cs.uiuc.edu

Ken Kennedy
Rice University
Houston, TX 77005.
ken@cs.rice.edu

Abstract

Recently, there have been several experimental and theoretical results showing significant performance benefits of recursive algorithms on both multi-level memory hierarchies and on shared-memory systems. In particular, such algorithms have the data reuse characteristics of a blocked algorithm that is simultaneously blocked at many different levels. Most existing applications, however, are written using ordinary loops. We present a new compiler transformation that can be used to convert loop nests into recursive form automatically. We show that the algorithm is fast and effective, handling loop nests with arbitrary nesting and control flow. The transformation achieves substantial performance improvements for several linear algebra codes even on a current system with a two level cache hierarchy. As a side-effect of this work, we also develop an improved algorithm for transitive dependence analysis (a powerful technique used in the recursion transformation and other loop transformations) that is much faster than the best previously known algorithm in practice.

1 Introduction

Emerging processor architectures rely on progressively deeper memory hierarchies to achieve high performance. For example, systems designed for the forthcoming Itanium processor are expected to use 3 levels of cache, two on chip and one off-chip [23]. Systems based on the IBM Power4 processor are also expected to use three levels of cache. Furthermore, in shared-memory multiprocessor systems, the memory shared between different processors effectively adds one or more additional levels of memory hierarchy that must be accounted for to achieve high performance. Managing performance on deep memory hierarchies is widely considered to be one of the most important open problems in achieving high performance on current and future systems.

In this context, recursive (e.g., divide-and-conquer) algorithms appear to have some potentially valuable locality properties. For example, consider the obvious recursive formulation of matrix multiplication shown in Figure 1. In a single recursive step, this multiplication can be broken down into eight smaller matrix multiplications (the four matrix addition operations can be done as part of the subproblems by accumulating the subproblem results directly into the C matrix). Each of the multiplication subproblems has essentially the same reuse properties as the original problem, but with a working set that is four times smaller, assuming equal-size submatrices. Effectively, the original problem has been blocked for each of the three loops. A second recursive division would then provide a second level of blocking for each loop, and so on. The recursive algorithm therefore has the data reuse characteristics of a blocked algorithm *that is simultaneously blocked at many different levels*, in effect providing a hierarchy of working sets. Many other divide-and-conquer algorithms exhibit a similar property.

Recently, other researchers have obtained both experimental and theoretical results that bear out these observations, showing significant performance benefits of recursive algorithms on both uniprocessor cache hierarchies and on shared-memory systems. In particular, Gustavson and Elmroth [13, 9] have demonstrated significant performance benefits from recursive versions of Cholesky and QR factorization, and Gaussian elimination with pivoting. For example, a single recursive version of Cholesky replaces both the level-2 and level-3 LAPACK versions and outperforms both on an IBM RS-6000 workstation (by as much as a factor of 3 compared with the level-2 version) [13]. Leiserson's group has observed significant improvements in memory locality by using recursive algorithms on uniprocessor cache hierarchies, on page-based software distributed shared memory systems, and for preserving locality while doing dynamic load-balancing in shared memory systems [21, 10]. Some algorithms they studied include FFT, matrix transpose, matrix multiplication, and sorting. Furthermore, they have shown that "cache-oblivious" divide-and-conquer algorithms provide asymptotically optimal performance on multi-level cache hierarchies (optimal in terms of moving data between different levels of cache) [10]. All these results argue that recursive computational structures have the potential to address one of the critical performance challenges in current and future systems, at least for the above types of codes.

Most existing applications, however, are not expressed in divide-and-conquer form, but use ordinary loops for expressing iteration. Manually converting such codes to a recursive form (even without any change in the underlying algorithm) would be a laborious and error-prone process. Furthermore, different recursive forms would usually be needed for uniprocessor and parallel architectures. If compiler techniques could be used to perform the conversion automatically (or with guidance from the programmer), they would greatly

*This research was sponsored by the Department of Energy's ASCI Academic Alliances program under research contract B347884, and supported in part by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159 and by NSF Research Instrumentation Award CDA-9617383.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI 2000, Vancouver, British Columbia, Canada.
Copyright 2000 ACM 1-58113-199-2/00/2000006 $5.00.

$$C = A \times B$$

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Figure 1: Recursive formulation of matrix multiplication.

simplify the programmer's task.

As the principal contribution of this paper, we present a compiler transformation that can be used to convert ordinary loop nests containing into recursive form automatically. The transformation has several potential applications, although this paper focuses on using it to improve uniprocessor cache performance. The transformation primarily relies on ordinary dependence analysis. It can be applied to arbitrary loop nests including imperfect loop nests, multiple consecutive loop nests, and loops containing conditionals. (It is also fast as discussed further below.) We have implemented this transformation in the Rice dHPF compiler infrastructure. The algorithm has successfully transformed loop nests containing unexploited reuse into recursive form in several codes including matrix multiplication, Cholesky and LU factorization without pivoting, and Erlebacher, an implicit finite-differencing scheme for computing partial derivatives. (Transforming the pivoting versions of LU and Cholesky would require an additional analysis step, as discussed in Section 4.)

A key step in our algorithm is based on a loop transformation technique called iteration space slicing, recently described by Pugh and Rosser [27, 28]. Iteration space slicing uses *transitive dependence analysis* on the dependence graph to compute the instances of a particular statement that must precede or follow a given set of instances of another statement. This is a powerful technique that we believe could have wide applicability in optimizing compilers in the future. Pugh and Rosser's algorithm is quite expensive, however, because it uses a very general dependence representation, and because it precomputes all transitive dependences in time $O(N^3)$ for a graph with N vertices.

A second contribution of this paper is an improved algorithm for transitive dependence analysis that is much more efficient in practice than the one used by Pugh and Rosser. In particular, our algorithm makes two major improvements. First, it uses an efficient and unorthodox matrix representation of dependence directions that greatly speeds up operations such as concatenation and union on pairs of dependences (defined in Section 3). Our representation is less detailed but it does not appear to lose significant information for common cases in practice. Second, our algorithm computes transitive dependence information to a single destination node on demand, and for many codes it is able to compute all transitive dependences incident on a particular statement in time that is close to linear in the size of the dependence graph. (although it can still require time that is $O(N^3)$ for worst case graphs, as described in Section 3). Together, these improvements allow us to transform benchmarks with a few thousand lines of code in a few minutes, and (we believe) make iteration space slicing (as well as the recursion transformation) practical even for large codes.

Finally, we evaluate the benefits of the recursion transformation using measurements of several matrix codes (mentioned above) on a uniprocessor SGI workstation with a two-level cache hierarchy. Compared with the original unblocked versions of the same codes, the generated recursive codes are faster by factors of 3x, 4x and 5x in LU, Cholesky and matrix-multiply. The recursive codes perform comparably with one and two-level blocked versions of matrix-multiply, and outperform a one-level blocked version of LU factorization. We observe, however, that the recursive versions suffer from similar problems with conflict misses as does blocking, and require similar strategies (e..g, buffer copying) to reduce such misses [17, 8, 12].

The next section describes our algorithm for the recursion transformation, assuming transitive dependence information exists. Section 3 describes our improved algorithm for transitive dependence analysis. The subsequent sections present our experimental results, compare our results with related work, and then conclude with a brief summary and description of future work.

2 Recursion Transformation

Given a code segment consisting of one or more consecutive loop nests, the recursion transformation creates one or more recursive procedures (for groups of related statements) and transforms the original code to include initial calls to these procedures. Although memory hierarchy optimization is the primary focus of this paper, the transformation can be used for different purposes by replacing four specific decision steps listed in section 2.2. The core analysis and code generation steps, however, need not change. Below, we first present the recursion transformation in a general framework, and then describe how we perform the four decision steps for the specific goal of improving cache performance of sequential codes.

The general strategy of the algorithm can be introduced using the loop nest from LU shown in Figure 3. The final generated code is shown in Figure 4. The algorithm first picks statement s_2 in LU as a "key" statement that will be used to drive the transformation. The j and i loops surrounding s_2 are chosen to be enumerated recursively, so that there are four recursive calls. The recursive procedure is parameterized by formal parameters (lb_j, ub_j, lb_i, ub_i) representing the bounds of these recursive loops, i.e., representing the iterations $\{(i,j) \mid lb_j \le j \le ub_j \wedge lb_i \le i \le ub_i\}$. The goal is to execute these iterations of s_2 with a single representative call to the procedure (called the "current" recursive call). The key analysis step in the algorithm is to determine which iterations of every statement (including s_2) in the original code must be executed within the same call to the recursive procedure, in order to preserve dependences. This step directly uses the results of transitive dependence analysis. The resulting symbolic iteration sets are used directly to generate the code for the recursion base case, as shown in Figure 4. Finally, additional code is synthesized for the recursive calls, and an IF statement is synthesized to decide whether the base case has been reached. This loop nest will be used as a running example to illustrate the details of the algorithm.

Before describing the algorithm, we first introduce some notation and terminology.

2.1 Notation and Definitions

Definition 1. A *recursive loop* is a loop whose iteration space will be enumerated recursively. A generated recursive procedure with N recursive loops will be parameterized by the formal parameters $\vec{R}_{formal} = (lb_1, ub_1, \ldots, lb_N, ub_N)$.

Definition 2. Let $\vec{r}_{actual}$ denote an arbitrary vector of actual values (symbolic expressions) for the parameters $\vec{R}_{formal}$. In the algorithm below, a single such $\vec{r}_{actual}$ may denote a range of iterations executed over several recursive calls.

