International Journal of Parallel Programming, Vol. 29, No. 3, 2001

Data-Centric Transformations for Locality Enhancement[1]

Induprakas Kodukula[2] and Keshav Pingali[2, 3]

Received October 1999; revised September 2000

On modern computers, the performance of programs is often limited by memory latency rather than by processor cycle time. To reduce the impact of memory latency, the restructuring compiler community has developed locality-enhancing program transformations such as loop permutation and tiling. These transformations work well for perfectly nested loops (loops in which all assignment statements are contained in the innermost loop), but their performance on codes such as matrix factorizations that contain imperfectly nested loops leaves much to be desired. In this paper, we propose an alternative approach called *data-centric transformation*. Instead of reasoning directly about the control structure of the program, a compiler using the data-centric approach chooses an order for the arrival of data elements in the cache, determines what computations should be performed when that data arrives, and generates the appropriate code. At runtime, program execution will automatically pull data into the cache in an order that corresponds approximately to the order chosen by the compiler; since statements that touch a data structure element are scheduled close together, locality is improved. The idea of data-centric transformation is very general, and in this paper, we discuss a particular transformation called *data-shackling*. We have implemented shackling in the SGI MIPSPro compiler which already has a sophisticated implementation of control-centric transformations for locality enhancement. We present experimental results on the SGI Octane comparing the performance of the two approaches, and show that for dense numerical linear algebra codes, data-shackling does better by factors of two to five.

KEY WORDS: Locality enhancement; restructuring compilers; caches; program transformation.

[1] This work was supported by NSF Grants EIA-9726388, ACI-9870687, EIA-9972853, and ACI-0085969.

[2] Department of Computer Science, Cornell University, Ithaca, New York 14853.

[3] To whom correspondence should be addressed. E-mail: pingali@cs.cornell.edu

0885-7458/01/0600-0319$19.50/0

1. INTRODUCTION

The memory system of modern computers is hierarchical in organization. Since the latency of data accesses may increase by an order of magnitude or more from one level of the hierarchy to the next, programs run well on such machines only if most of their accesses are satisfied by the faster levels of the memory hierarchy. For this to happen, a program must exhibit *locality of reference*. If accesses to a memory location are clustered together in time, the program is said to exhibit *temporal locality*, which is beneficial since it is likely that all these accesses other than the first one will be satisfied by the faster levels of the memory hierarchy. If the addresses of memory locations accessed successively by a program are close to each other, the program is said to exhibit *spatial locality*, which is beneficial because the unit of transfer between different levels of the memory hierarchy is a line or block, so successive memory accesses to addresses that are close to each other are likely to be satisfied mostly by the faster levels of the memory hierarchy.

For many applications, straight-forward coding of standard algorithms results in programs that exhibit poor locality of reference. Unfortunately, taking locality into consideration when writing programs complicates the task of programming enormously. One solution is to write optimized, machine-specific routines only for certain core operations, and code all other applications in a high-level language, invoking these routines when appropriate. This achieves high performance without sacrificing portability of the application code. The numerical analysis community has followed this approach by hand coding machine-specific programs for the Basic Linear Algebra Subroutines (BLAS),[1] and layering all other dense numerical linear algebra software on top of these routines. However, most applications have to be recoded at a fundamental level to use such libraries. To exploit the BLAS for example, the numerical analysis community has had to invest considerable effort in designing block algorithms and implementing them in the LAPACK library,[2] as described in Section 3. Furthermore, these libraries are not useful in writing other applications such as PDE solvers that use explicit methods since these codes cannot be restructured to expose BLAS operations.

The compiler community has explored a more general-purpose approach in which locality is enhanced by automatic program transformation. To explain this technology, it is necessary to introduce the following definitions.

Definition 1. A *perfectly nested* loop nest as a loop nest in which all assignment statements are contained an the innermost loop of the loop nest. The matrix multiplication code shown in Fig. 1e as in example.

```
sum = 0
do i = 1, n
  sum += x(i) * y(i)
```

(a) Inner (dot) Product

```
do i = 1, n
  z(i) += α * x(i) + y(i)
```

(b) Scalar * x + y (Saxpy)

```
do i = 1, m
  do j = 1, n
    y(i) += A(i,j) * x(j)
```

(c) Matrix Vector Product

```
do i = 1, n
  x(i) = b(i)
  do j = 1, i-1
    x(i) = x(i) - L(i,j)*x(j)
  x(i) = x(i)/L(i,i)
```

(d) Triangular solve

```
do i = 1, m
  do j = 1, n
    do k = 1, p
      C(i,j) += A(i,k) * B(k,j)
```

(e) Matrix Multiplication

Fig. 1. Basic linear algebra subroutines.

An *imperfectly nested loop* nest is a loop nest in which one or more assignment statements an contained an some but not all of the loops of the loop nest. The triangular solve code in Fig. 1d is an example.

An *instance* of a statement is an execution of that statement for particular index values of its surrounding loops.

For perfectly nested loops, there is an elegant matrix-based theory for synthesizing linear loop transformations for locality enhancement.[3–12] These transformations, followed by loop tiling,[13] are performed routinely by many production compilers such as the SGI MIPSPro. The theory of loop transformations is much less developed for imperfectly nested loops. Some compilers use transformations like jamming, distribution and statement sinking[13, 14] to convert imperfectly nested loops into perfectly nested loops, and enhance locality by suitably transforming the resulting perfectly nested loops. However, there is no systematic theory for determining the order in which these imperfectly nested loop transformations must be applied, and the quality of the final tiled code may depend critically on this order, as we explain in Section 4.

The approach described in the previous paragraph can be called *control-centric* program transformation because it reasons about the control structure (loop structure) of the program and modifies this control structure to enhance locality. In this paper, we describe a different approach to locality enhancement called *data-centric* program transformation which

addresses some of the limitations of control-centric approaches. Instead of reasoning directly about the control structure of the program, a data-centric approach fixes an order of traversal through data structure elements, and determines which computations should be performed when a data structure element is touched. Intuitively, the compiler chooses an order for the arrival of data elements in the cache, determines what computations should be performed when that data arrives, and generates the appropriate code. At runtime, program execution will automatically pull data into the cache in an order that corresponds approximately to the order chosen by the compiler; since statements that touch a data structure element are scheduled close together, locality is improved.

The idea of data-centric transformation is very general, but in this paper, we focus on a particular data-centric transformation called *data-shackling* which was designed for locality enhancement of dense numerical linear algebra codes. The traversals allowed are along the co-ordinate axes of the array (that is, left-to-right and top-to-bottom, and reversals of these), and code scheduling is done by choosing a *data-centric reference* for each statement, which determines when its instances are executed. The array itself is not physically copied to make the storage order of elements the same as the traversal order chosen by the data-shackle, although this can be done if the overhead of copying the array is small compared to the resulting performance enhancement.

The rest of this paper is organized as follows. In Section 2, we describe the dense numerical linear algebra codes that constitute the work-load for our research, These codes are divided into the Basic Linear Algebra Subroutines (BLAS) such as matrix multiplication and triangular solves, and matrix factorizations such as Cholesky and LU factorization. The advantage of this work-load is that hand coded versions of these programs are available publicly for most platforms, so it is possible to compare the performance of compiler-generated code with that of good hand-tuned code. This is not possible with other work-loads such as the SPEC and Perfect benchmarks. In Section 3, we describe *blocking* which is the approach taken by the numerical analysis community to improve the performance of these codes on memory hierarchies. In Section 4, we describe control-centric approaches to locality enhancement by program transformation. In Sections 5–7, we describe the data-centric approach.

Data-shackling has been implemented in SGIs MIPSPro compilers for the Octane work-station line by one of the authors (Kodukula). This implementation incorporated a number of heuristics for choosing various parameters required for data-shackling. These heuristics are described in Section 8; since compile time is an issue for production compilers, the heuristics are relatively simple. Experimental results based on this implementation are

described in Section 9. Finally, Section 10 describes ongoing work in data-centric compilation.

2. WORK LOAD AND EXPERIMENTAL PLATFORM

We now describe the dense numerical linear algebra codes that constitute our workload. Following the numerical analysis literature, we divide these codes into the *Basic Linear Algebra Subroutines* (*BLAS*) and *matrix factorizations*.[1] The BLAS contain simple codes like inner-product of vectors, matrix vector product, triangular solve, and matrix matrix multiplication. The important matrix factorizations are Cholesky, LU and QR factorizations. We also describe the hardware on which all experiments were performed.

2.1. Basic Linear Algebra Subroutines

The following five operations occur frequently in applications.[1]

- *Dot product*: Given two column vectors x and y, computes the inner product $x^T y$.
- *Saxpy*: Given a scalar α and column vectors x and y, computes the column vector z equal to $\alpha * x + y$.
- *Matrix Vector Product*: Given an $m \times n$ matrix A and a column vector x, computes $y = A * x$.
- *Triangular solve*: Given a square lower triangular matrix L with nonzero diagonal elements and a column vector b, solve the system $Lx = b$.
- *Matrix Multiplication*: Given an $m \times p$ matrix A, an $p \times n$ matrix B and an $m \times n$ matrix C, computes $C = C + A * B$.

Figure 1 shows pseudo-code for each of these core routines.

Three important properties of each of these core routines are (i) amount of data touched; (ii) the number of floating point operations; and (iii) the average amount of data reuse (which is the ratio of the two previous quantities). Table I summarizes this information for each of the core operations.

The information in Table I provides a guide to the potential performance of BLAS on a machine with a memory hierarchy. Dot product, called a Level-1 BLAS, performs $2 * n$ floating-point operations on $2 * n$ data

Table I. Behavior of Some BLAS

Operation	Data touched	Computation	Algorithmic reuse	Level
$z = \alpha * x + y$	$O(n)$	$O(n)$	x, y, z: $O(1)$	1
$A * x$	$O(n^2)$	$O(n^2)$	x: $O(n)$, A: $O(1)$	2
$C = C + A * B$	$O(n^2)$	$O(n^3)$	A, B, C: $O(n)$	3

elements. On the average, one data item must be touched for each floating-point operation that is performed, so there is little reuse of data even in the algorithm. The characteristics of saxpy are similar. Matrix vector product performs $2 * n^2$ operations on $n^2 + 2 * n$ data, so the amount of data touched per floating-point operation is half of that in the case of Level-1 BLAS. There is no reuse of matrix elements but each of the vectors exhibits $O(n)$ amount of reuse. Matrix vector product and triangular solve are called Level-2 BLAS. Finally, matrix multiplication performs $O(n^3)$ operations on $O(n^2)$ data and is called a Level-3 BLAS operation. Clearly, there is significant algorithmic reuse of data in matrix multiplication, and exploiting this reuse is the key to good performance on a machine with a memory hierarchy.

2.2. Matrix Factorizations

We will consider the following matrix factorization codes: (i) Cholesky factorization; (ii) LU factorization; and (iii) QR factorization.

2.2.1. Cholesky Factorization

Cholesky factorization is used to solve the system of equations $\mathsf{Ax} = \mathsf{b}$, where A is a symmetric positive-definite matrix, by factorizing A into the product LL^T, where L is lower-triangular, and solving the two resulting triangular systems. To save space, the lower triangular part of A is overwritten with the factor L.

Cholesky factorization, like matrix multiplication, has three nested loops although these loops are imperfectly nested. All six permutations of these loops are legal, and distributing the loops in one of these versions gives a total of seven versions of Cholesky factorization that we discuss in this paper. Pseudocode for one of these versions is shown in Fig. 2; pseudocode for the other versions are shown in Figs. 18–24.

The most commonly described version is the so-called kij version (also known as the right-looking version), and it is shown in Fig. 2. This version processes the columns of the matrix in left to right order as follows: the square root of the diagonal element of the current column is computed, the

```
do k = 1, n
  S1: A(k,k) = sqrt (A(k,k))   //square root step
  do i = k+1, n
    S2: A(i,k) = A(i,k) / A(k,k)   //scale step
    do j = k+1, i
      S3: A(i,j) -= A(i,k) * A(j,k) //update step
```

Fig. 2. kij-fused version of Cholesky factorization.

portion of this column below the diagonal is scaled with this value and the outer-product of this portion of the column with its transpose is used to update the lower triangular portion of the matrix to the right of the current column. These are known as the *square root*, *scale*, and *update* steps respectively. For obvious reasons, this version of Cholesky factorization is also known as *right-looking column Cholesky factorization.* Distributing the i loop over the scale step and the update loop produces another kij version of Cholesky factorization, shown in Fig. 18. Permuting the two update loops in the code of Fig. 18 gives the kji version shown in Fig. 20.

