Data-centric Multi-level Blocking

Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali
Department of Computer Science,
Cornell University, Ithaca, NY 14853.
{prakas,ahmed,pingali}@cs.cornell.edu

Abstract

We present a simple and novel framework for generating blocked codes for high-performance machines with a memory hierarchy. Unlike traditional compiler techniques like tiling, which are based on reasoning about the control flow of programs, our techniques are based on reasoning directly about the flow of data through the memory hierarchy. Our data-centric transformations permit a more direct solution to the problem of enhancing data locality than current control-centric techniques do, and generalize easily to multiple levels of memory hierarchy. We buttress these claims with performance numbers for standard benchmarks from the problem domain of dense numerical linear algebra. The simplicity and intuitive appeal of our approach should make it attractive to compiler writers as well as to library writers.

1 Introduction

Data reuse is imperative for good performance on modern high-performance computers because the memory architecture of these machines is a hierarchy in which the cost of accessing data increases roughly ten-fold from one level of the hierarchy to the next. Unfortunately, programs with good data reuse cannot be obtained for most problems by straight-forward coding of standard algorithms. In some cases, it is necessary to develop new algorithms which exploit structure in the underlying problem to reuse data effectively; a well-known example of this is Bischof and van Loan's WY algorithm for QR factorization with Householder reflections, which was developed explicitly for improving data reuse in orthogonal factorizations [15]. Even when existing algorithms are sufficient, reorganizing a program to reuse data effectively can increase its size by orders of magnitude, and make the program less abstract and less portable by introducing machine dependencies.

In this paper, we describe restructuring compiler technology that reorganizes computations in programs to enhance data reuse. We evaluate the performance of this technology in the problem domain of dense numerical linear algebra. This problem domain is appropriate because it is important in practice, and there are libraries of hand-crafted programs with good data reuse which can be used for comparisons with automatically generated code. Some of these programs are discussed in Section 2.

The rest of this paper is organized as follows. In Section 3, we discuss current solutions to the problem of developing software with good data reuse, including hand-crafted libraries like LAPACK [2], and automatic compiler techniques such as tiling [24]. The technology described in this paper was motivated by the limitations of these approaches, and is introduced in Section 4. This technology differs from standard restructuring compiler technology like tiling because it is based on reasoning about the *data flow* rather than the *control flow* of the program. In a sense that is made precise later in the paper, our approach is *data-centric*, and it should be contrasted with existing compiler techniques for promoting data reuse, which are *control-centric*. In Section 5, we present a general view of data-centric transformations, and show how to reason about the correctness of such transformations. In Section 6, we show how data-centric transformations can be combined together to produce new transformations, and use these ideas to generate transformations to enhance data reuse in common dense linear algebra benchmarks. In Section 7, we describe performance results. Finally, we discuss ongoing work in Section 8.

2 Running Examples

Figure 1 shows three important computational kernels that we will use to illustrate the concepts in this paper. Figure 1(i) shows matrix multiplication in the so-called `I-J-K` order of loops. It is elementary for a compiler to deduce the well-known fact that all six permutations of these three loops are legal. This loop is called a *perfectly nested loop* because all assignment statements are contained in the innermost loop. Figure 1(ii, iii) show two versions of Cholesky factorization called *right-looking* and *left-looking* Cholesky factorization; both these loop nests are *imperfectly nested* loops. Both codes traverse the matrix `A` a column at a time. In right-looking Cholesky, the columns to the right of the current column are updated by the `L-K` loop nest, using the outer product of the current column, as shown in Figure 2(i). The `L` and `K` loops are called the *update loops*. Left-looking Cholesky performs lazy updates in the sense a column is updated only when it is visited by

[0]This work was supported by NSF grant CCR-9503199, ONR grant N00014-93-1-0103, and the Cornell Theory Center.

```
do I = 1..N
  do J = 1..N
    do K = 1..N
      C[I,J] = C[I,J] + A[I,K] * B[K,J]
```

(i) Matrix Multiplication

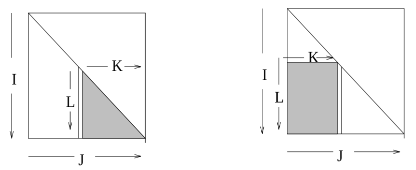
```
do J = 1..N
  S1: A[J,J] = sqrt (A[J,J])
  do I = J+1..N
    S2: A[I,J] = A[I,J] / A[J,J]
  do L = J+1..N
    do K = J+1..L
       S3: A[L,K] = A[L,K] - A[L,J] * A[K,J]
```