Definition 3. For each statement s that has been included in a recursive procedure, we define three symbolic iteration sets:

$Current(s)$ = iteration set of s that must be executed in the "current" recursive call.

$Previous(s)/Future(s)$ = the iteration set of s that must be executed before/after the current recursive call respectively.

All these iteration sets are implicitly parameterized by the formal parameters of the recursive procedure, $\vec{R}_{formal}$. For some actual (symbolic) parameter values $\vec{r}_{actual}$, we can compute specific instances of these sets, which we denote as $Current(s)[\vec{r}_{actual}]$.

Definition 4. For each pair of statements (s_1, s_2), we define two functions:

Before$(s_1, s_2)[I]$ = the iteration set of s_1 that must be executed before the iteration set I of s_2.

After$(s_1, s_2)[I]$ = the iteration set of s_1 that must be executed after the iteration set I of s_2.

$After[]$ is the inverse function of $Before[]$. These two functions capture the transitive dependence between statements s_1 and s_2, and are computed on demand by the transitive dependence analysis algorithm described in section 3.

All the iteration sets or functions described in this section are represented as symbolic integer sets or mappings using the Omega library [14].

2.2 Overview of Algorithm

The recursion transformation algorithm is shown in Figure 2. To simplify the description of the algorithm, we initially ignore IF statements and loops with non-unit strides. Section 2.3.3 describes simple extensions to the algorithm to handle these issues.

The major steps of the recursion transformation are as follows. This description closely follows the structure of the top-level procedure *Recursion-Transformation(C, D)*.

(1) Choose a set of statements called "key" statements to drive the algorithm ($KeyStmts$ in Figure 2). Multiple key statements can be included in a single recursive procedure, allowing us to capture more reuse. Furthermore, separate groups of independent statements can be put into separate recursive procedures generated from different key statements. For each key statement $skey \in KeyStmts$ that has not yet been included in any of the already generated recursive procedures, try to create a recursive procedure for it through the following steps.

(2) Choose a subset of the loops surrounding $skey$ (*Rloops* in Figure 2) to enumerate recursively. These will be the recursive loops with parameters lb_k and ub_k as in Definition 1 above.

(3) Decide the order of making recursive calls (*Rorder* in Figure 2). Recursive calls at all levels are made according to this order. For example, in the case of LU in Figure 3, loops j and i are chosen as the recursive loops, and the recursive order specifies simply that i is divided before j and the two halves of each loop are executed in forward order.

(4) Compute a set $Rstmts$ that holds all the statements that should be included in the recursive procedure. For each statement $s \in Rstmts$, compute $Current(s)$, i.e., the symbolic iteration set of s that must be executed in a single recursive call. Both of these are done by function *Compute-Iter-Sets* which uses information from transitive dependence analysis, i.e., the $Before$ and $After$ mappings defined above.

(5) The algorithm never fails to compute a legal iteration set $Current(s)$ for any statement. The way failure occurs, however, is that the iteration sets may not be sufficiently reduced (e.g., in the worst case, all the iterations will simply be executed in a single recursive call). Therefore, we examine $Current(s) \forall s \in Rstmts$ to determine whether the transformation will be profitable (this is done within function *Backward-Slicing*). If the transformation is unprofitable, an empty set of statements, $Rstmts$, is returned.

(6) If $Rstmts \neq \emptyset$, we have a legal, profitable transformation. *Create-Recur-Proc*$(Rstmts, \vec{R}_{formal}, Rorder)$ creates a recursive procedure with parameters $\vec{R}_{formal}$ that recursively executes $Current(s)$, $\forall s \in Rstmts$. *Transform-Original-Code*$(C, Rstmts, Rloops)$ transforms the original code so that all statement instances included in the recursive procedure are replaced by an initial call to the procedure.

The above process is repeated for all remaining key statements. All statement instances included in a previously created recursive procedure are excluded (by subtracting them from the "original" iteration sets) when processing the next key statement. This ensures that the above algorithm never duplicates any statement instances.[1]

The core of the algorithm is in steps (4) and (6), and these are described in more detail in Sections 2.3 and 2.4. The remaining four steps depend on the particular goal of the transformation and the specifics of these steps for improving cache performance are described in Section 2.5.

2.3 Computing Iteration Sets

In Figure 2, the function *Compute-Iter-Sets* computes $Current(s)$, $Previous(s)$ and $Future(s)$ for each statement s that should be included in the recursive procedure for a particular key statement $skey$. The function uses a technique called "iteration space slicing" [28] to compute these iteration sets. This technique is analogous to "program slicing" [26], except that it operates on iteration sets (i.e., instances) of statements rather than entire statements. For a given set of iterations, I_0, of a statement S_0, iteration space slicing computes the specific set of iterations of each statement s that must execute before (backward iteration space slicing) or after (forward iteration space slicing) the given iterations of S_0. These can be computed simply by applying the $Before$ and $After$ functions defined earlier, which are obtained from transitive dependence analysis.

For the recursion transformation, $skey$ serves as S_0 and the iteration set defined by the parameters $\vec{R}_{formal}$ is I_0. The three iteration sets of $skey$ are first initialized as follows. *Restrict-Bounds*$(skey, \vec{R}_{formal})$ initializes $Current(skey)$ by

[1] This does not apply to IF statements and DO loop headers. Side-effects in these statements are handled as described in Section 2.3.3.

```
      do k = 1, N - 1
         do i = k + 1, N
s1:         A(i, k) = A(i,k) / A(k,k)
J:       do j = k+1, N
I:          do i = k + 1, N
s2:            A(i, j) = A(i,j) - A(i,k) · A(k,j)
```

Initial decisions:

$skey = s_2$
$Rloops = \{J, I\}$
$\vec{R}_{formal} = (lb_J, ub_J, lb_I, ub_I)$
$Rorder = (lb_J, m_J, lb_I, m_I) \rightarrow (lb_J, m_J, m_I+1, ub_I) \rightarrow (m_J+1, ub_J, lb_I, m_I) \rightarrow (m_J+1, ub_J, m_I+1, ub_I)$

Compute-Iter-Sets:

$Fpars = \{(ub_J+1, N, 2, N), (lb_J, ub_J, ub_I+1, N)\}$
$Ppars = \{(2, lb_J-1, 2, N), (lb_J, ub_J, 2, lb_I-1)\}$
$Before(s_1, s_2) = \{(k,j,i) \rightarrow (k', i') \mid k' \leq k, i' \leq i\}$
$Before(s_2, s_2) = \{(k,j,i) \rightarrow (k', j', i') \mid k' \leq k, j' \leq j, i' \leq i\}$

$Current(s_2) = \{(k,j,i) \mid 1 \leq k \leq min(ub_J, ub_I) - 1,\ max(k+1, lb_J) \leq j \leq ub_J,\ max(k+1, lb_I) \leq i \leq ub_I\}$
$Previous(s_2) = \{(k,j,i) \mid 1 \leq k \leq lb_J - 2,\ k+1 \leq j \leq lb_j - 1,\ k+1 \leq I \leq N\ \ or\ \ 1 \leq k \leq min(ub_J - 1, lb_I - 2),\ max(k+1, lb_J) \leq j \leq ub_J,\ k+1 \leq i \leq lb_I - 1\}$

Backward-Slicing:

$Previous(s_2) = Before(s_2, s_2)[Previous(s_2)]$
$Before(s_2, s_2)[Current(s_2)] = \{(k,j,i) \mid 1 \leq k \leq min(ub_J, ub_I) - 1,\ k+1 \leq j \leq ub_J,\ k+1 \leq i \leq ub_I\}$
$Before(s_1, s_2)[Current(s_2)] = \{(k,i) \mid 1 \leq k \leq min(ub_J, ub_I) - 1,\ k+1 \leq i \leq ub_I\}$
$Previous(s_1) = Before(s_1, s_2)[Previous(s_2)]$
$= \{\ (k,i) \mid 1 \leq k \leq lb_J - 2,\ k+1 \leq i \leq N\ \ or\ \ 1 \leq k \leq min(ub_J - 1, lb_I - 2),\ k+1 \leq i \leq lb_I - 1\ \}$
$Current(s_1) = Before(s_1, s_2)[Current(s_2)] - Previous(s_1)$
$= \{(k,i) \mid max(lb_J, lb_I) - 1 \leq k \leq min(ub_j, ub_I) - 1,\ max(k+1, lb_I) \leq I \leq ub_I\}$

Figure 3: Computed Iteration Sets of LU

first constructing the original iteration set of $skey$, then restricting the iteration range of each recursive loop l_k to be within the lower and upper recursive bound parameters lb_k and ub_k. To initialize $Previous(skey)$ and $Future(skey)$, we first compute the iterations of the recursive loops that precede or follow the iterations $\vec{R}_{formal}$ in the recursive call order ($Ppars$ and $Fpars$). We then initialize $Previous(skey)$ by substituting each $\vec{r}_{actual} \in Ppars$ for $\vec{R}_{formal}$ in $Current(skey)$ and taking the union of the resulting sets ($Future(skey)$ is initialized similarly).

For example, in Figure 3, $Current(s_2)$ is first initialized by restricting the original iteration set of s_2 with the parameters in $\vec{R}_{formal}$. $Previous(s_2)$ can then be computed by first replacing lb_J, ub_J, lb_I, ub_I in $Current(s_2)$ with $(2, lb_J - 1, 2, N)$ and $(lb_J, ub_J, 2, lb_I - 1)$ respectively, then taking the union of the two iteration sets obtained.