Right-looking Cholesky factorization performs updates *eagerly* in the sense that the columns to the right of the current column are updated as soon as that column is computed. An alternative is to perform the updates *lazily*, which means that a column is updated only when it becomes current. This leads to the *left-looking column Cholesky factorization* code (also called the jki version) shown in Fig. 22 which applies updates from all columns to the left of the current column before performing the square root and scaling steps. Permuting the i and k loops gives the jik version shown in Fig. 21.

Finally, there are two versions of Cholesky factorization called the ijk and ikj versions that process the matrix by row rather than by column. These are shown in Figs. 23 and 24. The ijk version performs inner-products, so it is also known as *dot Cholesky* while the ikj version in contrast is rich in saxpy operations.

2.2.2. LU Factorization

LU factorization is used to solve linear systems when the matrix is not guaranteed to be symmetric positive definite. To improve the stability of this process, an operation called *partial pivoting* is performed during the factorization. LU factorization, like Cholesky factorization and matrix multiplication, has three nested loops that may be permuted to produce a number of versions. One version of LU factorization is shown in Fig. 3. The k loop, which is outermost, walks over the columns of the matrix. In each iteration of this loop, the entries in column k below the diagonal of the matrix are examined, and the row that contains the element of largest

```
do k = 1, n
  temp = 0.0d0
  m = k
  //find pivot row
  do i = k, n
    d = A(i,k)
    if (ABS (d) .gt. temp)
      temp = abs(d)
      m = i
  if (m .ne. k)
    ipvt(k) = m
    //row permutation
    do j = k, n
      temp = A(k,j)
      A(k,j) = A(ipvt(k),j)
      A(ipvt(k),j) = temp
  //scale loop
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
  //update loops
  do i = k+1, n
    do j = k+1, n
      A(i,j) -= A(i,k) * A(k,j)
```

Fig. 3. kij version of LU with pivoting.

magnitude is determined (call it row m). If m and k are distinct, the portion of row of k to the right of the diagonal is swapped with the corresponding portion of row m (this is called partial pivoting). Scale and update steps are then performed as in Cholesky factorization, but the update is applied to the sub-matrix below and to the right of the diagonal element A(k, k). Note that the i and j loops in the update step can be interchanged, giving rise to two different versions of LU factorization with pivoting.

This version of LU factorization is called right-looking LU factorization. As in the case of Cholesky factorization, there are two left-looking versions in which updates to a column are delayed until that column becomes current. Interactions between updates and pivoting are sufficiently complex that there is no direct analog of row Cholesky factorization for LU factorization with pivoting. It is possible to perform LU factorization row by row, but this code must do column pivoting, and is therefore a different algorithm than the left- and right-looking column versions. We do not consider this version in this paper.

2.2.3. QR Factorization

QR factorization, shown in Fig. 4 is used in eigenvalue computations, and it factorizes A into a product Q*R where Q is an orthonormal matrix

and R is upper triangular. The Householder variant of QR factorization proceeds through the matrix A column by column. For each column, a Householder vector is determined such that when the corresponding Householder reflection is applied, the portion of the current column strictly below the diagonal contains only zeros. For a vector x, if e_1 represents the unit vector with a 1 in the first entry, and zeros in all other entries, $v = (x - \|x\|_2 * e_1)/\|(x - \|x\|_2 * e_1)\|$ represents a unit-length Householder vector such that on applying a Householder reflection to x, all entries except the first are zeroed out. Once a Householder vector v has been determined for the current column, a Householder reflection can be applied to the rest of the matrix. Conceptually, a Householder reflection can be thought of as multiplying the rest of the matrix by $(I - 2vv^T)$, which would take $O(n^3)$ operations for v of length n. However, this reflection can actually be accomplished in $O(n^2)$ operations by using a two-step process. The key observation is that for a matrix B, $(I - 2vv^T) * B = B - 2v * v^T * B = B - 2v * w$, where $w = v^T * B$. Thus the first step computes w using a matrix vector computation and the second step updates the rest of the matrix using an outer product update computation.

2.3. Hardware Platform

The experimental results reported in this paper were obtained on an unloaded Octane workstation with an R10000 processor running at 195 MHz. The R10K can perform one load/store operation and two floating point operations per cycle, giving it a theoretical peak performance of 390

```
do k = 1, n
  norm = 0
  do i = k, n
    norm = norm + A(i,k) * A(i,k)
  norm2 = dsqrt (norm)
  asqr = A(k,k) * A(k,k)
  A(k,k) =
    dsqrt(norm-asqr+((A(k,k)-norm2)**2))
  //Householder vector computation
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
  //Reflection - (matrix vector product)
  do i = k+1, n
    w(i) = 0
    do j = i, n
      w(i) += A(j,i) * A(j,k)
  //Reflection - (outer product update)
  do j = k+1, n
    do i = k+1, n
      A(i,j) = A(i,j) - 2 * A(i,k) * w(j)
```

Fig. 4. QR factorization.

MFlops. The processor has 32 logical registers and 64 physical registers. The workstation was equipped with separate first-level (L1) instruction and data caches of size 32Kb each, and a second-level (L2) unified cache of size 1MB. The L1 cache is nonblocking with a miss penalty of 10 cycles, and it is organized as a two-way set associative cache with a line size of 32 bytes. The L2 cache is also nonblocking with a miss penalty of 70 cycles, and it is organized as a two-way set associative cache with a line size of 128 bytes. Therefore, the four highest levels of memory hierarchy are the registers, the L1 and L2 caches and main memory.

3. LIBRARY APPROACHES

In this section, we discuss the approach taken by the numerical linear algebra community to produce high-performance codes for the work-load described in Section 2, and argue that it is difficult for a restructuring compiler to mimic this approach directly.

The numerical linear algebra community has taken a layered approach to the problem of implementing portable, high-performance code for matrix applications.

1. Machine-specific code is written for each of the BLAS. These codes are not portable since the performance optimizations in these codes are machine-specific.
2. Matrix factorizations are expressed as *block algorithms* rather than as point algorithms. The restructuring of point algorithms to block algorithms exposes BLAS-3 like matrix multiplication and triangular solve with multiple right-hand sides which are executed by invoking machine-specific BLAS. This is the approach followed in the LAPACK library.[2]

The upshot of this strategy is that only the BLAS are not portable; the matrix factorization codes are layered on top of the BLAS, and are machine-independent.

3.1. BLAS Routines

We use matrix multiplication to discuss how the BLAS are implemented. The naive version in Fig. 1e does not exploit data reuse effectively; for example, for every iteration of the outermost loop, the matrix B is read in its entirety. If the matrices are much larger than the cache, none of the reuse of B is exploited. The solution is to use a block algorithm that performs a sequence of small matrix multiplications, each of which multiplies

```
do t1 = 1 .. ⌈( n/25)⌉
 do t2 = 1 .. ⌈( n/25)⌉
  do t3 = 1 .. ⌈( n/25)⌉
   do It = (t1-1)*25 +1 ..
min(t1*25,n)
    do Jt = (t2-1)*25 +1 ..
min(t2*25,n)
     do Kt = (t3-1)*25 +1 ..
min(t3*25,n)
      C(It,Jt) = C(It,Jt) + A(It,Kt) * B(Kt,Jt)
```

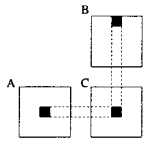


Fig. 5. Blocked code for matrix multiplication.

a submatrix of A with a submatrix of B, accumulating the result in a submatrix of C, as shown in Fig. 5. If these submatrices are small enough to fit into the cache, the number of capacity misses decreases substantially.

How big the submatrices should be clearly depends on the size of the cache and is therefore machine-specific, but it can conveyed as a parameter to the block code. The need for machine-specific code arises because a good code for matrix multiplication must pay careful attention to register allocation of array elements. If the k loop is innermost in the submatrix multiplication code, C(i, j) can be held in a register, reducing the number of loads and stores in each inner loop iteration by 2. This can improve performance for two reasons: (i) register accesses are faster than cache accesses; and (ii) most microprocessors have a small number of pipes to memory, so having many loads and stores in an inner loop can throttle the performance of the processor pipeline. On the other hand, it is also advantageous to have the i and j loops innermost since B(k, j) and A(i, k) respectively become invariant in the innermost loop, and can be read once and stored in a register. The solution to these conflicting demands is to *register tile* the submatrix multiplication itself, and choose the size of the tiles so that array values can be read and written in registers in the innermost loop. In addition to register tiling, the innermost loop must be software pipelined to reduce the effect of load latencies. These considerations mandate the use of very machine-specific code in writing high-performance BLAS.

Detailed information regarding the implementation of BLAS on a modern high-performance computer can be found (see Agarwal and Gustavson[15]). Figure 6 shows the performance of handcoded BLAS on the SGI Octane. These routines were implemented by Mimi Celes at SGI.

3.2. Block Matrix Factorizations

From Fig. 6, it is easy to see that the point versions of matrix factorization codes will perform poorly on a machine with a memory

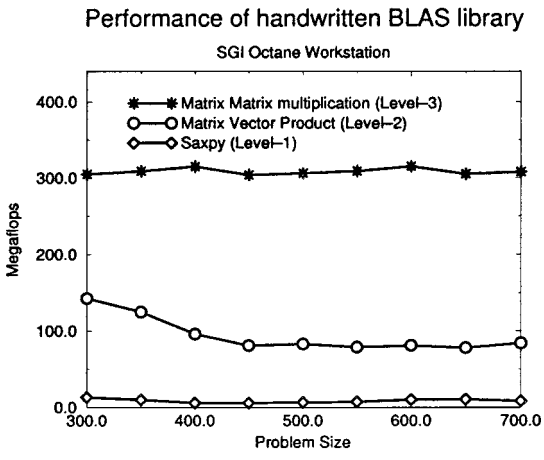


Fig. 6. Performance of computations from levels 1, 2 & 3.

hierarchy. For example, in the kij version of Cholesky factorization shown in Fig. 2, the innermost loop performs a saxpy operation (notice that A(i, k) is invariant in this loop, and its value is used to scale a portion of the kth column of A which is then added to a portion of the ith row of A). Since BLAS-1 operations perform poorly on a memory hierarchy, we would expect that this point version of Cholesky factorization would perform poorly as well. This is in fact the case, as we show in Section 9. Fortunately, it is possible to restructure the computation to expose BLAS-3 operations. To illustrate this, we show that the block algorithm in the LAPACK library can be derived from the point version by program restructuring.

It can be shown that the perfectly nested loop shown in Fig. 7 performs Cholesky factorization, and that the three loops are fully permutable. This perfectly nested version of Cholesky factorization can be generated from the kij-fused version shown in Fig. 2 by repeated application of code-sinking.[16]

The first step in generating the block-*j* version of Cholesky used in the LAPACK library is to stripmine the j loop in blocks of size B, and interchange loops to produce the code shown in Fig. 8.

```
do k = 1, n
  do i = k, n
    do j = k, i
      if (i == k && j == k) A(k,k) = sqrt (A(k,k));
      if (i >  k && j == k) A(i,k) = A(i,k) / A(k,k);
      if (i >  k && j >  k) A(i,j) -= A(i,k) * A(j,k);
```

Fig. 7. A fully permutable loop nest for Cholesky factorization.

```
do js = 0, n/B  //this counts the block columns
  do i = B*js +1, n
    do k = 1, min(i,B*js+B)
      do j = max(B*js +1,k), min(i,B*js+B)
         if (i == k && j == k) A(k,k) = sqrt (A(k,k));
         if (i >  k && j == k) A(i,k) = A(i,k) / A(k,k);
         if (i >  k && j  > k) A(i,j) -= A(i,k) * A(j,k);
```

Fig. 8. Stripmine-and-interchange of code in Fig. 7.

Next, we simplify the loop bounds to get rid of min's and max's by index-set splitting the i and k loops. The index-set of the i loop is split into two ranges $B*js+1$ to $B*js+B$, and $B*js+B+1$ to n. The index set of the k loop is split into the two ranges 1 to $B*js$, and $B*js+1$ to i. Simplifying the predicates then gives us the code shown in Fig. 9. A pictorial representation of this code is shown in Fig. 10.

To get good performance, the matrix multiplications and triangular solve must be performed by calling the appropriate BLAS.