(ii) Right-looking Cholesky Factorization

```
do J = 1..N
   do L = J..N
     do K = 1..(J-1)
        S3: A[L,J] = A[L,J] - A[L,K]*A[J,K]
   S1:A[J,J] = sqrt(A[J,J])
   do I = J+1..N
     S2: A[I,J] = A[I,J] / A[J,J]
```

(iii) Left-looking Cholesky Factorization

Figure 1: Running examples - Matrix Multiply and Cholesky



(i) Right-looking Cholesky *(ii) Left-looking Cholesky*

Figure 2: Pictorial View of Cholesky Factorization

the outermost loop. The shaded area to the left of the current column in Figure 2(*ii*) shows the region of the matrix that is read for performing this update.

3 Previous Work

The numerical analysis community has used a layered approach to the problem of writing portable software with good data reuse. The general idea is to (a) find a set of core operations for which algorithms with good data reuse are known, (b) implement carefully hand-tuned implementations of these algorithms on all platforms, and (c) use those operations, wherever possible, in writing programs for applications problems. From Amdahl's law, it follows that if most of the computational work of a program is done in the core operations, the program will perform well on a machine with a memory hierarchy. While the implementations of these operations are not portable, the rest of the software is machine-independent.

In the context of dense numerical linear algebra, the core operation is matrix multiplication. The standard algorithm for multiplying two $n \times n$ matrices performs n^3 operations on n^2 data, so it has excellent data reuse. Most vendors provide so-called *Level-3 BLAS* routines which are carefully hand-optimized, machine-tuned versions of matrix multiplication. To exploit these routines, the numerical analysis community has invested considerable effort in developing *block-matrix algorithms* for standard dense linear algebra problems such as Cholesky, LU and QR factorizations. These block algorithms operate on entire submatrices at a time, rather than on individual matrix elements, and are rich in matrix multiplications. The well-known LAPACK library contains block matrix algorithms implemented on top of the BLAS routines, and is written for good data reuse on a machine with a two-level memory hierarchy [2].

The LAPACK library has been successful in practice. However, it requires a set of machine-specific, hand-coded BLAS routines to run well. Since it is not a general-purpose tool, it cannot be used outside the realm of the dense numerical linear algebra. It is also specifically written for a two-level memory hierarchy, and it must be re-implemented for machines with deeper memory hierarchies. Therefore, automatic program restructuring tools that promote data reuse through transformations provide an attractive alternative.

The restructuring compiler community has devoted much attention to the development of such technology. The most important transformation is Mike Wolfe's *iteration space tiling* [24], preceded by *linear loop transformations* if necessary [4, 17, 23]. This approach is restricted to perfectly nested loops, although it can be extended to imperfectly nested loops if they are first transformed into perfectly nested loops. A loop in a loop nest is said to *carry reuse* if the same data is touched by multiple iterations of that loop for fixed outer loop iterations. For example, loop `K` in Figure 1(*i*) carries reuse because for fixed `I` and `J`, all iterations of the loop touch `C[I,J]`; similarly, loop `I` carries reuse because successive iterations of the `I` loop touch `B[K,J]`. Loops that carry reuse are moved as far inside the loop nest as possible by using linear loop transformations; if two or more inner loops carry reuse and they are *fully permutable*, these loops are tiled [24]. Intuitively, tiling improves performance by interleaving iterations of the tiled loops, which exploits data reuse in all those loops rather than in just the innermost one. It is easy to verify that all the three loops in the matrix multiplication code carry reuse and are fully permutable. Tiling all three loops produces the code shown in Figure 3 (for 25×25 tiles). The three outer loops enumerate the iteration space tiles, while the three inner loops enumerate the iteration space points within a tile. In this case, iteration space tiling produces the same code as the equivalent block matrix code [15].