Rstmts is initialized with the empty set instead of containing *skey*. This will cause the three iteration sets of *skey* to be recomputed and examined for profitability in *Backward-Slicing*. This is necessary to guarantee the correctness and profitability of the chosen key statement.

In Figure 2, *Backward-Slicing* is called first within *Compute-Iter-Sets*. *Backward-Slicing* and *Forward-Slicing* then call each other repeatedly until no more statements should be included in *Rtmts*. These two functions are described next.

2.3.1 Backward Recursion Slicing

This step is necessary to guarantee the correctness of the transformation. It ensures that any computation that must execute before a particular iteration I of statement *skey* is either performed in the same recursive call as $I(skey)$ or in previous recursive calls. Later on, the code generation step ensures that the original order among $Current(s)\ \forall s \in Rstmts$ is preserved within each recursive call. Taken together, these ensure that no semantics of the original code will be violated.

In Figure 2, *Reachable-Into*$(D, skey)$ returns all the statements that have some dependence paths into *skey* in the dependence graph. Subtracting *Rstmts* gives *AddStmts*, the additional statements that need to be included in the recursive procedure.

For an arbitrary statement $s \in AddStmts$, $Before(s, skey)\ [Previous(skey)]$ gives the iteration set of s that must execute before the iteration set of *skey* in previous recursive calls. This is exactly the iteration set of s that must execute before the current recursive call (i.e., $Previous(s)$). Similarly, $Before(s, skey)[Current(skey)]$ gives the iteration set of s that must execute before *or during* the current recursive call. The difference of these two sets gives the iteration set of s that must execute during the current recursive call ($Current(s)$). Once $Current(s)$ is computed, $Future(s)$ can be computed as $\cup_{\vec{r}_{actual} \in Fpars} Current(s)[\vec{r}_{actual}]$. (This is used later in *Forward-Slicing*.)

Consider the code of LU in Figure 3. Initially $Rstmts = \emptyset$, $AddStmts =$ *Reachable-Into*$(D, s_2) = \{s_1, s_2\}$. Recomputing $Previous(s_2) = Before(s_2, s_2)\ [Previous(s_2)]$ and $Current(s_2) = Before(s_2, s_2)[Current(s_2)] - Previous(s_2)$ produces no change. $Previous(s_1)$ and $Current(s_1)$ are also computed using expression $Before(s_1, s_2)\ [Previous(s_2)]$ and $Before(s_1, s_2)\ [Current(s_2)] - Previous(s_1)$ respectively.

At this point, the computed iteration sets are verified for profitability. If $Profitable\ (Current(s))$ returns $false$ for any statement s, none of the statements in *AddStmts* will be added into *Rstmts*. If all the verifications succeed, we union *Rstmts* with *AddStmts*, and apply forward slicing from *AddStmts* so that other statements that use the values computed by *AddStmts* can also be included.

2.3.2 Forward Recursion Slicing

Forward slicing is not necessary for correctness, and is therefore an optional pass. The goal of forward slicing is to keep together iterations of statements that *use* values computed by the iterations $Current(s)$ of all statements already included in the recursive procedure. If this step is used, however, it must again invoke backward slicing for each additional statement that is included in *Rstmts* to guarantee that any previous values required for these additional statements will be computed first. If backward slicing fails for any of the additional statements, we simply don't include it in *Rstmts*. This process of forward and backward slicing could be repeated as long as it terminates with a pass of backward slicing.

The function *Forward-Slicing* is similar to *Backward-Slicing*. It first uses *Reachable-From*$(D, AddStmts)$ to identify all the statements having some incoming dependence paths from statements in *AddStmts*, then uses $After$ functions instead of $Before$ functions to compute the $Future$ and $Current$ iteration sets. *Forward-Slicing* starts not just from one statement *skey* but from the set of statements, *AddStmts*, when computing the $Future$ and $Current$ iteration sets for

```
Recursion-Transformation(C, D)
      C:original code; D: dependence graph
  KeyStmts = Choose-Key-Statements (C, D)
  while KeyStmts ≠ ∅ do
    extract next statement skey from KeyStmts
    if (skey has been already processed) then continue
    // recursively computing key statement skey
    Rloops = Choose-Recur-Loops (C, D, skey)
    R⃗_formal = Create-Recur-Params (Rloops)
    Rorder = Decide-Recur-Order (C, D, Rloops)
    Rstmts =Compute-Iter-Sets(D, skey, R⃗_formal, Rorder)
    // code transformation
    if (Rstmts ≠ ∅) then
      Create-Recur-Proc (Rstmts, R⃗_formal, Rorder)
      Transform-Orig-Code (C, Rstmts, Rloops)

Compute-Iter-Sets (D, skey, R⃗_formal, Rorder)
      D: dependence graph; skey: key statement;
      R⃗_formal: recursive params; Rorder: recur call order;
      return: stmts to be included in recursive procedure
  Ppars = Previous-Recur-Calls (Rorder, R⃗_formal)
  Fpars = Future-Recur-Calls (Rorder, R⃗_formal)
  Current(skey) = Restrict-Bounds (skey, R⃗_formal)
  Previous(skey) = ∪_{r⃗_actual ∈ Ppars} Current(skey)[r⃗_actual]
  Future(skey) = ∪_{r⃗_actual ∈ Fpars} Current(skey)[r⃗_actual]
  Rstmts = ∅
  Backward-Slicing (D, skey, Rstmts, Ppars, Fpars)
  return Rstmts

Backward-Slicing(D, skey, Rstmts, Ppars, Fpars)
      D: dependence graph; skey: key statement;
      Rstmts: statements already processed;
      Ppars, Fpars: params for previous/future recur-calls
  AddStmts = Reachable-into(D, skey) − Rstmts
  if (AddStmts == ∅) then return
  for each statement s ∈ AddStmts do
    Previous(s) = Before(s, skey)[Previous(skey)]
    Current(s) = Before(s, skey)[Current(skey)]
    Current(s) = Current(s) − Previous(s)
    Future(s) = ∪_{r⃗_actual ∈ Fpars} Current(s)[r⃗_actual]
    if (! Profitable[Current(s)]) then return ∅
  Rstmts = Rstmts ∪ AddStmts
  Forward-Slicing (D, AddStmts, Rstmts, Ppars, Fpars)

Forward-Slicing (D, AddStmts, Rstmts, Ppars, Fpars)
      AddStmts: stmts to start forward slicing;
      Rstmts: statements already processed;
      Ppars, Fpars: params of previous/future recur-calls
  ExtraStmts =Reachable-From(D, AddStmts) − Rstmts
  for each statement s ∈ ExtraStmts do
    if (s ∈ Rstmts) then continue
    Future(s) = Current(s) = ∅
    for each statement ss ∈ StartStmts do
      Future(s)∪ = After(s, ss)[Future(ss)]
      Current(s)∪ = After(s, ss)[Current(ss)]
    Current(s) = Current(s) − Future(s)
    Previous(s) = ∪_{r⃗_actual ∈ Ppars} Current(s)[r⃗_actual]
    Backward-Slicing (D, s, Rstmts, Ppars, Fpars)
```

Figure 2: Recursion Transformation Algorithm

a statement $s \in ExtraStmts$. It is therefore important to union together information for all the transitive dependence paths from *AddStmts* to s.

In Figure 3, The *Future* iteration sets are not shown. Because both s_1 and s_2 are already included in *Rstmts*, there are no leftover statements. The algorithm terminates without doing any forward slicing.

2.3.3 Conditionals and Non-Unit Strides

Control dependences are a little more difficult to handle than data dependences. In particular, for a control dependence from c to s (both within the same loop), we must keep c and s together within the *same* loop iteration, in addition to preserving their order. This extra requirement could be modeled in transitive dependence analysis by adding dependence edges both from c to s and s to c. This, however, creates a strongly connected component involving all statements control-dependent on c and, therefore, none of these statements would be transformed.

To avoid such restrictions, we identify four cases:

1. *An IF statement controlling a jump out of one or more loops*: The iterations of such loops cannot be reordered, and therefore the recursion transformation is not legal for such loops (and neither is blocking). For now, we simply ignore any loop nest containing such a jump. The remaining cases therefore only consider block-structured control flow.
2. *An IF statement enclosing all loops in the input program fragment*: Such an IF statement can simply be ignored while transforming the loop nests into recursive procedures.
3. *An IF statement with no loops enclosed within it*: Such an IF statement can be handled simply by adding the dependence cycle as described above. Forcing all statements control-dependent on the IF to be executed together is not a significant restriction in this case.
4. *An IF statement with loops inside and outside it, the most complex case.* We handle such statements without including them in *Rstmts* in Figure 2. For each data dependence incident on such an IF statement c, a data dependence edge is added to each statement s control dependent on c, as if c is an integral part of statement s. After $Current(s)\ \forall s \in Rstmts$ are computed, we compute the *Current* iteration set of the IF statement c as $\cup_{s:c\rightarrow s} Current(s)$.

The transformation algorithm may duplicate either an IF statement or a DO statement in two different recursive procedures. To correctly account for side-effects into control flow statements, a preprocessing step pulls out all expressions from DO loop bounds and from IF statements, and assigns them to temporary variables just before each DO or IF statement. (We assume Fortran DO loop semantics, i.e., that loop bounds are evaluated before beginning execution of a loop, so that there are no side effects when executing the loop header itself.) This also ensures that all data dependences into the original DO or IF statement are correctly handled. If a control-flow statement is duplicated *and* any of the expressions have side-effects, array expansion would be needed to pre-compute and store the values of the temporary variables for a range of loop iterations.