Similar block algorithms can be derived for LU with pivoting and QR factorization. For LU with pivoting, the block algorithm in LAPACK

```
do js = 0, n/B

  //Computation 1: matrix multiplication
   do i= B*js +1, B*js+B
     do k = 1,B*js
       do j = B*js +1,i
         A(i,j) -= A(i,k) * A(j,k);

   //Computation 2: small Cholesky factorization
    do i = B*js +1,B*js+B
      do k = B*js+1,i
        do j = k,i
          if (i == k && j == k) A(k,k) = sqrt (A(k,k));
          if (i >  k && j == k) A(i,k) = A(i,k) / A(k,k);
          if (i >  k && j  > k) A(i,j) -= A(i,k) * A(j,k);

   //Computation 3: matrix multiplication
     do i = B*js+B+1,n
       do k = 1,B*js
         do j = B*js+1,B*js+B
           A(i,j) -= A(i,k) * A(j,k);

   //Computation 4: triangular solve with multiple RHS
     do i = B*js+B+1,n
       do k = B*js+1,B*js+B
         do j = k,B*js+B
            if (j == k) A(i,k) = A(i,k) / A(k,k);
            if (j  > k) A(i,j) -= A(i,k) * A(j,k);
```

Fig. 9. Index-set splitting of code in Fig. 8.

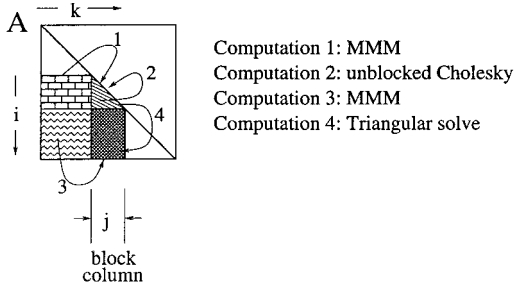


Fig. 10. Pictorial view of code in Fig. 9.

exploits the fact that row permutations commute with updates, while the block QR factorization exploits associativity of matrix products.

The performance of LAPACK factorization codes on the SGI Octane is shown in Fig. 11. Cholesky factorization runs at roughly 250 MFlops while QR and LU with pivoting run at roughly 200 MFlops.

3.3. Discussion

It is seems unlikely that a compiler can mimic the steps outlined earlier for deriving block Cholesky code from point Cholesky. Transforming the kij-fused version of Cholesky factorization into the fully permutable loop nest of Fig. 7 by code sinking is reasonably straight-forward, but it should be noted that there are many ways to apply code sinking to the program in Fig. 2, and each of these produces different perfectly nested loops. For example, in Fig. 7, the square root and scale steps can be done when

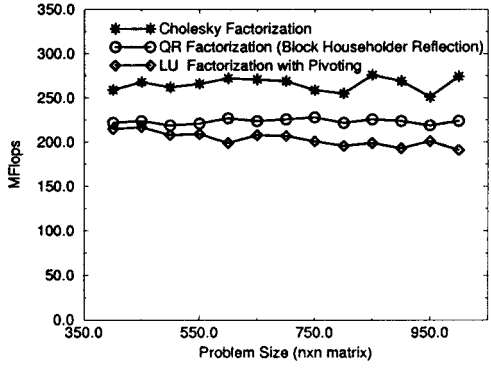


Fig. 11. Performance of Cholesky, QR, and LU factorization codes from LAPACK.

(j == k + 1) without changing the semantics of the program. Moreover, generating the perfectly nested version from other versions of Cholesky factorization is non-trivial. In the kji version for example, the j and i loops must be interchanged, then the two i loops must be fused, after which code sinking can be applied to generate the fully permutable version. It is not clear what would drive a compiler to make these choices.

Once the perfectly nested loop is generated, another sequence of transformations is required to extract the BLAS-3 sub-computations from the fully permutable loop nest. As before, it is not clear how one might automate this sequence of transformations.

The approach of restructuring code to expose BLAS-3 operations and executing these operations using machine-tuned BLAS has been used successfully to produce high-performance library code, but we believe that it is difficult for a compiler to imitate this strategy directly.

4. EXISTING COMPILER APPROACHES

The compiler community has developed many techniques for enhancing locality by restructuring perfectly nested loops. In contrast, much less is known about locality enhancement in imperfectly nested loops.

4.1. Perfectly Nested Loops

The most effective transformation is loop tiling, preceded if necessary by linear loop transformations like permutation and skewing.[16–19] For example, consider a nested loop that adds the elements of a two-dimensional array stored in column-major order. If the code is written so that the array is accessed by row, spatial locality is enhanced by permuting the loops so that the innermost loop walks over the array with unit stride. This example demonstrates the use of linear loop transformations for locality enhancement, but it does not require tiling. In codes like matrix multiplication, locality can improved by moving any one of several loops into the innermost position, as discussed in Section 3.1. Tiling is beneficial for such codes since it gives the effect of interleaving the iterations of these loops, thereby providing most of the benefits of having all these loops in the innermost position. Matrix multiplication therefore demonstrates the need for tiling. Tiling is not always legal, so the most general strategy is to apply linear loop transformations to convert a loop nest into a fully permutable loop nest which can be tiled. A critical parameter in tiling a loop nest is *tile size* which must be chosen so that cache misses are as small as possible during the execution of the tile. Many heuristics for choosing good tile sizes have been developed.[4, 7, 20]

A very sophisticated implementation of these techniques can be found in the SGI MIPSPro compiler for the Octane workstation.[14] This compiler converts *singly nested loop nests* into perfectly nested loop nests using code sinking, and then tiles the resulting perfectly nested loop nest.

Definition 2. A *singly nested loop* (*SNL*) is an imperfectly nested loop in which there is at most one loop nested immediately inside every loop.

The triangular solve code in Fig. 1 and the kij-fused version of Cholesky factorization in Fig. 2 are SNL's while the kij version of the Cholesky factorization code in Fig. 18 is an example of an imperfectly nested loop that is not an SNL. All the BLAS codes are SNL's, so the SGI MIPSPro compiler achieves good performance on these code, as can be seen in Fig. 12. The performance of the compiled code is roughly equal to that of handwritten code for BLAS-1 and BLAS-2; for BLAS-3, the handwritten version is moderately better. The structure of the compiled code for BLAS-3 is identical to that of the handwritten code, but the handwritten version does a better job of choosing block sizes.

4.2. Imperfectly Nested Loops

Matrix factorization codes are imperfectly nested and only the kij-fused version of Cholesky factorization is an SNL.

The MIPSPro compiler attempts to perform loop transformations like fusion and distribution to transform these codes into perfectly nested loops that can be tiled, but it is successful in doing this only for the kij-fused

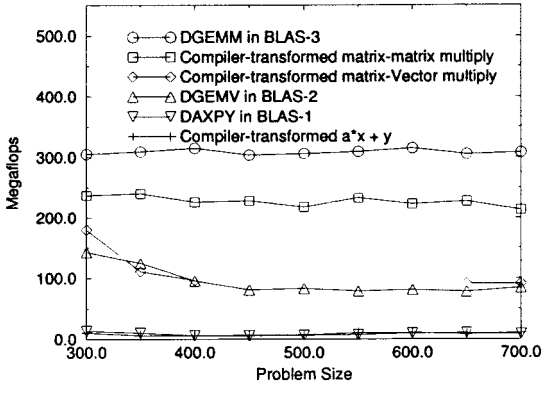


Fig. 12. Performance of handwritten and compiled BLAS.

version of Cholesky factorization. Therefore, the performance of compiled code for matrix factorizations is quite poor, as we discuss in detail in Section 9.

A number of approaches for enhancing locality in imperfectly nested loops have been proposed in the research literature. Carr and Kennedy analyzed block factorizations such as Cholesky and LU, and showed that sequences of loop transformations such as index-set splitting, loop distribution and stripmine-and-interchange can produce blocked codes from unblocked ones.[21] This is like the approach we took in Section 3, but it does not require conversion to a fully permutable, perfectly nested loop nest as an intermediate step since imperfectly nested loop transformations such as index-set splitting are applied directly. However, it is not clear what would drive a compiler to synthesize such sequences of transformations. In addition, their work considered only the kij version of Cholesky factorization; other versions of Cholesky factorization require different sequences of transformations. To block LU factorization with pivoting, they propose to introduce pattern matching into the compiler to permit it to recognize that row permutation commutes with updates. This work was extended by Carr and Lehoucq[22] to QR factorization, although the blocked version they developed is different from the one in LAPACK.

Ramanujam and Schreiber[23] used a combination of code sinking and loop fusion to convert some of the imperfectly nested variants of Cholesky to the fully permutable, perfectly nested intermediate form of Fig. 7. Their strategy works well for the kij and kij-fused versions of Cholesky factorization, but it cannot be applied to other versions because loop fusion is not legal in these codes.

McKinley *et al.*[24] present a cost model for memory accesses and use it to determine the best version of Cholesky factorization, but they do not consider tiling.

More recently, Song and Li developed a technique for tiling imperfectly nested loops that arise in relaxation codes.[25] However, this technique is very specific to relaxation codes, and cannot be used to improve locality in matrix factorization codes for example.

5. DATA-CENTRIC TRANSFORMATION

We now describe *data-shackling*, a locality enhancement technique designed to be applicable to imperfectly nested loops. In data-shackling, the compiler picks an order in which data structure elements should be brought into the cache at runtime, and it restructures code so that statement instances that access a given element are scheduled close together in time. When the transformed program is executed, data structure elements

will be brought into the cache in approximately the order chosen by the compiler. In this way, the locality enhancement problem is converted to a scheduling problem in which the schedule of statement execution is matched to the schedule of data movement chosen by the compiler.

5.1. Data-Shackling

The key concept in data-shackling is the idea of a *data-shackle*.

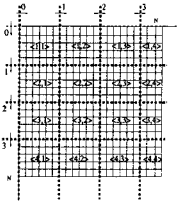
Definition 3. A data-shackle is a specification in three parts.

1. One of the arrays in the program is divided into blocks using sets of parallel, equally spaced cutting planes. Each set of cutting planes is specified by an orientation and a pitch.
2. An order for visiting the blocks of data is specified.
3. One reference to that array is selected for each statement in the program. This reference is called the *data-centric reference* for that statement.

Intuitively, the data-shackle specifies an order in which blocks of the array are touched, and the data-centric reference is used to determine which iterations of each statement get performed when that block of the array is touched—code is generated to perform all iterations of that statement for which the data-centric reference touches data within the current block.

We illustrate this with matrix multiplication. One data-shackle is obtained by dividing C into two-dimensional 25×25 blocks using horizontal and vertical sets of cutting planes, as shown in Fig. 13a. These blocks can be visited in left-to-right, top-to-bottom order. C(i, j) is the only reference to this array in the assignment statement, and it is chosen to be the data-centric reference for that statement.

Figure 13b shows naive code generated by using this data-shackle. There are two outermost loops which enumerate over the blocks of C. For each block, the entire initial loop nest is executed, and two conditionals are inserted at the innermost level to ensure that the data touched by the data-centric reference lies within the current block. This code generation strategy is reminiscent of the *runtime resolution* code generation strategy which is used in compiling shared-memory programs for distributed-memory machines.[26] This code is shown only to illustrate the high-level idea of a data-shackle. In the implementation, standard integer linear programming tools are used to fold the bounds of the data blocks into the inner loop bounds, producing the optimized code shown in Fig. 13c. Notice that



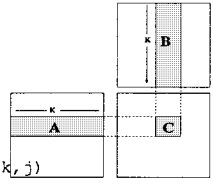
(a) Blocking of C matrix

```
do b1 = 1 .. ⌈(N/25)⌉
  do b2 = 1 .. ⌈(N/25)⌉
  do i = 1 .. N
   do j = 1 .. N
    do k = 1 .. N
     if ((b1-1)*25 < i <= b1*25) &&
        ((b2-1)*25 < j <= b2*25)
          C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

(b) Naive code produced by shackling C

```
do t1 = 1 .. ⌈(N/25)⌉
 do t2 = 1 .. ⌈(N/25)⌉
  do i = (t1-1)*25 +1 ..
min(t1*25,N)
   do j = (t2-1)*25 +1 ..
min(t2*25,N)
    do k = 1 .. N
     C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

(c) Simplified code produced by IP tool



(d) Data touched by code

Fig. 13. Code produced by shackling C in matrix-multiply.

within the context of a single block, iterations are done in the same order as in the original code (these are called *intra-block iterations*), but in the program as a whole, the order in which iterations are performed is different from their order in the source program. Therefore, shackling is not always legal. In Section 6, we show that the determination of whether a data-shackle is legal can be reduced to the standard problem of determining the emptiness of the union of certain polyhedra, a problem for which many algorithms exist.