Tiling can be applied to imperfectly nested loops if these loops are converted to perfectly nested loops through the use of *code sinking* [25]. Code sinking moves all statements into the innermost loop, inserting appropriate guards to ensure that these statements are executed the right number of times. There is no unique way to sink code in a given loop nest; for example, in left-

```
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]
```

Figure 3: Blocked Code for matrix matrix multiplication

looking Cholesky, statement S1 can be sunk into the I loop or into the L-K loop nest. Other choices arise from the possibility of doing imperfectly nested loop transformations (especially loop jamming) during the code sinking process. Depending on how these choices are made, one ends up with different perfectly nested loops, and the resulting programs after linear loop transformations and tiling may exhibit very different performance. In right-looking Cholesky for example, if S1 is sunk into the I loop, and the resulting loop is sunk into the L-K loop nest, we end up with a 4-deep loop nest in which only the update loops can be tiled, even if linear loop transformations are performed on the perfectly nested loop. If on the other hand, we jam the I and L loops together, and then perform sinking, we get a fully permutable loop nest; tiling this loop nest produces code with much better performance. Similarly, it can be shown that for left-looking Cholesky, the best sequence of transformations is to first sink S1 into the I loop, jam the I and L loops together and then sink S1 and S2 into K loop. Interactions between loop jamming/distribution and linear loop transformations have been studied by McKinley *et al* [18]. However, no systematic procedure for exploring these options for obtaining perfectly nested loops from imperfectly nested loops is known.

A somewhat different approach has been taken by Carr and Kennedy [7]. By doing a detailed study of matrix factorization codes in the LAPACK library, they came up with a list of transformations that must be performed to get code competitive with LAPACK code. These include strip-mine-and-interchange, preceded by index-set-splitting and loop distribution to make the interchange legal [8]. Additional transformations such as unroll-and-jam [7] and scalar replacement are performed on this code to obtain code competitive with hand-blocked codes used in conjunction with BLAS [8]. However, it is unclear how a compiler can discover automatically the right sequence of transformations to perform; it is also unclear whether this approach can be generalized for a machine with a multi-level memory hierarchy.

Finally, there is a large body of work on determining good tile sizes [6, 14, 20, 22, 12]. This research focuses on perfectly nested loops with uniform dependences (i.e. dependence vectors can be represented as distances). While this work is not directly comparable to ours, the detailed memory models used in some of this research [18, 22, 16] are useful in general for estimating program performance.

4 Data-centric Transformations

Since the goal of program transformation is to enhance data reuse and reduce data movement through the memory hierarchy, it would seem advantageous to have a tool that orchestrates data movement *directly*, rather than as a side-effect of control flow manipulations. The ultimate result of the orchestration is, of course, a transformed program with the desired data reuse, but to get that program, the tool would reason directly about the desired data flow rather than about the control flow of the program. A useful analogy is signal processing. The input and the output of signal processing is a signal that varies with time, and in principle, all processing can be done in the time domain. However, it is often more convenient to take a Fourier transform of the signal, work in the frequency domain and then take an inverse Fourier transform back into the time domain.

4.1 Data Shackle

In the rest of the paper, the phrase *statement instance* refers to the execution of a statement for given values of index variables of loops surrounding that statement.

Definition 1 *A* data shackle *is a specification in three parts.*

- *We choose a data object and divide it into* blocks.
- *We determine a sequence in which these blocks are "touched" by the processor.*
- *For each block, we determine a set of statement instances to be performed when that block is touched by the processor. However, we leave unspecified the order of enumeration of statement instances within this set.*[1]

We now look at this in detail.

- The data objects of interest to us are multidimensional arrays. An array can be sliced into blocks by using a set of parallel *cutting planes* with normal n, separated by a constant distance d. Further slicing can be performed by using additional sets of planes inclined with respect to this set of planes. We define the *cutting planes matrix* as the matrix whose columns are the normals to the different sets of cutting planes; the order of these columns is determined by the order in which the sets of cutting planes are applied to block the data. Figure 4 shows the blocking of a two-dimensional array with two sets of cutting planes; the cutting planes matrix is $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.
- A block is assigned the *block co-ordinate* $(x_1, \ldots, x_m)$ if it is bounded by cutting planes numbered $x_i - 1$ and x_i from the i^{th} set of cutting planes. The code that we generate 'schedules' blocks by enumerating them in lexicographic order of block co-ordinates.

[1] One possible order for executing these statement instances is to use the same order as the initial code. We leave the order unspecified because it permits us to join data shackles together to get finer degrees of control on the execution, as we will see in Section 6.

- The final step is to specify the statement instances that should be performed when a block is scheduled. From each statement `S`, we choose a *single* reference `R` of the array that is being blocked. For now, we assume that a reference to this array appears in every statement. We will relax this condition in Section 5.
  When a block of data is scheduled, we execute all instances of `S` for which the data touched by reference `R` is contained in this block[2]. As mentioned before, the order in which these instances should be done is left unspecified.