Finally, loops with non-unit strides are straightforwrd to handle. When constructing the *Before* and *After* functions (see section 3), we restrict the domain and range of these functions to the original iteration sets of the loops. This directly imposes the necessary stride constraint on all the iteration sets computed thereafter. Since the initial call to the recursive procedure starts with the correct loop bounds, only the required loop iterations are executed.

2.4 Code Generation

Once we have successfully computed the iteration sets, we create a recursive procedure including all the statements in $Rstmts$, with formal parameters $\vec{R}_{formal}$ created in step (2) of Section 2.2.

The code for the base case of the recursion is generated by restricting the iteration set of each statement $s \in Rstmts$ to $Current(s)$. We use Omega's algorithm for code generation from multiple mappings which, given a vector of iteration sets for the statements in a loop nest, directly synthesizes a loop nest to enumerate exactly those instances of the statements while preserving the lexicographic order of statement instances [15]. The techniques we use are similar to those described for code generation in [1].

To generate code for the recursive calls, we divide the ranges of all recursive loops. For example, one simple choice is to divide each range by half. Given m recursive loops with bound parameters $lb_1, ub_1, ..., lb_m, ub_m$, this means to find the middle points $(lb_i + ub_i)/2$, and make 2^m deeper recursive calls. The order of recursive calls is decided by function *Decide-Recur-Order* in Figure 2.

After creating the recursive procedure, we insert an initial call to the procedure before the original code segment, passing the original loop bounds of the recursive loops as actual parameters (say, $\vec{r}_{original}$). It is important to place the initial call before all other statements because the forward-slicing step is not necessarily performed, so that some statement instances that *use* values computed in a recursive procedure may be left out of that procedure.

The original code needs to be transformed so that all the iterations of statements already executed in the initial call will not be executed again. We transform the original code by only executing the *leftover iterations* of each statement $s \in Rstmts$, which is computed by subtracting $Current(s)[\vec{r}_{original}]$ from the original iteration set of s. We generate code for these leftover iteration sets using the same technique as generating code for the base case of the recursion.

Figure 4 shows the pseudo code of the generated recursive procedure for LU based on the computed iteration sets in Figure 3. Because there are no leftover iterations for either s_1 or s_2, the original code is completely replaced with an initial call to LU-recur.

```
call LU-recur(1, N, 1, N)

subroutine LU-recur(lb_J, ub_J, lb_I, ub_I)
      if (stop recursive call) then
          do k = 1, min(N - 1, ub_I-1, ub_J-1)
              if (k ≥ max(lb_J-1, lb_I-1)) then
                  do i = max(k + 1, lb_I), min(N, ub_I)
s1:                   A(i, k) = A(i, k) / A(k,k)
J:                do j = max(k+1,lb_J), min(N, ub_J)
I:                    do i = max(k+1,lb_I), min(N, ub_I)
s2:                       A(i,j) = A(i,j) - A(i,k) · A(k,j)
      else
          m_J = (lb_J + ub_J) / 2
          m_I = (lb_I + ub_I) / 2
          call LU-recur(lb_J,m_J,lb_I,m_I)
          call LU-recur(lb_J,m_J,m_I+1,ub_I)
          call LU-recur(m_J+1,ub_J,lb_I,m_I)
          call LU-recur(m_J+1,ub_J,m_I+1,ub_I)
```

Figure 4: Recursive Code Generated for LU

```
LargeLoops = ∅
UnknownLoops = loops with large iteration range
for each array reference r do
  for each l ∈ UnknownLoops surrounding r do
    if l does not carry reuse of r then
      LargeLoops = LargeLoops ∪ {l}
      UnknownLoops = UnknownLoops − {l}
if (LargeLoops == ∅) then return ∅

KeyStmts = ∅
for each statement s do
  LargeLevel(s) = deepest level of LargeLoops surrounding s
  ReuseLevel(s) = outermost loop level of reuse carried by s
  if (ReuseLevel(s) < LargeLevel(s)) then
    KeyStmts = KeyStmts ∪ {s}
return KeyStmts
```

Figure 5: Choosing Key Statements

2.5 Transformation Decisions For Locality

In this section, we describe how we specialize steps 1, 2, 3 and 5 listed earlier to improve cache performance for sequential programs running on machines with one or more levels of cache.

Choosing Key Statements We choose key statements to be those statements carrying reuse that is not being exploited because the data size swept between reuses is larger than the cache size. All of these identified *candidate key statements* are put into the set $KeyStmts$ in Figure 2. Because any statement $skey \in KeyStmts$ already included in some recursive procedure will not be processed again, we can include redundant key statements without sacrificing either correctness or efficiency.

Figure 5 shows our algorithm to compute $KeyStmts$. We identify reuse simply as data dependences, including input dependences because reuse on read references is important. We identify loops that access large volumes of data ($LargeLoops$ in Figure 5) by analyzing the dependence graph. If there is some array reference inside loop l_i that does not carry any reuse (true, input or output dependence) at l_i, this means that the reference accesses a different data element at each iteration of l_i. Unless loop l_i has a known small range of iterations, we assume that it accesses a large volume of data.

The more loops surrounding a key statement, the more recursive loops can be selected, and therefore the more likely *Compute-Iter-Sets* will succeed in reducing the iteration space for each recursive call. In Figure 2, therefore, we extract key statements from $KeyStmts$ in decreasing order of the number of loops surrounding the key statements.

Choosing Recursive Loops The goal here is to reduce the size of data swept between reuses of values. Therefore for any key statement $skey$ with unexploited data reuse, we choose recursive loops to be the loops surrounding $skey$ that cause skey to access a large volume of data between reuses, i.e., the loops in $LargeLoops$ $(skey)$ that are nested deeper than $ReuseLevel$ $(skey)$ in Figure 5. By reducing the ranges of these recursive loops in the iteration set of $skey$, the size of data accessed between reuses of $skey$ would be reduced.

It is sometimes beneficial to select loops in $LargeLoops$ ($skey$) outside $ReuseLevel$ ($skey$) as recursive loops as well. This may improve inter-reuse among adjacent recursive procedure calls. For example in Matrix Multiply, we choose all the three loops surrounding $skey$ as recursive loops to achieve the hierarchical working set effect in Figure 1. We use compiler options to specify whether or not to include these additional outer loops when selecting recursive loops.

Recursive Order and Termination The choice of recursive call orders can have significant impact on inter-reuse across adjacent recursive procedure calls. In our current implementation, we divide the range of each recursive loop into approximately half, and order the recursive calls so that the innermost recursive loop (in the original loop nest order) is divided first. This preserves the original loop ordering across recursive calls. However, this strategy does not always produce the best performance. Deriving and comparing different strategies for choosing the recursive order requires further research.

To stop recursion, we derive a symbolic expression for the approximate volume of data touched by the outermost recursive loop in each recursive call as a function of the formal parameters of the recursive procedure, taking reuse within the code into account. We directly execute the base code when the estimated data volume is smaller than some minimum volume threshold, specified at runtime.

Verifying Profitability For our purposes, the recursion transformation would not be profitable if *any* statement $s \in Rstmts$ continue to access a large volume of data within a base-level recursive call. This is decided similarly to computing $LargeLoops$ in Figure 5. If $Current(s)$ has a large iteration range at some loop level at or within the outermost recursive loop, and there is some array reference in s that does not carry any dependence at that loop level, the profitability verification of $Current(s)$ fails.

3 Transitive Dependence Analysis

Transitive dependence analysis is a core analysis technique used by the recursion transformation. It computes the $Before$ functions defined in section 2, and the $After$ functions are computed by inverting the $Before$ functions.

For two arbitrary statements s_1 and s_2, $Before(s_1, s_2)$ captures dependence information for all paths from s_1 to s_2 in the dependence graph. For most applications, we require attributes of these dependence paths (such as direction vectors), and therefore transitive dependence analysis is a path summary problem instead of simply a reachability problem on directed graphs. Previous work used symbolic integer sets to represent and propagate transitive dependences [22], and an adapted Floyd-Warshall algorithm to solve the all-pairs path summary problem up front. Because integer set operations are costly, and the adapted Floyd-Warshall algorithm has $O(N^3)$ complexity in all cases for a graph with N nodes, the analysis is very expensive for real world programs.

In this paper, we use a new dependence attribute representation – an Extended Direction Matrix – to represent and propagate dependence information. The computed transitive dependences are then translated into symbolic integer sets to be used as $Before$ functions in section 2. The EDM representation is less precise than the symbolic integer set representation, but it is much more efficient and we believe it will be sufficient for a large class of programs and analysis problems.

We also developed a new demand-driven algorithm to compute the transitive dependences to a single destination vertex (transitive dependences from a single source vertex can be computed similarly). The algorithm is independent of any specific dependence representation, such as EDM or integer sets. Although our algorithm also has $O(N^3)$ worst case complexity to compute all-pairs path summaries, computing single destination transitive dependences can be done in linear time on many dependence graphs in practice.

Below, we first describe the new dependence representation and how to translate it into symbolic integer sets, then present our algorithm for transitive dependence analysis.

3.1 Transitive Dependence Representation

Consider two statements s_1 and s_2 in a dependence graph (not necessarily in the same loop nest). We represent the transitive dependence from s_1 to s_2 using a set of Extended Direction Matrices(EDMs), each EDM representing a dependence relation along some paths from s_1 to s_2.