This shackle does not produce the standard block matrix multiplication code discussed in Section 2.1. For a given block of C, the data-shackle specified above constrains the i and j loop indices, but does not constrain the k index in any way, as can be seen in Fig. 13d. This results in poor locality for the A(i, k) and B(k, j) references. This problem can be addressed by *composing shackles*, as explained in Section 7.

A more complicated example is data-shackling of the kij-fused version of Cholesky factorization. The array A can be blocked into 64×64 blocks in a manner similar to Fig. 13a. When a block is scheduled, all statements that *write* to that block can be executed in program order. In other words, the reference chosen from each statement of the loop nest is the left-hand side reference in that statement. The code obtained after simplification with polyhedral algebra tools is shown in Fig. 14. The reader who wants some insight into the structure of this code should study Fig. 15. Data-shackling regroups the iteration space into four sections. Initially, all updates to the diagonal block from the left are performed (Fig. 15(i)). This is followed by

```
Source code:
do k = 1, n
  S1: A(k,k) = dsqrt (A(k,k))
  do i = k+1, n
    S2: A(i,k) = A(i,k) / A(k,k)
  do i = k+1, n
    do j = k+1, i
      S3: A(i,j) -= A(i,k) * A(j,k)

Shackled code:
do t1 = 1, (n+63)/64
  /* Apply updates from left to diagonal block */
  do t3 = 1, 64*t1-64
    do t6 = 64*t1-63, min(n,64*t1)
      do t7 = t6, min(n,64*t1)
        A(t7,t6) = A(t7,t6) - A(t7,t3) * A(t6,t3)

  /* Cholesky factor diagonal block */
  do t3 = 64*t1-63, min(64*t1,n)
    A(t3,t3) = sqrt(A(t3, t3))
    do t5 = t3+1, min(64*t1,n)
      A(t5,t3) = A(t5,t3)/ A(t3,t3)
    do t6 = t3+1, min(n,64*t1)
      do t7 = t6, min(n,64*t1)
        A(t7,t6) = A(t7,t6) - A(t7,t3) * A(t6,t3)

  /* Apply updates from left to
     off-diagonal block */
  do t2 = t1+1, (n+63)/64
    do t3 = 1, 64*t1-64
      do t6 = 64*t1-63, 64*t1
        do t7 = 64*t2-63, min(n,64*t2)
          A(t7,t6) = A(t7,t6) - A(t7,t3) * A(t6,t3)

  /* Apply internal scale/updates to
     off-diagonal block */
  do t3 = 64*t1-63, 64*t1
    do t5 = 64*t2-63, min(64*t2,n)
      A(t5,t3) = A(t5,t3)/ A(t3,t3)
    do t6 = t3+1, 64*t1
      do t7 = 64*t2-63, min(n,64*t2)
        A(t7,t6) = A(t7,t6) - A(t7,t3) * A(t6,t3)
```

Fig. 14. Data-shackling applied to right-looking Cholesky factorization.

a *small Cholesky factorization*[1] of the diagonal block (Fig. 15(ii)). For each off-diagonal block, updates from the left (Fig. 15(iii)) are followed by interleaved scaling of the columns of the block by the diagonal block, and local updates (Fig. 15(iv)).

As in the case of matrix multiplication, this code is only partially blocked, compared to the block factorization code in the LAPACK library. Although all the writes are performed into a block when that block is

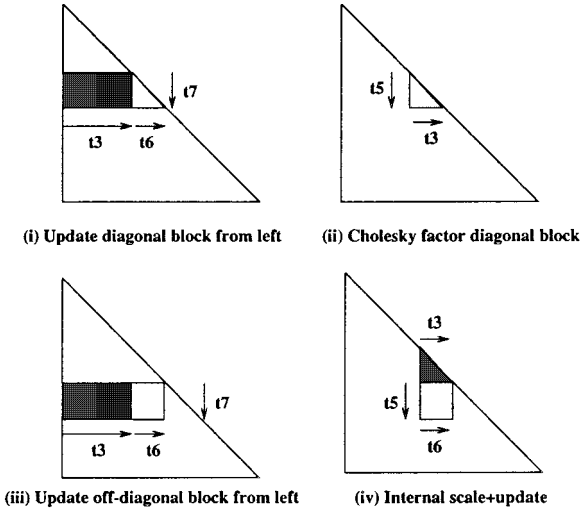


Fig. 15. Pictorial view of code in Fig. 14.

visited, reads are not localized to blocks but are distributed over the entire left portion of the matrix. As with matrix multiplication, this problem is solved by composing shackles.

5.2. Discussion

By shackling a data reference R in a source program statement S, we ensure that the memory access made from that data reference at any point in program execution will be constrained to the "current" data block. Turning this around, we see that when a block becomes current, we perform all instances of statement S for which the reference R accesses data in that block. Therefore, this reference enjoys perfect *self-temporal locality*.[12] Considering all shackled references together, we see that we also have perfect *group-temporal locality* for this set of references; of course, references outside this set may not necessarily enjoy group-temporal locality with respect to this set. As mentioned earlier, we do not mandate any particular order in which the data points within a block are visited. However, if all dimensions of the array are blocked and the block fits in cache (or whatever level of the memory hierarchy is under consideration), we obviously exploit *spatial locality*, regardless of whether the array is stored in column-major or row-major order. An interesting observation is that if stride-1 accesses are mandated for a particular reference, we can simply use cutting planes with unit separation which enumerate the elements of the array in storage order.

Enforcing stride-1 accesses *within* the blocks of a particular blocking can be accomplished by composing shackles as described in Section 7.

The code shown in Fig. 14 can certainly be obtained by a (long) sequence of traditional iteration space transformations like sinking, tiling, index-set splitting, distribution etc. As we discussed in the introduction, it is not clear for imperfectly nested loops in general how a compiler determines which transformations to carry out and in what sequence these transformations should be performed.

6. LEGALITY OF DATA SHACKLING

Since data-shackling reorders statement instances, we must ensure that it does not violate dependences. An instance of a statement S can be identified by a vector $\underline{i}$ which specifies the values of the index variables of the loops surrounding S. The tuple $(S, \underline{i})$ represents instance $\underline{i}$ of statement S. Suppose there is a dependence from $(S1, \underline{i1})$ to $(S2, \underline{i2})$ and suppose that these two instances are executed when blocks b_1 and b_2 are touched respectively. For the data-shackle to be legal, either b_1 must be touched before b_2, or b_1 and b_2 must be identical (note that if b_1 and b_2 are identical, the code generation strategy in Section 5 ensures that the statement instances are executed in original program order). In this case, we say that the data-shackle *respects* that dependence. A data-shackle is legal if it respects all dependences in the program. Since our techniques apply to imperfectly nested loops like Cholesky factorization, it is not possible to use dependence abstractions like distance and direction to verify legality. Instead, we solve integer linear programming problems, as we discuss in this section.

6.1. An Example of Testing Legality

To understand the general algorithm, it is useful to consider first a simple example distilled from the Cholesky factorization code of Fig. 14. In the source code, there is a flow dependence from the assignment of A(k, k) in S1 to the use of A(k, k) in S2. We must ensure that this dependence is respected in the shackled code shown in Fig. 14.

We first write down a set of integer inequalities that represents the existence of a flow dependence between an instance of S1 and an instance of S2. Let S1 write to an array location in iteration k_w of the k loop, and let S2 read from that location in iteration (k_r, i_r) of the k and i loops. A flow dependence exists if the following linear inequalities have an integer solution:[16]

$$\begin{cases} k_r = k_w & \text{(same location)} \\ n \geqslant k_w \geqslant 1 & \text{(loop bounds)} \\ n \geqslant k_r \geqslant 1 & \text{(loop bounds)} \\ n \geqslant i_r \geqslant k_r + 1 & \text{(loop bounds)} \\ k_r \geqslant k_w & \text{(read after write)} \end{cases} \tag{6.1}$$

Next, we assume that the instance of **S1** is performed when a block (b_{11}, b_{12}) is scheduled, and the instance of **S2** is done when block (b_{21}, b_{22}) is scheduled. Finally, we add a condition that represents the condition that the dependence is violated in the transformed code. In other words, we formulate the condition that block (b_{21}, b_{22}) is touched strictly before block (b_{11}, b_{12}). These conditions are represented as:

$$\begin{aligned} &\text{Writing iteration done in } (b_{11}, b_{12}) \\ &b_{11} * 25 - 24 \leqslant k_w \leqslant b_{11} * 25 \\ &b_{12} * 25 - 24 \leqslant k_w \leqslant b_{12} * 25 \\ &\text{Reading iteration done in } (b_{21}, b_{22}) \\ &b_{21} * 25 - 24 \leqslant k_r \leqslant b_{21} * 25 \\ &b_{22} * 25 - 24 \leqslant i_r \leqslant b_{22} * 25 \\ &\text{Blocks visited in bad order} \\ &(b_{21} < b_{11}) \vee ((b_{11} = b_{21}) \wedge (b_{22} < b_{12})) \end{aligned} \tag{6.2}$$

If the conjunction of the two sets of conditions (6.1) and (6.2) has an integer solution, it means that there is a dependence, and that dependent instances are performed in the wrong order. If so, the data-shackle violates the dependence and is not legal. This problem can be viewed geometrically as asking whether the union of certain polyhedra contains an integer point. This problem can be solved using standard polyhedral algebra tools.

Such a test can be performed for each dependence in the program. If no dependences are violated, the data-shackle is legal.

6.2. General View of Legal Data-Shackles

The formulation of the general problem of testing for legality of a data-shackle becomes simpler if we first generalize the notion of blocking data. A data blocking such as the one shown in Fig. 13a can be viewed simply as a map that assigns co-ordinates in some new space to every data element in the array. For example, if the block size in this figure is 25×25, array element (a_1, a_2) is mapped to the co-ordinate $(((a_1 - 1) \text{ div } 25) + 1,$

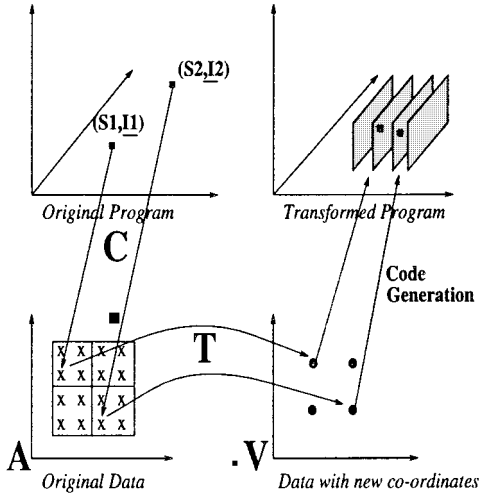


Fig. 16. Testing for legality.

$((a_2 - 1) \text{ div } 25) + 1)$ in a new two-dimensional space. Note that this map is not one-to-one. The bottom part of Fig. 16 shows such a map pictorially. The new space is totally ordered under lexicographic ordering. The data shackle can be viewed as traversing the remapped data in lexicographic order in the new co-ordinates; when it visits a point in the new space, all statement instances mapped to that point are performed.

Therefore, a data-shackle can be viewed as a function $\mathbf{M}$ that maps statement instances to a totally ordered set $(\mathsf{V}, \prec)$. For the blocking shown in Fig. 16, $\mathsf{C}: (\mathsf{S}, \mathsf{I}) \rightarrow A$ maps statement instances to elements of array A through data-centric references, and $\mathsf{T}: \mathsf{A} \rightarrow \mathsf{V}$ maps array elements to block co-ordinates. Concisely, $\mathbf{M} = \mathsf{T} \circ \mathsf{C}$.

Given a function $\mathbf{M}: (\mathsf{S}, \mathsf{I}) \rightarrow (\mathsf{V}, \prec)$, the transformed code is obtained by traversing V in increasing order, and for each element $\mathsf{v} \in \mathsf{V}$, executing the statement instances $\mathbf{M}^{-1}(\mathsf{v})$ in program order in the original program.

Theorem 1. A map $\mathbf{M}: (\mathsf{S}, \mathsf{I}) \rightarrow (\mathsf{V}, \prec)$ generates legal code if the following condition is satisfied for every pair of dependent statements $\mathsf{S1}$ and $\mathsf{S2}$.

- Introduce vectors of unknowns $\underline{\mathsf{i1}}$ and $\underline{\mathsf{i2}}$ that represent instances of dependent statements $\mathsf{S1}$ and $\mathsf{S2}$ respectively.
- Formulate the inequalities that must be satisfied for a dependence to exist from instance $\underline{\mathsf{i1}}$ of statement $\mathsf{S1}$ to instance $\underline{\mathsf{i2}}$ of statement $\mathsf{S2}$. This is standard.[16]

- Formulate the predicate $\mathbf{M}(S2, \underline{i2}) \prec \mathbf{M}(S1, \underline{i1})$.
- The conjunction of these conditions does not have an integer solution.

Proof. Obvious, hence omitted. ▮

6.3. Discussion

Viewing blocking as a remapping of data co-ordinates simplifies the development of the legality test. This remapping is merely an abstract mathematical device to enforce a desired order of traversal through the array, and the physical array itself is not necessarily reshaped. For example, in the blocked matrix multiplication code in Fig. 13, array C need not be laid out in block order to obtain the benefits of blocking this array. This is similar to the situation in BLAS/LAPACK where it is assumed that the FORTRAN column-major order is used to store arrays. Of course, nothing prevents us from reshaping the physical data array if the cost of converting back and forth from a standard representation is tolerable. Automatic data reshaping has been explored by other researchers.[27, 28]

7. PRODUCTS OF SHACKLES

We now show that there is a natural notion of taking the Cartesian product of a set of shackles, and that this notion is the data-centric equivalent of loop nesting.

The motivation for this operation comes from the matrix multiplication code of Fig. 13c, in which an entire block row of A is multiplied with a block of column of B to produce a block of C. The order in which the iterations of this computation are done is left unspecified by the data-shackle (the default code generation scheme of Section 5 will execute these iterations in original program order). Note that the shackle on reference C(I, J) constrains both I and J, but leaves K unconstrained; therefore, the references A(I, K) and B(K, J) can touch an unbounded amount of data in arbitrary ways during the execution of the iterations shackled to a block of C(I, J)). Instead of C, we can block A or B, but this still results its unconstrained references to the other two arrays. To get BLAS-style blocked matrix multiplication, we need to block all three arrays. We show that this effect can be achieved by taking Cartesian products.

Informally, the notion of taking the Cartesian product of two shackles can be viewed as follows. The first shackle partitions the statement instances of the original program, and imposes an order on these partitions. However, it does not mandate an order in which the statement instances in

a given partition should be performed. The second shackle refines each of these partitions separately into smaller, ordered partitions, without reordering statement instances in different partitions obtained from the first shackle. In other words, if two statement instances are ordered by the first shackle, they are not reordered by the second shackle. The notion of a binary Cartesian product can be extended the usual way to an n-ary Cartesian product; each extra factor in the Cartesian product gives us finer control over the granularity of data accesses.

A formal definition of the Cartesian product of data-shackles is the following. Recall from the discussion in Section 6 that a data-shackle for a program P can be viewed as a map $\mathbf{M}$: $(\mathsf{S}, \mathsf{I}) \rightarrow \mathsf{V}$, whose domain is the set of statement instances and whose range is a totally ordered set.

Definition 4. For any program P, let

$$\begin{cases} \mathbf{M}_1 : (\mathsf{S}, \mathsf{I}) \rightarrow \mathsf{V}_1 \\ \mathbf{M}_2 : (\mathsf{S}, \mathsf{I}) \rightarrow \mathsf{V}_2 \end{cases}$$

be two data-shackles. The Cartesian product $\mathbf{M}_1 \times \mathbf{M}_2$ of these shackles is defined as the map whose domain as the set of statement instances, whose range is the Cartesian product $\mathsf{V}_1 \times \mathsf{V}_2$ and whose values are defined as follows: for any statement instance $(\mathsf{S}, \underline{\mathsf{i}})$, $(\mathbf{M}_1 \times \mathbf{M}_2)(\mathsf{S}, \underline{\mathsf{i}}) = \langle \mathbf{M}_1(\mathsf{S}, \underline{\mathsf{i}}), \mathbf{M}_2(\mathsf{S}, \underline{\mathsf{i}}) \rangle$.

The product domain $\mathsf{V}_1 \times \mathsf{V}_2$ of two totally ordered sets is itself a totally ordered set under standard lexicographic order. Therefore, the code generation strategy and associated legality condition are identical to those in Section 6. It is easy to see that for each point $\underline{v_1} \times \underline{v_2}$ in the product domain $\mathsf{V}_1 \times \mathsf{V}_2$, we perform the statement instances in the set $(\mathbf{M}_1 \times \mathbf{M}_2)^{-1}(\underline{v_1}, \underline{v_2}) = \mathbf{M}_1{}^{-1}(\underline{v_1}) \cap \mathbf{M}_2{}^{-1}(\underline{v_2})$.

In the implementation, each term in an n-ary Cartesian product contributes a guard around each statement. The conjunction of these guards determines which statement instances are performed at each step of execution. Therefore, these guards still consist of conjuncts of affine constraints. As with single data-shackles, the guards can be simplified using any polyhedral algebra tool.

Note that the product of two shackles is always legal if the two shackles are legal by themselves. However, a product $\mathbf{M}_1 \times \mathbf{M}_2$ can be legal even if $\mathbf{M}_2$ by itself is illegal. This is analogous to the situation in loop nests where a loop nest may be legal even if there is an inner loop that cannot be moved to the outermost position; the outer loop in the loop nest "carries" the dependence that causes difficulty for the inner loop.

7.1. Examples

In matrix multiplication, it is easy to see that shackling any of the three references (C(I, J), A(I, K), B(K, J)) to the appropriate blocked array is legal. Therefore, all Cartesian products of these shackles are also legal. The Cartesian product $\mathbf{M}_C \times \mathbf{M}_A$ of the C and A shackles produces the code in Fig. 5. It is interesting to note that further shackling with the B shackle (that is the product $\mathbf{M}_C \times \mathbf{M}_A \times \mathbf{M}_B$) does not change the code that is produced. This is because shackling C(I, J) to the blocks of C and shackling A(I, K) to blocks of A imposes constraints on the reference B(K, J) as well. A similar effect can be achieved by shackling the references C(I, J) and B(K, J), or A(I, K) and B(K, J).

A more interesting example is the Cholesky code. In Fig. 2, it is easy to verify that there are six ways to shackle references in the source program to blocks of the matrix (choosing A(K, K) from statement S1, either A(I, K) or A(K, K) from statement S2 and either A(I, J), A(I, K) or A(J, K) from statement S3). Of these, only two are legal: choosing A(K, K) from S1, A(I, K) from S2 and A(I, J) from S3, or choosing A(K, K) from S1, A(K, K) from S2 and A(I, K) from S3. The first shackle chooses references that write to the block, while the second shackle chooses references that read from the block. Since both these shackles are legal, their Cartesian product (in either order) is legal. It can be shown that one order gives a fully-blocked left-looking Cholesky, similar to the blocked Cholesky algorithm in LAPACK, while the other order gives a fully-blocked right-looking Cholesky. The left-looking code produced by shackling is shown in Fig. 27.

7.2. Discussion

Taking the Cartesian product of data-shackles gives us finer control over data accesses in the blocked code. As discussed earlier, shackling just one reference in matrix multiplication (say C(I, J)) does not constrain all the data accesses. On the other hand, shackling all three references in this code is over-kill since shackling any two references constraints the third automatically. Taking a larger Cartesian product than is necessary does not affect the correctness of the code, but it introduces unnecessary loops into the resulting code which must be optimized away by the code generation process to get good code. In Section 8, we explain how our implementation of data-shackling determines when to stop composing data-shackles.

8. HEURISTICS USED IN IMPLEMENTATION

Sections 5–7 described the mechanisms that underlie data shackling. An implementation of data-shackling must make certain policy decisions

as well. One of the authors (Kodukula) has implemented data-shackling in the SGI MIPSPro compiler. In this section, we describe the policies implemented in this compiler. In a production compiler, the time taken to compile programs must be kept small, so these policies are based on heuristic choices that are simple to implement. Their effectiveness for our workload is discussed in Section 9. We believe that our heuristics are reasonable, although it is certainly easy to invent other plausible ones.

8.1. Policy Decisions

An implementation of shackling must address the following questions.

1. What is the program unit to which data-shackling is applied?
2. How are the parameters for a single data-shackle chosen?
   (a) Which array is shackled?
   (b) What is the orientation of the cutting planes?
   (c) What is the order of traversal of blocks?
   (d) What is the separation of cutting planes (block sizes)?
   (e) How are data-centric references chosen?
3. How many shackles are composed?

One approach to answering many of these questions is to treat them as classical optimization problems, and try to find optimal solutions in the context of an accurate model. However, this approach is impractical in a production compiler, so we developed heuristics instead.

8.2. Program Unit for Shackling

Shackling is applied to one imperfectly nested loop at a time. Shackling is essentially statement scheduling, so it can be applied in principle to multiple imperfectly nested loops or even to entire programs, but we do not have enough experience at this point to do this effectively.

8.3. Determining the Parameters of a Single Shackle

We now describe the decisions that must be made to determine a single data-shackle.

Picking an array for the shackle: Arrays are ordered by the following criteria.

```
do i = 1, n
  do j = 1, m
    S1: s(i) = A(i,2*j+1)
    S2: r = 9.1 + j
    S3: A(i,2*j) += s(i)*s(i) + r
    S4: A(i,2*j) += r + B(i,j)
```

Fig. 17. Choosing an array for shackling.

- What is the largest row rank of a data access matrix[4] of an unconstrained reference to the array?
- How many unconstrained references of this rank are there?
- Has the array already been used in a shackle?

If two or more arrays are tied according to one criterion, we attempt to break the tie using succeeding criteria. If there are multiple choices at the end of this process, the tie is broken arbitrarily. The rationale for this heuristic is that an array to which there are multiple unconstrained references of large rank is likely to be a major participant in cache traffic.

For example, in Fig. 17, array A is given highest priority for shackling because there are two unconstrained references of row-rank two to this array. Note that array B has only one reference of rank two, while array s has three references, all of rank one.

Orientation of cutting planes: Cutting plane orientations are always chosen to be parallel to the data co-ordinate axes. Dongarra and Schreiber have explored the use of *skewed* blocks for locality enhancement,[7] but skewed blocks are likely to produce variable trip-count inner loops which are detrimental to subsequent phases in the compiler such as software pipelining (the MIPSPro compiler does not use loop skewing for the same reason).

Order of traversal of blocks: An n-dimensional array is blocked by choosing n sets of cutting planes; for example, a two-dimensional array is blocked along both rows and columns. The order of traversal of blocks is chosen to be a lexicographic order on block co-ordinates. For a two-dimensional array, the blocks are visited from left to right, and within a given block column, from top to bottom. If this order is not legal, the compiler tries a right to left order, and also a bottom to top order. If none of these four orders of traversal is legal, the compiler tries to block only

[4] If all array access functions are linear functions of loop variables (if the functions are affine, the constant terms can be dropped), an array access function can be written as $F * \underline{I}$ where F is the **data access matrix**[9] and $\underline{I}$ is the vector of iteration space variables of loops surrounding this data reference.

rows or only columns. If that fails as well, the imperfectly nested loop is not shackled.

Choice of data-centric references: Picking a data-centric reference for each statement is a two step process. In the first step, all candidate data-centric references for a statement are determined. The second step is simply an exhaustive enumeration of all candidate data-centric references for each statement, searching for a legal shackle. We focus on the first step in the following discussion.

After an array has been picked for a shackle, data-centric references to this array must be selected for each statement. At the one extreme, candidate references for a statement could be limited to references to the array that actually occur in that statement. This causes difficulties in programs like the one in Fig. 17 because statement S2 does not contain a reference to array A which is the array chosen for shackling. At the other extreme, all references to the shackled array in the entire imperfectly nested loop can be candidate references for every statement, but this may result in a combinatorial explosion in the number of possibilities that must be considered.

Our implementation chooses an intermediate position which is easy to understand by considering the program in Fig. 17. There is a flow dependence from statement S2 to S3 because S2 writes to the scalar variable r while S3 reads from it. This dependence may not be respected if the data-centric references chosen for the two statements are different (for example, if the data-centric reference for S2 is A(i, 2j + 1) and the data-centric reference for S3 is A(i, 2j)). Therefore, candidate data-centric references for S2 should be candidates for S4 and vice versa.