The Extended Direction Matrix Assuming that the iteration spaces of s_1 and s_2 are $(I_1,I_2,\ldots I_m)$ and $(J_1,J_2,\ldots J_n)$ respectively, an EDM representing a dependence relation from s_1 to s_2 is an $m \times n$ matrix D_{mn}. Each entry $D[i,j]$, represents a dependence direction condition that must hold between iteration I_i of s_1 and iteration J_j of s_2. Each entry can have the following values: $\emptyset$, $=$, $<$, $\le$, $>$, $\ge$, $\neq$, $*$. These dependence direction values have the traditional meaning, except that $\emptyset$ means no condition is required.

The symbolic integer set translated from D_{mn} is a mapping from the iteration space of s_2 to the iteration space of s_1, which given an iteration set of s_2, outputs those iterations of s_1 that satisfy the condition

$$R(D) = \wedge_{1\le i\le m}(\vee_{1\le j\le n}(I_i\ D[i,j]\ J_j)) \qquad (1)$$

where $I_i * J_j = true$, $I_i\ \emptyset\ J_j = false$, other possible values for $D[i,j]$s are translated directly as comparison operators (e.g., $I_i \le J_j$).

The EDM can be computed similarly as traditional data dependences. The only difference is that we need to compute dependence directions among not only iterations of common loops, but also iterations of non-common loops. The extra information is important to precisely represent and propagate dependences along paths involving statements inside non-common loops, eg., the dependence between the k-j-i iteration of s_2 and k-i iteration of s_1 in LU in Figure 3. This enables global transitive dependence analysis as well as compiler transformations for whole procedures.

The Extended Direction Matrix can represent dependence information for both data and control dependences. For data dependences, the direction values have the meaning described above. For a control dependence from s_1 to s_2, the EDM D_{mn} is defined to be:

$$D[i,j] = \begin{array}{l} '=',\ if\ I_i = J_j\ (i.e.,\ are\ the\ same\ loop) \\ '\emptyset',\ if\ I_i \neq J_j, I_i\ is\ a\ common\ loop \\ '*',\ if\ I_i\ is\ not\ a\ common\ loop \end{array}$$

Here, a common loop is one that surrounds both s_1 and s_2.

We also define a particular class of EDMs called identity EDMs. An identity EDM D_{mm} models an identity dependence from a statement s to itself, defined as:

$$D[i,j] = '=' \ if\ i = j;\ '\emptyset'\ if\ i \neq j, \forall 1 \le i,j \le m \qquad (2)$$

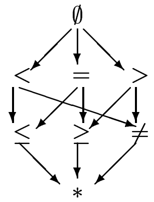
Operations on an EDM: The algorithm below requires computing the *concatenation* of two EDMs, $D_{mn} = D^1_{ml} \cdot D^2_{ln}$, which is defined by:

$$D[i,j] = \sum_{1 \le k \le l} (D^1[i,k] \cdot D^2[k,j]) \tag{3}$$

If D^1_{ml} represents a dependence path p_1 from s_1 to s_2, D^2_{ln} represents a dependence path p_2 from s_2 to s_3, $D^1 \cdot D^2$ represents a dependence path from s_1 to s_3, going through path $p_1 p_2$.

The concatenation $(\cdot)$ and addition $(+)$ operations on dependence directions used in this equation are defined in Figure 6. In the equation, $D^1[i,k] \cdot D^2[k,j]$ computes the dependence direction condition that must hold between loop l_i of s_1 and loop L_j of s_3, due to a dependence path passing through loop J_k of s_2. For example, if $D^1[i,k] = <$ and $D^2[k,j] = \le$, this means that $I_i < J_k \le L_j$, therefore $D^1[i,k] \cdot D^2[k,j] = <$ (i.e. $I_i < L_j$). The addition on direction values in equation 3 summarizes the dependence directions from each loop I_i of s_1 to loop L_j of s_3 due to all the loops surrounding s_2.

$\emptyset \cdot x \rightarrow \emptyset \; \forall x$	$\emptyset + x \rightarrow x \; \forall x$
$* \cdot x \rightarrow * \; \forall x \ne \emptyset$	$* + x \rightarrow * \; \forall x$
$= \cdot x \rightarrow x \; \forall x$	$< + = \rightarrow \le$
$< \cdot \le \rightarrow <$	$= + > \rightarrow \ge$
$> \cdot \ge \rightarrow >$	$< + > \rightarrow \ne$
$\{< \cdot >, < \cdot \ge, \le \cdot >, \ne \cdot <, \ne \cdot >\} \rightarrow *$	$\{= + \ne, > + \le, < + \ge\} \rightarrow *$
(a) concatenation	(b) addition



(c) lattice

Figure 6: Operations on Dependence Directions

Representing a Transitive Dependence Using EDMs: A transitive dependence is represented by a set of EDMs, all having identical numbers of rows and columns. We use T_{mn} to denote a transitive dependence from statement s_1 to s_2 consisting of a set of $m \times n$ EDMs, $\{D^1_{mn}, D^2_{mn}, \ldots\}$. The symbolic integer set representation of T_{mn} is a mapping from $(J_1, J_2, \ldots J_n)$ to $(I_1, I_2, \ldots I_m)$ satisfying

$$R(T) = (R(D^1)) \vee (R(D^2)) \vee \ldots \tag{4}$$

where $R(D^i)$, $i = 1, 2, \ldots$ is defined in equation 1. $Before(s1, s2)$ is simply $R(T)$.

Three operations are defined for arbitrary transitive dependences T^1 and T^2:

$$T^1_{mn} \cup T^2_{mn} = \{D_{mn} | D \in T^1 \; or \; D \in T^2\} \tag{5}$$

$$T^1_{ml} \cdot T^2_{ln} = \{D^1_{ml} \cdot D^2_{ln} \mid D^1 \in T^1 \; and \; D^2 \in T^2\} \tag{6}$$

$$T^*_{mm} = T \cup (T \cdot T) \cup (T \cdot T \cdot T) \cup \ldots \tag{7}$$

The union of T^1 and T^2 is used in the algorithm to combine dependence information along different paths between s_1 and s_2. The concatenation of T^1 and T^2 concatenates all dependence paths in T^1 with all dependence paths in T^2. The transitive closure $(*)$ operation computes all cycles formed by paths in T_{mm} (from a statement s to itself). The infinite numbers of unions and concatenations in equation 7 stop when a fixed point of T^*_{mm} is reached.

Finally, if we define $d^1 \le d^2 \Leftrightarrow d^1 + d^2 = d^2$, where d^1 and d^2 are dependence directions, the set of direction values forms a lattice of depth 4, shown in Figure 6. We define $D^1_{mn} \le D^2_{mn}$ if $D^1[i,j] \le D^2[i,j] \; \forall 1 \le i \le m, 1 \le j \le n$. If $D^1_{mn} \le D^2_{mn}$, the dependence modeled by D^1 is a subset of that modeled by D^2. Given $D^1, D^2 \in T_{mn}$, we can simplify T_{mn} by removing D^1. In practice, transitive dependences usually only contain a few direction matrices after simplification, enabling all operations on transitive dependences to be done within some small constant time (assuming the maximum loop depth is bounded by a small constant).

3.2 Transitive Dependence Analysis Algorithm

Figure 7 shows the algorithm that performs transitive dependence analysis for a single destination vertex. The function *Transitive-Dependence-Analysis* first preprocesses the

```
Transitive-Dependence-Analysis(G, v)
      G: dependence graph
      v: destination vertex
  if (G has not been preprocessed) then
      Preprocess-Cycles(G)
  Compute-Path-Summaries(G, v)

Preprocess-Cycles(G)
      G: dependence graph
  // transforming G into DAG
  Find-SCCs(G, SCC, BackEdges)
  for each scc ∈ SCC do
      for each vertex v_i ∈ scc do
          if (∃(p, v_i) ∈ BackEdges) then
              v'_i = Create-twin-vertex(v_i)
              Change all (p, v_i) ∈ BackEdges to (p, v'_i)
      // computing cycle info.
      // Let the created twin vertices be (v_1, v'_1), ... (v_m, v'_m)
      if (m == 0) continue
      for i = 1, m do
          for each vertex p ∈ scc do
              Compute-Path-Summary-On-DAG(G, p, v'_i)
          for k = 1, i − 1 do
              for each vertex p ∈ scc do
                  T(p, v'_i) ∪= T(p, v'_k) · C(v_k, v'_k) · T(v_k, v'_i)
          C(v_i, v'_i) = (T(v_i, v'_i) ∪ {identity EDM})*

Compute-Path-Summaries(G, v)
      G: dependence graph, v: destination vertex
  for each scc ∈ SCC in reverse topological order do
      // Let twin vertices of scc be (v_1, v'_1), ... (v_m, v'_m)
      for each vertex p ∈ scc do
          Compute-Path-Summary-On-DAG(G, p, v)
      if (m == 0) then continue
      for k = 1, m do
          for each vertex p ∈ scc do
              T(p, v) ∪= T(p, v'_k) · C(v_k, v'_k) · T(v_k, v)

Compute-Path-Summary-On-DAG(G, p, v)
      G: acyclic dependence graph
      p: source vertex, v: destination vertex
  T(p, p) = {identity EDM}
  for each self-cycle e = (p, p) do
      T(p, p) = T(p, p) ∪ {EDM(e)}
  T(p, p) = T(p, p)*
  if (p == v) then return
  T(p, v) = ∅
  if (p is before v in topological order) then
      for each edge e = (p, q) do
          if T(q, v) is not already computed then
              Compute-Path-Summary-On-DAG(G, q, v)
          T(p, v) = T(p, v) ∪ {EDM(e)} · T(q, v)
  T(p, v) = T(p, p) · T(p, v)
```