Our implementation therefore divides statements into equivalence classes such that two statements are in the same equivalence class if and only if there is a dependence between them that is induced by a scalar or an array that will not be shackled. For example, if there are dependences from S1 to S2 and from S1 to S3, all three statements are placed in the same equivalence class even if there is no dependence from S2 to S3. The candidate references for a statement are *all* references to the shackled array that occur in the statements in its equivalence class. For the program in Fig. 17 for example, all statements will be placed in the same equivalence class, so all array references in the loop will be candidate references for all statements.

Our implementation then tries each candidate reference for each statement, and checks if the resulting program is legal. If legality is violated because of a dependence due to a scalar, our implementation performs scalar expansion to eliminate the problem. Array expansion of low-dimensional array that cause this problem is possible in principle, but we have not implemented this.

Finally, we mention that all assignment statements within a conditional statement are placed in the same equivalence class because they are all control dependent on the predicate.

Block Size Determination: To determine block sizes, we used a simplified version of the algorithm used in the MIPSPro compiler for determining *tile sizes* when it tiles perfectly nested loops. This algorithm estimates the amount of data touched in computing a tile (this is called the *footprint* of the tile), and chooses a tile size such that this footprint is a certain fraction of the cache size called the *effective cache size* (between 5–10%). It might appear that cache misses are minimized when the effective cache size is equal to the cache size, but experience has shown that the use of a smaller effective cache size reduces conflict misses without much impact on capacity misses.

We adapted this model to shackling by computing the block size at which the footprint of the intra-block iterations of the shackled code was equal to the effective cache size. The computation of this footprint was done as follows.

In the first step, a set of references with large contributions to the footprint are identified in each statement. For statements that are not most deeply nested in the imperfectly nested loop, this set is defined to be empty. For statements that are most deeply nested, this set is defined to be all the references from that statement with the highest (row)-ranked access matrices. For example, in matrix multiplication, $\mathsf{A}(\mathsf{i}, \mathsf{k})$, $\mathsf{B}(\mathsf{k}, \mathsf{j})$ and $\mathsf{C}(\mathsf{i}, \mathsf{j})$ all correspond to access matrices of row-rank 2, and are all chosen in this step. In Cholesky factorization, the update statement is most deeply nested, and the three references chosen from it are $\mathsf{A}(\mathsf{i}, \mathsf{j})$, $\mathsf{A}(\mathsf{i}, \mathsf{k})$, and $\mathsf{A}(\mathsf{j}, \mathsf{k})$.

The second step performs the following computations for each statement. The references chosen in the previous step are partitioned into groups—two references to the same array fall into the same group if their access matrices have the same linear part, but possibly different affine parts. References belonging to two different arrays always fall into different groups. For example, in matrix multiplication, $\mathsf{A}(\mathsf{i}, \mathsf{k})$, $\mathsf{B}(\mathsf{k}, \mathsf{j})$, and $\mathsf{C}(\mathsf{i}, \mathsf{j})$ all fall into different groups. Similarly, in Cholesky factorization, $\mathsf{A}(\mathsf{i}, \mathsf{j})$, $\mathsf{A}(\mathsf{i}, \mathsf{k})$ and $\mathsf{A}(\mathsf{j}, \mathsf{k})$ fall into different groups. On the other hand, two references of the form $\mathsf{A}(\mathsf{i}, \mathsf{j})$ and $\mathsf{A}(\mathsf{i}+1, \mathsf{j})$ will be assigned to the same group. The assumption is that all references assigned to the same group enjoy perfect reuse, while references assigned to different groups enjoy no reuse. Finally, two groups from two different statements referring to the same array are merged if the references in the two groups have the same linear part, *and* the two statements under consideration have identical data-centric references. The assumption is that if the data-centric references for the two statements are identical, instances of the two statements that

touch the same data will be scheduled close enough together that they will enjoy perfect reuse. If E represents the effective size of the cache and g represents the number of groups, then each group is allowed to have a footprint as large as E/g.

The last step involves computing the footprint of every group for a single instance of a composite shackle. Two simplifications are applied to this computation

(i) the footprint of a group is approximated by the footprint of a *single* reference picked at random from the group, and

(ii) it is assumed that the footprint of the group is identical for all instances of the data loops introduced by the shackle.

The first simplification is justified by the assumption that all references in a group enjoy perfect reuse, and the second simplification is justified since boundary effects are not significant when array sizes and loop bounds are large. The reference picked for each group is called the *representative reference.*

Evaluating the footprint of a single reference for the intrablock iterations of shackled code is straightforward, and variations of this problem have been addressed in the literature.[29] A single instance of the composite shackle is completely specified by a specific set of values for the block coordinates for each level of a composite shackle—for the sake of simplicity, each of the block coordinates can be assumed to be 0. In addition, we only consider square blocks, so for every array a, a single unknown parameter B_a denotes the block size for that array. A system of linear integer equations expressing the localization constraints corresponding to the composite shackle is assembled for the statement containing the *representative reference.* The number of distinct elements touched by the representative reference under this system of equations is multiplied by the size of each element to yield a polynomial in a single parameter for the footprint for this reference. Determining the number of distinct elements touched by the representative reference is thus reduced to the problem of counting the number of integers inside a convex polyhedron. In our current implementation, this is estimated by counting the number of integer solutions inside the *bounding box* of the convex polyhedron. More sophisticated approaches such as Ehrhart Polynomials[30] can potentially be used to obtain better solutions in practice, but it is not clear whether this improvement in accuracy leads to better overall performance.

We implemented shackling only to improve performance of the L2 cache; the miss latency for the L1 cache is small enough that we decided

not to shackle for the L1 cache. Therefore, block sizes were determined using the size of the L2 cache.

8.4. How Many Shackles are Composed?

Composing data-shackles provides finer control over data accesses in the blocked code. As discussed earlier, shackling just one reference in matrix multiplication (say C(i, j)) does not constrain all the data accesses. On the other hand, shackling all three references in this code is over-kill since shackling any two references constraints the third automatically. Applying too many levels of composition does not affect the correctness of the code, but it introduces unnecessary loops into the resulting code which must be optimized away by the code generation process to get good code. The following obvious result is useful to determine how far to carry the process of taking Cartesian products.

Theorem 2. For a given statement S, let $F_1,..., F_n$ be the access matrices for the shackled data references in this statement. Let F_{n+1} be the access matrix for an unshackled reference in S. Assume that the data accessed by the shackled references are bounded by block size parameters. Then the data accessed by F_{n+1} is bounded by block size parameters iff every row of F_{n+1} is spanned by the rows of $F_1,..., F_n$.

Stronger versions of this result can be proved, but it sufficed for purposes of the implementation. For example, the access matrix for the reference C(i, j) is $\left[\begin{smallmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{smallmatrix}\right]$. Shackling this reference does not bound the data accessed by row [0 0 1] of the access matrix $\left[\begin{smallmatrix}0 & 0 & 1\\ 0 & 1 & 0\end{smallmatrix}\right]$ of reference B(k, j). However, taking the Cartesian product of this shackle with the shackle obtained from A(i, k) constrains the data accessed by B(k, j), because all rows of the corresponding access matrix are spanned by the set of rows from the access matrices of C(i, j) and A(i, k). Composition is applied if even a single assignment statement from the loop nest under consideration stands to benefit as a result.

8.5. Discussion

Once all shackles have been determined, localization constraints must be folded into loop bounds (for example, we must generate the code shown in Fig. 13c rather than the code in Fig. 13b). This code generation step has been implemented in polyhedral algebra tools like PIP and Omega. Our implementation uses a simple version of these techniques to generate the shackled code quickly. The interested reader is referred to Kodukula's thesis.[31]

9. EXPERIMENTS

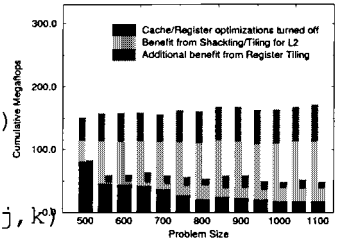
We present experimental results showing the performance of different versions of Cholesky, LU, and QR factorizations. The performance of compiler-generated BLAS codes was shown in Fig. 12.

9.1. Cholesky Factorization

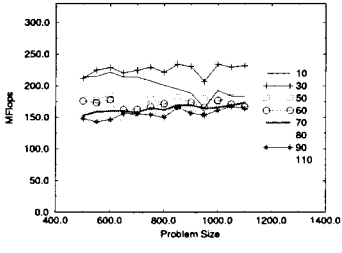
Like matrix multiplication, Cholesky factorization has three nested loops, but these loops are imperfectly nested. All six permutations of these three loops are legal and one of these permutations comes into two versions, giving a total of seven versions of the Cholesky program. Figures 18–24 show pseudo-code for these versions. Ideally, a restructuring compiler would be able to generate the best code for Cholesky factorization from any of these versions of Cholesky factorization, just as many state-of-the-art restructuring compilers do not care which one of the six permutations of matrix multiplication is given as input. As we show below, the performance of code generated by the control-centric approach implemented in the MIPSPro compiler depends quite critically on which version of Cholesky is given as input. In principle, the data-centric approach does not care which version of Cholesky is given to it. However, since our

```
do k = 1, n
  A(k,k) = dsqrt (A(k,k))
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
  do i = k+1, n
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
```

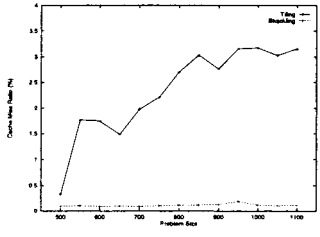
(a) Source code



(b) Performance of tiled and shackled codes



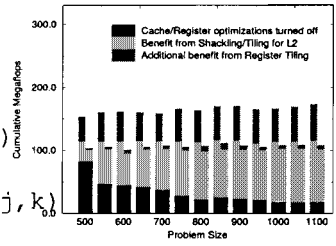
(c) Effect of varying block size



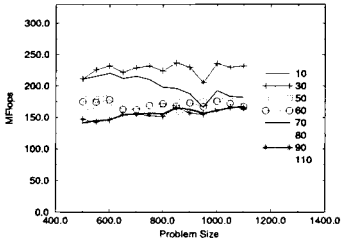
(d) L2 miss ratios for tiled and shackled codes

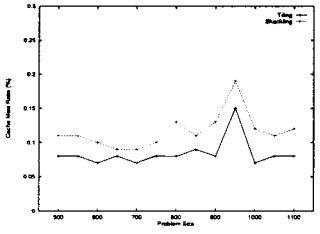
Fig. 18. Performance of Cholesky factorization: kij version.

```
do k = 1, n
  A(k,k) = dsqrt (A(k,k))
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
```



(a) Source code (b) Performance of tiled and shackled codes

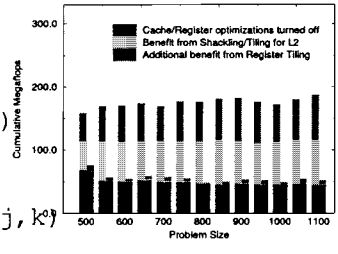




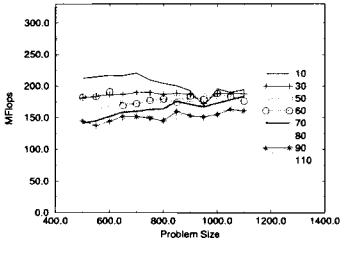
(c) Effect of varying block size (d) L2 miss ratios for tiled and shackled codes

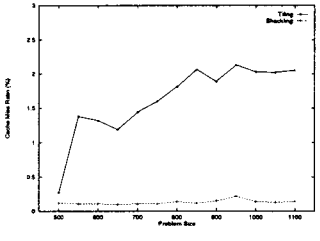
Fig. 19. Performance of Cholesky factorization: kij-fused version.

```
do k = 1, n
  A(k,k) = dsqrt (A(k,k))
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
  do j = k+1, n
    do i = j, n
      A(i,j) -= A(i,k) * A(j,k)
```



(a) Source code (b) Performance of tiled and shackled codes



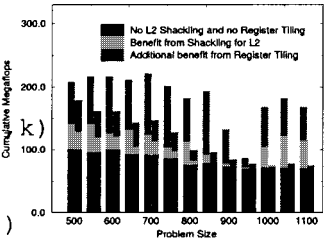


(c) Effect of varying block size (d) L2 miss ratios for tiled and shackled codes

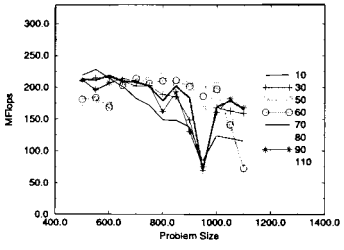
Fig. 20. Performance of Cholesky factorization: kji version.

```
do j = 1, n
  do i = j, n
    do k = 1, j-1
      A(i,j) -= A(i,k) * A(j,k)
  A(j,j) = dsqrt (A(j,j))
  do i = j+1, n
    A(i,j) = A(i,j) / A(j,j)
```

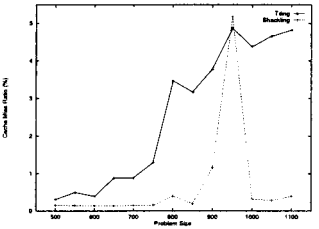
(a) Source code



(b) Performance of tiled and shackled codes



(c) Effect of varying block size

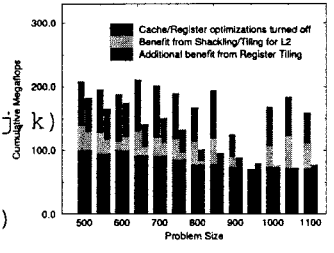


(d) L2 miss ratios for tiled and shackled codes

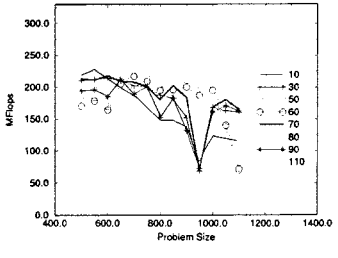
Fig. 21. Performance of Cholesky factorization: jik version.

```
do j = 1, n
  do k = 1, j-1
    do i = j, n
      A(i,j) -= A(i,k) * A(j,k)
  A(j,j) = dsqrt (A(j,j))
  do i = j+1, n
    A(i,j) = A(i,j) / A(j,j)
```

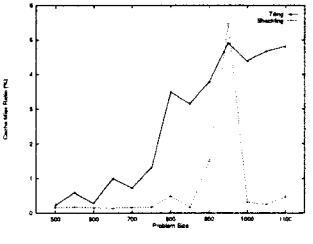
(a) Source code



(b) Performance of tiled and shackled codes



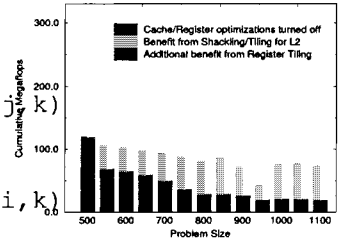
(c) Effect of varying block size



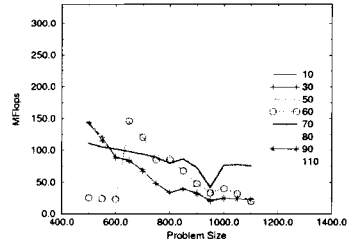
(d) L2 miss ratios for tiled and shackled code

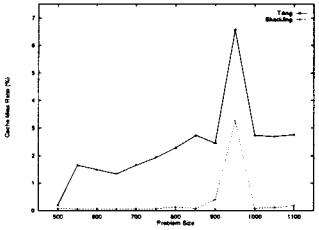
Fig. 22. Performance of Cholesky factorization: jki version.

```
do i = 1, n
  do j = 1, i-1
    do k = 1, j-1
      A(i,j) -= A(i,k) * A(j,k)
    A(i,j) = A(i,j)/A(j,j)
  do k = 1, i-1
    A(i,i) -= A(i,k) * A(i,k)
  A(i,i) = dsqrt (A(i,i))
```



(a) Source code (b) Performance of tiled and shackled codes

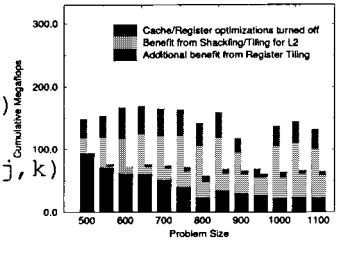




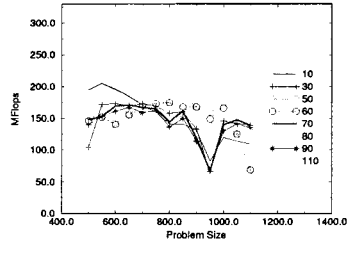
(c) Effect of varying block size (d) L2 miss ratios for tiled and shackled codes

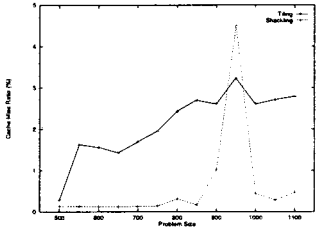
Fig. 23. Performance of Cholesky factorization: ijk version.

```
do i = 1, n
  do k = 1, i-1
    A(i,k) = A(i,k) / A(k,k)
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
  A(i,i) = dsqrt (A(i,i))
```



(a) Source code (b) Performance of tiled and shackled codes





(c) Effect of varying block size (d) L2 miss ratios for tiled and shackled codes

Fig. 24. Cholesky factorization: ikj version.

implementation of the data-centric approach performs intra-block computations in the same order as in the input program, the performance of the shackled code does depend on which version is given as input although as we show below, the variation is less than it is for control-centric transformations.

As discussed in Section 2, the LAPACK version of Cholesky factorization runs at 260 MFlops for matrix sizes between 400 to 1200.

We implemented shackling only to improve performance of the L2 cache; the miss latency for the L1 cache is small enough that we decided not to shackle for the L1 cache. The shackled code produced by the compiler was generated by composing two shackles. In both shackles, the array was divided into rectangular blocks (the compiler heuristic chose 70×70 blocks), and these blocks were visited in left-to-right, top-to-bottom order. In the outer shackle, the compiler chose the left-hand side reference from each assignment statement for shackling, while in the inner shackle, the compiler selected a reference from the right-hand side of each statement: A(k, k) for the square root statement, and A(i, k) for the scale and update statements. The same shackle was used for all other versions of Cholesky factorization as well.

The shackled code performs better than the tiled code in both versions of kij Cholesky, as shown in Figs. 18 and 19. Figure 18d shows that the miss ratio for the tiled code increases rapidly with array size, showing that tiling in the SGI compiler is not effective; in contrast, the miss ratio of the shackled code is almost independent of array size. The kij-fused version in Fig. 19 is an SNL, so the SGI compiler is more successful in enhancing locality in this version. These figures also show the relative contributions of shackling (for the L2 cache) and of register tiling to overall performance. With neither of these locality optimizations, the performance of the baseline code drops to about 10–20 MFlops! Register tiling eliminates many loads and stores of array locations and boosts the performance of the shackled code from 110 MFlops to 175 MFlops. This is consistent with other reports in the literature about the importance of register tiling.[5]

Figures 18c and 19c show the effect of varying the block size in the shackled code. It can be seen the optimal block size is 30×30 rather than the 70×70 chosen by the compiler. With this block size, the performance of the shackled code is boosted to 240 Mflops which is very close to LAPACK performance. These figures suggest that the heuristic for choosing block sizes needs to be improved.

Permuting the two update loops in the kij version gives the kji version shown in Fig. 20. This version is not an SNL, so tiling is not effective. Fusing the scale loop with the outer update loop is illegal. The only way to get an SNL is to interchange the two update loops and then fuse the

new outer update loop with the scale loop, generating the code of Fig. 19, but this is too complicated for the MIPSPro compiler's imperfectly nested loop transformation heuristics to reason about. Therefore, cache tiling has little benefit as is evident in Fig. 20b. The performance of the baseline code (no cache or register tiling) is modestly better than that of the baseline kij versions because of better spatial locality in the update loops. This also explains why the shackled code performs a little better than the shackled code from the kij version.

Figure 21 shows the performance of a left-looking version of Cholesky factorization. The loop nest is not an SNL, but the computational work in the update loops is essentially a matrix-vector product which is performed by the MIPSPro compiler by accumulating the updates to A(i, j) in a register. The shackled version exploits reuse at all loop levels and outperforms the tiled code substantially except when the array size is around 950. This sudden drop in performance is caused by conflict misses. Figure 21c shows that choosing the block size adaptively to reduce conflict misses is one solution. However, the current implementation of shackling does not choose block sizes adaptively. In the jik version, all the updates to an element of the current column are performed before succeeding elements are updated. Permuting the i and k loops gives the jki version. The MIPSPro compiler interchanges the update loops in the jki back to the jik version, so the performance of the baseline and tiled versions is identical to the performance of the jik versions. There is no difference in the performance of the shackled code either.

Finally, Figs. 23 and 24 show the performance of row-Cholesky versions. The ijk version performs inner-products, so it is also known as *ddot Cholesky* while the ikj version is rich in saxpy operations. Figure 23b shows that while the shackled code outperforms the tiled code, it performs poorly compared to LAPACK code. To understand why, note that Fig. 23d shows that the L2 cache miss ratios are similar to those of Fig. 18, but Fig. 24b shows that register tiling is not effective in this code. The shackled code for the ikj version performs better, but it too appears exploits register tiling to a limited extent. Improving the performance of the ikj and ijk versions requires closer integration of shackling with register tiling.

9.2. LU Factorization

Figure 25 compares the performance of shackling and tiling for LU factorization with pivoting. As mentioned before, the entire loop nest is not an SNL, and therefore cannot be tiled. However, the update loop nest can be tiled, and this has a small benefit because it permits spatial locality to be exploited.

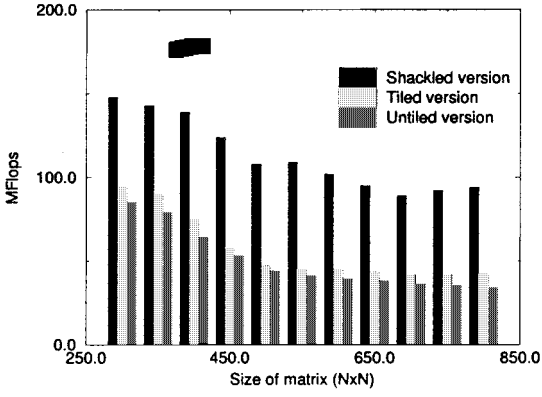


Fig. 25. LU factorization with partial pivoting.

Using simple data-flow analysis, it can be determined that for the LU factorization code in Fig. 25, the scalar m needs to be expanded. The data shackle chosen by the compiler divides array A into block columns with block sizes ranging from 10 to 25 depending on the size of the problem. For the scale and update statements, the shackling references are chosen to be A(i, k) and A(j, 1) respectively. For the three statements implementing the row permutations, the shackling references are A(k, j), A(k, j) and A(ipvt(k), j) respectively, and for all the other statements, the shackling reference is A(i, k). We finally note that in this particular example, the expansion of m can be completely free, since ipvt(k) represents precisely a scalar expanded m; however this analysis is not currently implemented.

While the performance of the shackled code beats the performance of the tiled code, it is still slower than the LAPACK version by a factor of 2. This is because the LAPACK code uses information about the commutativity of permutations and row-updates; this permits it in essence to use two-dimensional blocking rather than block columns, which results in better code. Restructuring based on dependence analysis does not change the order of writes to a given memory location, so we believe that it is unlikely that a compiler that uses dependence analysis alone can deduce this commutativity condition automatically. We have recently shown that *fractal symbolic analysis* can solve this problem.[32]

9.3. QR Factorization

QR factorization performs orthogonal factorization of a matrix A into the product QR where Q is an orthonormal matrix and R is upper triangular. It is a key kernel in eigenvalue calculations. Figure 26 compares

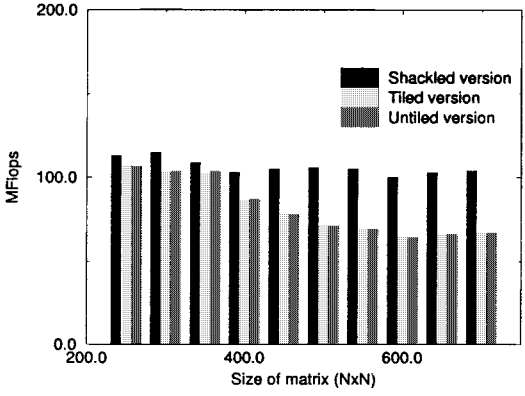


Fig. 26. QR factorization using householder reflections.

the performance of shackling and tiling on QR factorization using Householder reflections.[1] As in the case of LU factorization with partial pivoting, the array A is partitioned into block columns because a two-dimensional blocking is not legal. QR is similar to LU factorization except that in this case, *array expansion* of the vector x is required for legality. The necessary array expansion has not yet been implemented, so we modified the standard code for QR factorization to perform array expansion. Figure 26 shows this program. The need to expand x raises an important profitability question—scalar expansion is usually quite cheap, however expanding x creates an array as large as A in this case. Although shackling once again outperforms tiling, the performance of the shackled code is a factor of 2 worse than that of the LAPACK code which obtains roughly 225 MFlops on this code. The LAPACK code uses a number of deep properties of matrices, such as associativity of matrix product, to generate efficient code. It is conceivable that a compiler could exploit some of this information if the input program were written in a language like MATLAB or FORTRAN-90 in which array operations are primitives.