Figure 7: Transitive Dependence Analysis

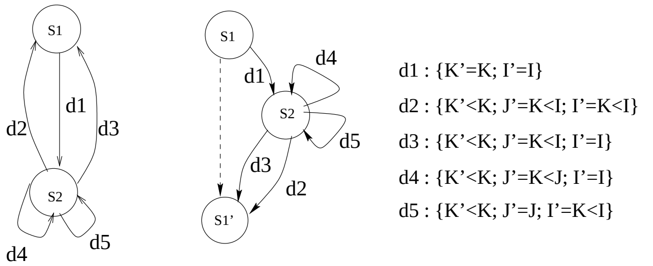


Figure 8: Dependence Graph of LU

dependence graph by calling *Preprocess-Cycles*, then computes single destination transitive dependences on demand for each given destination vertex v by calling *Compute-Path-Summaries*. The overall approach is to transform the graph (once) into an acyclic graph (a DAG) by splitting certain nodes, then to compute path summaries on acyclic paths in the DAG, and then propagate path summaries around the cycles in the original graph.

Preprocess-Cycles transforms the original dependence graph into a DAG by breaking all cycles. The function *Find-SCCs* uses the well-known Tarjan SCC algorithm to find all the strongly connected components and identify all the back edges (for simplicity, self-cycles are not considered back edges). For each vertex v_i which has an incoming back edge $((p, v_i) \in BackEdges)$, we create a new twin vertex v_i' and change all the back edges (p, v_i) to go into v_i' instead. Because all the back edges now go to the new twin vertices which have no outgoing edges, all the original cycles are broken, except for self-cycles. The second half of *Preprocess-Cycles* summarizes cycle information so that the cycles can be recovered later, and will be described below.

Transitive dependences into a single destination vertex are straightforward to compute in linear time on a DAG, as shown in function *Compute-Path-Summary-On-DAG*. In this function, $EDM(e)$ denotes the extended direction matrix associated with edge e in the dependence DAG. Essentially, $T(p, v)$ is computed by first computing the transitive closure of all the self cycles into p $(T(p,p)^*)$, then concatenating $T(p,p)$ with the union of $EDM(e) \cdot T(q, v)$ for each edge $e : p \to q$.

To compute transitive dependences in the original dependence graph, we need to further propagate information computed from the dependence DAG through the broken cycles[2]. In the second part of *Preprocess-Cycles*, for each $scc \in SCC$ with twin vertices $(v_1, v_1'), \ldots (v_m, v_m')$, we precompute two types of cycle information:

- $T(p, v_i')$ $\forall$ nodes $p \in scc$ represents the transitive dependence from p to v_i', including all the original cycles involving $v_1, \ldots v_{i-1}$. To compute $T(p, v_i')$, we first compute dependence information for the paths from p to v_i' on the dependence DAG, then compute $T(p, v_i')$ as the union over all paths $p \rightsquigarrow v_k' \to v_k \rightsquigarrow \ldots \rightsquigarrow v_k' \to v_k \rightsquigarrow v_i'$ $\forall 1 \leq k < i$. Note that $v_k' \to v_k$ is conceptually an edge with an identity EDM connecting the split vertices v_k and v_k', and $v_k \rightsquigarrow \ldots \rightsquigarrow v_k'$ includes all cycles involving $v_1, \ldots v_k$. The ordering of twin vertices for each SCC is crucial here, otherwise, we cannot assume that $C(v_k, v_k')$ and $T(p, v_k')$ have already been computed correctly for all $p \in scc, 1 \leq k < i$.
- $C(v_i, v_i') = (T(v_i, v_i') \cup \{identity\ EDM\})^*$ $\forall 1 \leq i \leq m$ represents the transitive dependence from v_i to itself, including all the original cycles involving $v_1, \ldots v_i$.

[2] Conceptually, this corresponds to restoring the original cycles by adding an edge with identity EDM from each new twin vertex v_i' to its corresponding original vertex v_i, thus obtaining a directed graph equivalent to the original dependence graph.

Function *Compute-Path-Summaries* then uses this information to compute transitive dependences to an arbitrary destination vertex v on the original dependence graph. This is done by computing transitive dependences for each $scc \in SCC$ at a time, and propagating dependences through broken cycles for each vertex $p \in scc$. This step propagates transitive dependences through broken cycles in the same way as in the second part of *Preprocess-Cycles*, using the equation

$$T(p, v) = T(p, v) \cup T(p, v_k') \cdot C(v_k, v_k') \cdot T(v_k, v). \quad (8)$$

This equation computes the union of $T(p, v)$ with dependence information for all paths $p \rightsquigarrow v_k' \rightsquigarrow \ldots \rightsquigarrow v_k \rightsquigarrow v$, where $v_k' \rightsquigarrow \ldots \rightsquigarrow v_k$ includes all cycles involving $v_1, \ldots v_k$, $p \rightsquigarrow v_k'$ and $v_k \rightsquigarrow v$ include all cycles involving $v_1, \ldots v_{k-1}$.

Figure 8 shows the dependence graph of the LU code in Figure 3 both before and after preprocessing. The original dependence graph of LU has only one SCC, including both vertices s_1 and s_2. After preprocessing, vertex s_1 is split. $T(s_1, s_1') = \{d_1\} \cdot \{d_4, d_5\}^* \cdot \{d_2, d_3\}$, $C(s_1, s_1') = (T(s_1, s_1') \cup \{identity\ EDM\})^*$. To compute $T(s_1, s_2)$, we first compute $T(s_1, s_2)$ on the transformed DAG, which yields $\{d_1\} \cdot \{d_4, d_5\}^*$, then propagate the paths through broken cycles, which unions $T(s_1, s_2)$ with $C(s1, s1') \cdot T(s_1, s_2)$. We can compute $T(s_2, s_2)$ similarly.

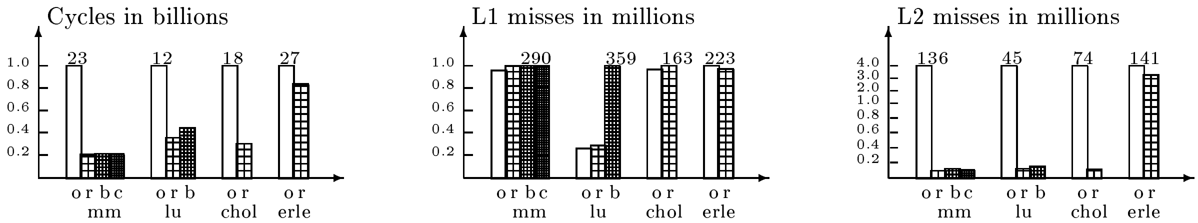
Let $G = (V, E)$, where V and E are the numbers of vertices and edges in dependence graph G respectively. Let $M (M \leq V)$ denote the maximum number of twin vertices created for any SCC in G. Then, the complexity of finding SCCs and creating twin vertices is $O(V + E)$, the complexity of computing cycle information for each SCC is $O(VM^2)$. Therefore the complexity of *Preprocess-Cycles* is $O(V + E + VM^2)$. The complexity of *Compute-Path-Summaries-On-DAG* is $O(V + E)$, the complexity of propagating paths through cycles is $O(VM)$. Therefore the complexity of the *Compute-Path-Summaries* is $O(V + E + VM)$. Although M is $O(V)$ in the worst case (e.g., for an SCC of size $O(V)$ that is fully connected), in practice SCCs in the dependence graph are not densely connected and only a small number of nodes need to be split to break all cycles in each SCC. In such cases, M can be assumed to be bounded by a small constant, and both *PreprocessGraph* and *Compute-Path-Summaries* would require time that is linear in the size of the graph, i.e., $O(V + E)$.

4 Experimental Evaluation

There is already a significant body of experience with recursive algorithms that has demonstrated the value of this approach for a number of different applications. Since the key innovation in our work is to *automate* the transformation of computations to recursive form within a compiler, it is necessary to examine the applicability, compile-time cost, and performance impact of the compiler transformation. We present preliminary results addressing these issues using both measurements and scaled simulations of an existing system.

4.1 Benchmarks and Compiler Performance

We study two classes of benchmark codes. First, we examine three linear algebra codes, matrix multiplication



(o-original version; b-one level blocked version; c-two level blocked version; r-recursive version)
(mm-1025*1025; lu-1024*1024; chol-1536*1536; erle: 256*256*256)

Figure 9: Results from measurements on a uniprocessor SGI workstation

(*mm*), LU factorization (*lu*), and Cholesky factorization (*chol*). Locality transformations such as blocking have traditionally focused on these codes because cache behavior is critical in such codes and because they are very well suited to blocking. So far, we are only able to transform non-pivoting versions of LU and Cholesky. To implement either blocking or recursion for LU or Cholesky with pivoting, a compiler would need to recognize that row interchange and whole column update are commutative [3]. We are confident that, in a compiler with that analysis, conversion to recursive form would be possible for the pivoting versions.