10. CONCLUSIONS

In this paper, we introduced the data-centric approach to optimizing scientific programs to improve cache performance, described an implementation of this approach within the SGI MIPSPro compiler, and compared the performance of data-centric code with that of code generated by more conventional control-centric techniques.

The key insight behind the data-centric approach is that it is often easier for a compiler to perform locality-enhancing transformations if it reasons about data structure traversals rather than control structure transformations. Intuitively, the compiler determines a schedule for the arrival of data structure elements in the cache, determines what computations should be performed when that data arrives, and generates the appropriate code. At runtime, program execution will automatically pull data into the cache in roughly that order, and the hit ratio should improve because statements that touch the same data are scheduled together.

We discussed a particular implementation of the data-centric approach called data-shackling which was designed for locality enhancement of dense numerical linear algebra codes. The traversals allowed are along the co-ordinate axes of the array (that is, left-to-right and top-to-bottom, and reversals of these); for each statement, a data-centric reference is chosen heuristically which determines the instances of that statement that are executed when a given data structure element is brought into the cache. The array itself is not physically copied to make the storage order of elements the same as the traversal order chosen by the data-shackle, although this can be done if it is deemed to be profitable.

This work can be extended in many ways.

As the experimental results in Section 9 show, data-shackling does not generate code competitive with LAPACK library code for LU factorization with pivoting and QR factorization. The blocked code in the LAPACK library for LU factorization with pivoting exploits the fact that row permutations commute with updates (see the code in Fig. 3). A compiler that uses dependence analysis preserves the order of reads and writes to a given memory location in the original program, so better analysis techniques must be developed. We have recently invented a technique called *fractal symbolic analysis* that addresses this problem[32] but its use in automatic locality enhancement remains to be explored. Blocking QR factorization requires the exploitation of domain-specific information like the associativity of matrix multiplication, and it is not clear how a compiler could be given such information or how it would use such information.

Our current implementation does not change the storage order of blocks of the array to make it the same as the traversal order chosen by the data-shackle, but nothing prevents us from reshaping the physical data array if the cost of converting back and forth from a standard representation is tolerable. Automatic data reshaping has been explored by other researchers,[27, 28] and it would be interesting to see what benefits this would have in our context.

Data-centric transformations other than data-shackling need to be explored. For example, Pugh and Rosser have proposed a data-centric

```
subroutine chol(A, n)
integer n, t1, t2, t3, t4, t5, t6, t7, t8, t9
double precision A(n,n)
do t1 = 1, (n-1)/64
   do t3 = 1, t1-1
      do t5 = 64*t3-63, 64*t3
         do t8 = 64*t1-63, 64*t1
            do t9 = t8, 64*t1
               s3(t5,t9,t8)
            enddo
         enddo
      enddo
   enddo
   do t5 = 64*t1-63, 64*t1
      s1(t5)
      do t7 = t5+1, 64*t1
         s2(t5,t7)
      enddo
      do t8 = t5+1, 64*t1
         do t9 = t8, 64*t1
            s3(t5,t9,t8)
         enddo
      enddo
   enddo
   do t2 = t1+1, (n-1)/64
      do t3 = 1, t1-1
         do t5 = 64*t3-63, 64*t3
            do t8 = 64*t1-63, 64*t1
               do t9 = 64*t2-63, 64*t2
                  s3(t5,t9,t8)
               enddo
            enddo
         enddo
      enddo
      do t5 = 64*t1-63, 64*t1
         do t7 = 64*t2-63, 64*t2
            s2(t5,t7)
         enddo

         do t8 = t5+1, 64*t1
            do t9 = 64*t2-63, 64*t2
               s3(t5,t9,t8)
            enddo
         enddo
      enddo
   enddo
   do t2 = (n+63)/64, (n+63)/64
      do t3 = 1, t1-1
         do t5 = 64*t3-63, 64*t3
            do t8 = 64*t1-63, 64*t1
               do t9 = 64*t2-63, n
                  s3(t5,t9,t8)
               enddo
            enddo
         enddo
      enddo
      do t5 = 64*t1-63, 64*t1
         do t7 = 64*t2-63, n
            s2(t5,t7)
         enddo
         do t8 = t5+1, 64*t1
            do t9 = 64*t2-63, n
               s3(t5,t9,t8)
            enddo
         enddo
      enddo
   enddo
enddo
if (n .GE. 1) then
   do t1 = (n+63)/64, (n+63)/64
      do t3 = 1, t1-1
         do t5 = 64*t3-63, 64*t3
            do t8 = 64*t1-63, n
               do t9 = t8, n
                  s3(t5,t9,t8)
               enddo
            enddo
         enddo
      enddo
      do t5 = 64*t1-63, n
         s1(t5)
         do t7 = t5+1, n
            s2(t5,t7)
         enddo
         do t8 = t5+1, n
            do t9 = t8, n
               s3(t5,t9,t8)
            enddo
         enddo
      enddo
   enddo
endif
```

Fig. 27. Doubly-blocked Cholesky factorization code produced by data shackling.

transformation called *iteration space slicing*[33] which can be viewed as a more systematic way of using dependences than the heuristics described in Section 8 for finding data-centric references. A backward slice for a variable at a given point in the program is the portion of the program that affects the value of that variable at that program point.[34] Given an order in which array elements are to be computed, iteration space slicing produces a program that is effectively the sequence of incremental slices of successive elements of the array. However, computing incremental slices is computationally expensive, and the performance advantage over simple heuristics is unclear.

Finally, large computational science problems usually involve sparse matrices produced by the applications of methods like the finite-element method. Sparse matrix programs cannot be restructured or optimized by compilers because array accesses in these programs involve indirect subscripts that are difficult for a compiler to reason about. We have shown that this problem can be circumvented by using the data-centric approach to synthesize sparse matrix programs from dense matrix programs and specifications of sparse formats.[35] The sparse format is specified in part by describing efficient traversal orders for enumerating the elements of the matrix, and the restructuring compiler attempts to use one of these data element traversal orders to perform the specified computation.

APPENDIX

Figure 27 shows the code produced by the product of shackles discussed in Section 7.

ACKNOWLEDGMENTS

Nawaaz Ahmed participated in some of the early work on data-shackling. Dror Maydan and Robert Cox helped with the SGI implementation of data shackling. Rob Schreiber and Charlie van Loan explained many of the LAPACK algorithms to us. Finally, Nikolai Mateev, Vijay Menon, and Paul Stodghill gave us valuable feedback on this paper.

REFERENCES

1. G. Golub and C. Van Loan, *Matrix Computations*, Johns Hopkins University Press (1996).
2. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, (eds.), *LAPACK Users' Guide*, Second Edition, SIAM, Philadelphia (1995).
3. U. Banerjee, Unimodular Transformations of Double Loops, *Proc. Workshop Adv. Lang. Compilers for Parallel Processing*, pp. 192–219 (August 1990).

4. P. Boulet, A. Darte, T. Risset, and Y. Robert, (Pen)-Ultimate Tiling? *INTEGRATION, VLSI J.*, **17**:33–51 (1994).
5. S. Carr and R. B. Lehoucq, Compiler Blockability of Dense Matrix Factorizations. Technical Report, Argonne National Laboratory (October 1996).
6. L. Carter, J. Ferrante, and S. Flynn Hummel, Hierarchical Tiling for Improved Superscalar Performance, *Int'l. Parallel Processing Symp.* (April 1995).
7. J. Dongarra and R. Schreiber, Automatic Blocking of Nested Loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee (May 1990).
8. F. Irigoin and R. Triolet, Supernode Partitioning, *ACM Symp. Principles of Progr. Lang.*, pp. 319–329 (January 1988).
9. W. Li and K. Pingali, Access Normalization: Loop Restructuring for NUMA Compilers, *ACM Trans. Computer Syst.* (1993).
10. J. Ramanujam and P. Sadayappan, Tiling Multidimensional Iteration Spaces for Multicomputers, *J. Parallel and Distributed Computing*, **16**(2):108–120 (October 1992).
11. V. Sarkar, Automatic Selection of High-Order Transformations in the IBM ASTI Optimizer. Technical Report ADTI-96-004, Application Development Technology Institute, IBM Software Solutions Division (July 1996). Submitted to special issue of *IBM Journal of Research and Development*.
12. M. E. Wolf and M. S. Lam, A Data Locality Optimizing Algorithm, *SIGPLAN Conf. Progr. Lang. Design Implementation* (June 1991).
13. M. Wolfe, Iteration Space Tiling for Memory Hierarchies, *Third SIAM Conf. Parallel Processing for Scientific Computing* (December 1987).
14. M. E. Wolf, D. E. Maydan, and D.-K. Chen, Combining Loop Transformations Considering Caches and Scheduling, *MICRO 29*, Silicon Graphics, Mountain View, California, pp. 274–286 (1996).
15. R. C. Agarwal and F. G. Gustavson, *Algorithm and Architecture Aspects of Producing ESSL BLAS on POWER2.*
16. M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison–Wesley Publishing Company (1995).
17. U. Banerjee, Unimodular Transformations of Double Loops, *Lang. and Compilers for Parallel Computing*, pp. 192–219 (1990).
18. M. S. Lam, E. E. Rothberg, and M. E. Wolf, The Cache Performance and Optimizations of Blocked Algorithms, *Proc. Fourth Int'l. Conf. Architectural Support Progr. Lang. Oper. Syst.*, pp. 63–74, Santa Clara, California, ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society (April 1991).
19. W. Li and K. Pingali, A Singular Loop Transformation Based on Nonsingular Matrices, *IJPP*, **22**(2) (April 1994).
20. S. Coleman and K. S. McKinley, Tile Size Selection Using Cache Organization and Data Layout, *ACM SIGPLAN Conf. Progr. Lang. Design and Implementation (PLDI)*, ACM Press (June 1995).
21. S. Carr and K. Kennedy, Compiler Blockability of Numerical Algorithms, *Supercomputing* (1992).
22. S. Carr and R. B. Lehoucq, A Compiler-Blockable Algorithm for QR Decomposition (1994).
23. R. Schreiber and J. Ramanujam, Personal communication (September 1997).
24. K. S. McKinley, S. Carr, and C.-W. Tseng, Improving Data Locality with Loop Transformations, *ACM Trans. Progr. Lang. Syst.*, **18**(4):424–453 (July 1996).

25. Y. Song and Z. Li, New Tiling Techniques to Improve Cache Locality, *SIGPLAN99 Conf. Progr. Lang. Design Implementation* (June 1999).
26. A. Rogers and K. Pingali, Process Decomposition Through Locality of Reference, *SIGPLAN Conf. Progr. Lang. Design and Implementation* (June 1989).
27. J. Anderson, S. Amarsinghe, and M. Lam, Data and Computation Transformations for Multiprocessors, *ACM Symp. Principles and Practice of Parallel Programming* (June 1995).
28. M. Cierniak and W. Li, Unifying Data and Control Transformations for Distributed Shared Memory Machines, *SIGPLAN Conf. Progr. Lang. Design and Implementation* (June 1995).
29. W. Pugh, Counting Solutions to Presburger Formulas: How and Why. Technical report, University of Maryland (1993).
30. P. Clauss, Counting Solutions to Linear and Nonlinear Constraints through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs, *ACM Int'l. Conf. Supercomputing* (May 1996).
31. I. Kodukula, Data-centric Compilation, Ph.D. thesis, Cornell University (1998).
32. N. Mateev, V. Menon, and K. Pingali, Fractal Symbolic Analysis for Program Transformations. To appear as a Cornell CS Technical Report.
33. W. Pugh and E. Rosser, Iteration Space Slicing for Locality, *Proc. 12th Int'l. Workshop Lang. Compilers for Parallel Computing, (LCPC99)* (August 1999).
34. M. Weiser, Program Slicing, *IEEE Trans. Software Engineering*, **10**(4):352–357 (1984).
35. V. Kotlyar, K. Pingali, and P. Stodghill, A Relational Approach to the Compilation of Sparse Matrix Programs, in *EUROPAR* (1997).

Printed in Belgium