We also study one physical simulation application, Erlebacher (*erle*), which is an implicit finite-differencing scheme for computing partial derivatives on a 3D discretized domain. One interesting effect shown in transforming Erlebacher is that, when multiple different loop nests are transformed into a single recursive procedure, the effect is that of performing loop-fusion on those loop nests. This is because, at a sufficiently low level in the recursion tree, the data produced in the earlier loop nests stay in cache to be reused by the later loop nests within the same recursive call.

We compiled all the benchmarks essentially unmodified. The compiler was able to transform all loops containing unexploited reuse in these benchmarks. All transformation decisions (see section 2.5) are made automatically. Code to estimate the effective data volume accessed by each recursive call is also synthesized automatically. The data volume threshold parameter used to control the recursion depth is specified at runtime. We present simulation results for different values of this parameter. The same generated code for each benchmark was used for both measurement and simulation experiments.

We measured the *compile time* for Erlebacher (the largest of the codes for which we present results here) on a Sun server with 250MHz. UltraSPARC-II processors. Erlebacher has 460 lines, and 10 loop nests are made recursive. The total compile time for this code was 14.7 seconds, including all I/O. A significant part of that time is spent on basic analysis such as control-flow and dependence analysis (these are unfortunately hard to isolate because they happen in a demand-driven fashion in the compiler). The single-loop-nest codes, matrix multiply and LU, are each compiled in under 1 second (total compile time). Although these codes are small, the results show that the compile time for each individual loop nest is extremely fast.

4.2 Performance Measurements

Our measurements were performed on an SGI workstation with a 195 MHz. R10000 processor, 256MB of main memory, separate 32KB first-level instruction and data caches (L1), and a unified 1MB second-level cache (L2). Both caches are two-way set-associative. The cache line size is 32 bytes in L1 and 128 bytes in L2. We used SGI's `perfex` tool (which is based on two hardware counters) to count the *total* number of cycles and the L1 and L2 cache misses. We repeated each measurement 5 or more times and present the average across these runs. The variations across runs was very small.

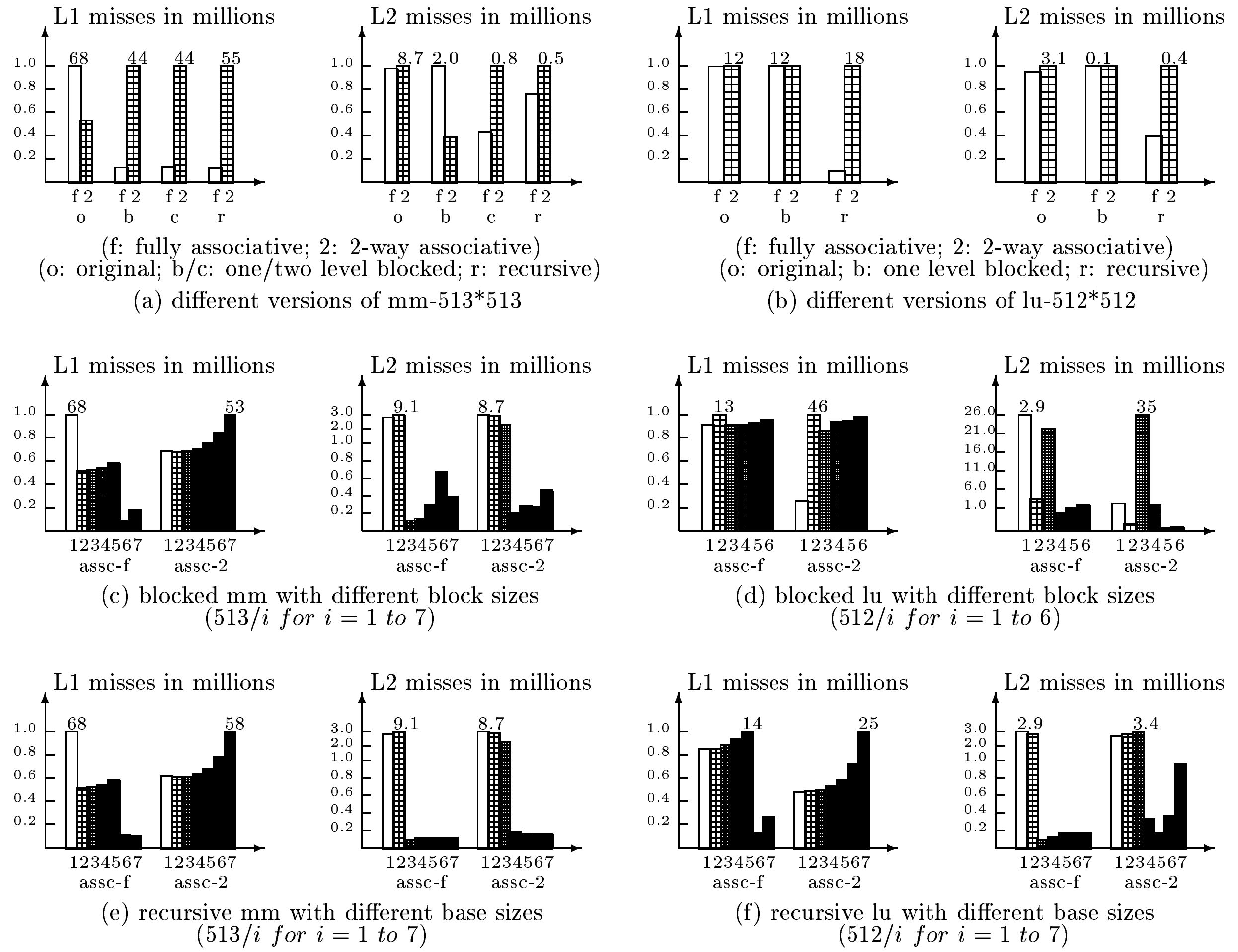
Figure 9 presents performance measurements (cycle times, L1 and L2 cache misses) for the various benchmarks on the SGI workstation. In the graphs, we compare the performance of the compiler-generated recursive code with the original code for each benchmark, with one-and two-level blocked versions of *mm*, and with a one-level blocked version of *lu* that we adopted from [5]. In each group of bars, the bars are scaled relative to the tallest bar and the absolute value of the tallest bar is shown above it. (Note that different groups of bars in the same graph may be scaled differently.)

All the blocked versions were written by hand but should be representative of what a sophisticated compiler would produce. We used odd matrix sizes for matrix multiply to reduce cache conflict misses. We experimented with different block sizes and recursion base sizes and present the ones that performed best.

Compared with the original unblocked codes, the recursion transformation provides large improvements in execution time in all cases except *erle*. These range from a factor of 1.2 in *erle* to factors of roughly 3, 4 and 5 for *lu*, *chol* and *mm*. In *erle*, the only benefits we observe are in fact from loop fusion and not from the blocking effect of recursion. (This was determined by disabling the loop fusion effect, i.e., by forcing the compiler to transform only one loop nest at a time. Those results are not shown here.)

Compared with the blocked versions, the recursion transformation is similar in performance to one-level and two-level blocking for *mm*, and performs about 25% better than one-level blocking for *lu*.

The cache miss measurements show little improvement from either blocking or recursion in the L1 cache compared with the original (and the blocked version of *lu* is significantly worse). In the L2 cache, however, both blocking and recursion show large improvements, except for Erlebacher. The lack of benefits in the L1 cache are directly attributable to increased conflict misses, as the next section illustrates. This sensitivity of conflict misses to block sizes is well known for blocking. In this context, it is interesting to explore the effect of base recursion sizes on conflict misses and on overall performance. We used cache simulation to study these issues in more detail, as described next.



(f: fully associative; 2: 2-way associative)
(o: original; b/c: one/two level blocked; r: recursive)
(a) different versions of mm-513*513

(f: fully associative; 2: 2-way associative)
(o: original; b: one level blocked; r: recursive)
(b) different versions of lu-512*512

(c) blocked mm with different block sizes ($513/i$ *for* $i = 1$ *to* 7)

(d) blocked lu with different block sizes ($512/i$ *for* $i = 1$ *to* 6)

(e) recursive mm with different base sizes ($513/i$ *for* $i = 1$ *to* 7)

(f) recursive lu with different base sizes ($512/i$ *for* $i = 1$ *to* 7)

Figure 10: Results from simulation of 2-way and fully-associative caches

4.3 Cache Simulation

We used the Memory Hierarchy Simulator (MHSIM) from Rice University [19] to study the cache performance of the recursive and blocked codes, focusing on *mm* and *lu*. In order to study cache conflicts, we compared a two-way associative cache with a fully associative one of the same total capacity. In order to reduce simulation time, we scaled down the benchmark problem sizes and scaled down cache sizes proportionately, so that both the L1 and L2 caches were 1/4 of the sizes on the SGI workstation used for the measurements above (i.e., 8K L1 and 256K L2 caches). The line sizes were unchanged.

Figure 10 presents simulation results, where figures (a)-(b) present data on the effect of cache conflicts, (c)-(d) present the effect of different block sizes, and (e)-(f) present the effect of different recursion base sizes. The bars are scaled as in Figure 10. Briefly, the primary observations we make are as follows.

First, for all the versions of *mm* and for the recursive version of *lu*, conflict misses are much more severe in the L1 than in the L2 cache. This is as expected because the L1 cache is much smaller than L2. In a few cases, the fully associative cache actually performs worse than the 2-way associative cache. This can be seen in the L1 cache for the original and one-level blocked versions of *mm*. This phenomenon is a known defect of the LRU replacement policy [24]. It happens because a row of matrix C and a row of matrix A together just exceed the cache size, so that each element of C is evicted from the fully associative cache just before it would be reused. In the two-way cache, a more recently used value is evicted in many cases, keeping the value of C in cache for the next use. This effect is very specific to particular combinations of problem size, cache size, and access pattern, and does not happen in most of the cases shown in Figure 10.

Second, the cache conflict miss effect is much more severe for the blocked and recursive versions than for the original versions, except for blocked *lu*. This is because the working sets of the original versions of *mm* and *lu* are much larger than the L1 cache, so that cache conflict misses are overshadowed by capacity misses. The same is true for blocked *lu* because only one loop dimension is blocked in *lu*. The effect of L1 cache conflicts on blocking and recursion are clearly seen in Figure 10(c)-(e). When the block sizes or recursion base sizes become small enough so that the working set fits in L1 cache, L1 cache misses decrease dramatically in the fully associative cache, but start to increase steadily in the 2-way associative cache.

The above results indicate that techniques proposed to manage conflict misses for blocked codes [17, 8] could be important for recursion as well. Including such techniques in the recursion transformation is outside the scope of this paper. Recently however, Gatlin and Carter [12] showed that the performance of divide-and-conquer programs can be greatly enhanced by choosing whether or not to apply conflict reduction techniques (e.g., data copying) at each recursive level, based on architectural parameters.

All of these results suggest that it is important to examine the inherent benefits of recursion and blocking, in the absence of conflict misses. We can study this question for a range of recursion base sizes and block sizes, using the results for the fully associative cache in Figure 11 (c)-(e). These results show that for the recursive versions, after the recursion base working set fits in L2 cache, further increasing recursion depth leaves the L2 cache misses relatively constant. The same conclusion also holds for the L1 cache. This enables us to increase recursion depth until the working set fits in the smallest cache, therefore achieving the best performance for multiple levels of cache simultaneously. In contrast, for the blocked versions, after the working set fits in L2 cache, further reducing the block size makes L2 misses increase steadily (because of lost inter-block reuse). Therefore if we choose the block size so that the working set fits in the smallest cache, we cannot achieve the best performance for the larger caches. Either multi-level blocking is needed, or a compromise must be made to choose a block size that achieves the best overall performance with non-optimal performance for some levels of caches.

5 Related Work

To the best of our knowledge, no previous compiler work exists that automatically converts existing loops into recursive form. If that is true, this represents a new compiler transformation with multiple potential applications, including locality management for memory hierarchies and effective parallelization for shared memory systems. As described in the Introduction, researchers have applied recursion by hand for both single-processor and shared-memory multiprocessor codes [13, 21, 10, 12, 2]. The variety of experimental benefits these studies have demonstrated, as well as the theoretical results of Frigo et al. [10], provide strong motivation for developing compiler support to make this transformation more widely accessible to application programmers.

A number of code transformations have been proposed for improving locality in programs, including blocking, loop fusion, loop interchange, and loop reversal [11, 30, 29, 17, 4, 18, 8, 25]. The recursion transformation (as used here) is essentially a form of blocking, with two key differences. First, it combines the effect of blocking at multiple different levels into a single transformation. Second, the recursion transformation unifies both blocking and loop fusion when applied to multiple different loop nests.

One disadvantage of most compiler algorithms for blocking (as well as the other control structure transformations described above) is that they are limited by the loop nest structure of the original program. For example, before automatic loop blocking can be performed on LU, a compiler must first perform index-set splitting, strip-mining, and loop interchange to obtain a loop structure amenable to blocking [5]. Wolf and Lam [29] present a unified algorithm that selects such compound sequences of transformations directly using a model of reuse in loops. Kodukula et al. [16] proposed an alternative approach called data shackling, where a tiling transformation on a loop-nest is described in terms of a tiling of key arrays in the loop nest (a data shackle). Multi-level blocking is described by composing multiple data shackles. This approach is independent of the original control structure. Like the data shackling approach, the recursion transformation directly transforms a loop nest (e.g., the one in LU mentioned above) into a recursively blocked structure in a single transformation, using an algorithm that is independent of loop structure.

A key advantage of blocking over recursion is that much smaller block sizes can be used with blocking (including blocking for registers [6]), whereas recursion would incur high overhead for very small block sizes. This suggests that it might be beneficial to use blocking within the base-case code to achieve small block sizes, while using the recursive structure to achieve the effect of multi-level blocking.

Rosser and Pugh [27, 28] proposed iteration space slicing, which is a powerful technique that uses transitive dependence analysis to achieve loop transformations independent of the original loop control structure. They applied iteration space slicing for improving cache locality, primarily by using it to fuse different loop nests. We have used iteration space slicing in our recursion transformation. The loop fusion effect we obtain follows directly from this use of iteration space slicing.

Finally, the automatic recursion transformation can play an important complementary role to several recursive data organizing techniques that have been proposed [7, 20]. For example, Chatterjee et al. show that recursive reordering of data produces significant performance benefits on modern memory hierarchies, and they argue that recursive control structures may be needed to fully exploit their potential. Conversely, we believe that our work can specially benefit from such data reorganizing techniques by matching the data organization carefully to the computation order. This is a rich avenue for further research.

6 Summary and Future Work

This paper presents a new compiler transformation that converts ordinary loop nests into recursive form automatically. Evidence from previous research shows that such a transformation is of potential value for several different purposes. We apply this transformation to improve locality for uniprocessor cache hierarchies. Our preliminary experiments indicate that the transformation is powerful enough to transform complex loop nests, and achieves substantial benefits on several linear algebra codes even for a simple two-level cache hierarchy.

This paper also presents a new, very efficient algorithm for computing transitive dependence information on a dependence graph. The new algorithm makes the recursion transformation very fast in practice. Moreover, transitive dependence analysis is a powerful technique that has much wider applicability beyond the transformation presented here.

The wider potential of both the recursion transformation and of transitive dependence analysis suggests that further research on these techniques could be very fruitful. One direction we plan to pursue is to explore the benefits of combining recursive data organizations with the recursion transformation, as discussed in Section 5. A second direction would be to explore how the recursion transformation can be extended to parallel shared memory (e.g., OpenMP) applications. Finally, it would be interesting to examine how transitive dependence analysis could be used to improve existing compiler optimizations or support new ones.

7 Acknowlegement

We thank John Mellor-Crummey for providing and helping us with the MHSIM simulator, Evan Rosser for valuable comments on the presentation of our transitive dependence analysis algorithm, and Chen Ding for helping us explain certain cache behavior in the simulations.

References

[1] V. Adve and J. Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[2] B. Alpern, L. Carter, and J. Ferrante. Space-limited Procedures: A Methodology for Portable High-Performance. In *International Working Conference on Massively Parallel Programming Models*, 1995.

[3] S. Carr and K. Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, Dec. 1989.

[4] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[5] S. Carr and R. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Transactions on Mathematical Software*, 23(3), 1997.

[6] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical Tiling for Improved Superscalar Performance. In *Proc. 9th International Parallel Processing Symposium*, Santa Barbara, CA, Apr. 1995.

[7] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts For Hierarchical Memory Systems. In *Proc. 13th ACM Int'l Conference on Supercomputing, Phodes Greece*, 1999.

[8] S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[9] E. Elmroth and F. Gustavson. Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance. Technical report, IBM T.J. Watson Research Center.

[10] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. *Submitted for publication.*

[11] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct. 1988.

[12] K. S. Gatlin and L. Carter. Architecture-Cognizant Divide and Conquer Algorithms. In *Proc. SC99: High Performance Networking and Computing*, Nov 1999.

[13] F. G. Gustavson. Recursion Leads To Automatic Variable Blocking For Dense Linear-algebra Algorithms. *IBM J. Res. Develop*, 41(6), Nov 1997.

[14] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, Apr. 1996.

[15] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, Feb. 1995.

[16] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[17] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA, Apr. 1991.

[18] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[19] J. Mellor-Crummy and D. Whalley. MHSIM: A Configurable Simulator for Multi-level Memory Hierarchies. Technical Report TR-00-357, Rice University.

[20] J. Mellor-Crummy, D. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance For Irregular Applications. In *Proc. 13th ACM Int'l Conference on Supercomputing, Phodes, Greece.*, 1999.

[21] Robert D. Blumofe and Matteo Frigo and Christopher F. Joerg and Charles E. Leiserson and Keith H. Randall. An Analysis Of Dag-Consistent Distributed Shared-Memory Algorithms. In *Proc. Eighth Annual ACM Symposium on Parallel Algorithms and Architectures(SPAA), Padua, Italy*, June 1996.

[22] E. J. Rosser. *Fine Grained Analysis Of Array Computations.* PhD thesis, Dept. of Computer Science, University of Maryland, Sep 1998.

[23] H. Sharangpani. Intel Itanium Processor Microarchitecture Overview. *Presentation at Intel Microprocessor Forum.*

[24] J. E. Smith and J. R. Goodman. A Study of Instruction Cache Organizations and Replacement Policies. In *Proc. Tenth Annual Symposium on Computer Architecture*, Stockholm, Sweden, June 1983.

[25] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proc. SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[26] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, pages 352–357, July 1984.

[27] William Pugh and Evan Rosser. Iteration Space Slicing And Its Application To Communication Optimization. In *11th ACM International Conference on Supercomputing, Vienna, Austria*, July 1997.

[28] William Pugh and Evan Rosser. Iteration Space Slicing For Locality. In *LCPC 99*, July 1999.

[29] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[30] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Reno, NV, Nov. 1989.