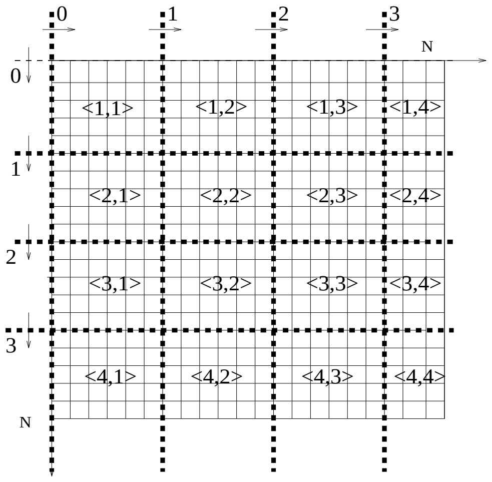


Figure 4: Cutting planes on a data object

The rationale for the term "data shackle" should now be clear. One thinks of an instrument like a pantograph in which a master device visits the blocks of data in lexicographic order, while a slave device shackled to it is dragged along some trajectory in the iteration space of the program. It is also convenient to be able to refer to statement instances "shackled" to a particular block of data.

Since a data shackle reorders computations, we must check that the resulting code respects dependences in the original program. Legality can be checked using standard techniques from polyhedral algebra, and is discussed in Section 5.

As an example of a data shackle, consider the matrix multiplication code of Figure 1(i). Let us block matrix `C` as shown in Figure 4, using a block size of 25 X 25, and shackle the reference `C[I,J]` in Figure 1(i) to this blocking. This data shackle requires that when a particular block of `C` is scheduled, all statement instances that write into this block of data must be performed by the processor. We can require these to be performed in program order of the source program. Naive code for accomplishing this is shown in Figure 5. The two outer loops iterate over the blocks of `C`. For every block of `C`, the entire original iteration space is visited, and every iteration is examined to see if it should be executed. If the location of `C` accessed by the iteration falls in the current block (which is determined by the conditional in the code), that iteration is executed. It is easy to see that in the presence of affine references, the conditionals are all affine conditions on the loop indices corresponding to the cutting plane sets and the initial loop index variables.

```
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]
```

Figure 5: Naive code produced by blocking C for matrix matrix multiply

The code in Figure 5 is not very efficient, and is similar to *runtime resolution* code generated when shared-memory programs are compiled for distributed-memory machines [21]. Fortunately, since the conditionals are affine conditions on surrounding loop bounds, they can be simplified using any polyhedral algebra tool. We have used the Omega calculator [19] to produce the code shown in Figure 6. It is simple to verify that this code has the desired effect of blocking the `C` array since blocks of `C` are computed at a time by taking the product of a block row of `A` and a block column of `B`. This code is *not* the same as the code for matrix matrix product in the BLAS library used by LAPACK since the block row/column of `A/B` are not blocked themselves. Our data shackle constrains the values of the loop indices `I` and `J` in Figure 1(i), but leaves the values of `K` unconstrained. This problem is addressed in Section 6 where we discuss how data shackles can be combined.

In the Cholesky factorization code, array `A` can be blocked in a manner similar to Figure 4. When a block is scheduled, we can choose to perform all statement instances that *write* to that block (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. Using polyhedral algebra tools, we obtain the code in Figure 7. In this code, data shackling regroups the iteration space into four sections as shown in Figure 8. Initially, all updates to the diagonal block from the left are performed (Figure 8(i)), followed by a *baby Cholesky factorization* [15] of the diagonal block (Figure 8(ii)). For each off-diagonal block, updates from the left (Figure 8(iii)) are followed by interleaved scaling of the columns of the block by the diagonal block, and local updates(Figure 8(iv)).

Note that just as in the case of matrix matrix product, this code is only partially blocked (compared to LAPACK code) — although all the writes are performed into a block when we visit it, the reads are not localized to blocks. Instead, the reads are distributed over the entire left portion of the matrix. As before, this problem is solved in Section 6.

[2] Because of other data references in statement `S`, these statement instances may touch data outside that block.

```
do t1 = 1 ..  ⌈(N/25)⌉
 do t2 = 1 ..  ⌈(N/25)⌉
  do It = (t1-1)*25 +1 .. min(t1*25,N)
   do Jt = (t2-1)*25 +1 .. min(t2*25,N)
    do K = 1 .. N
     C[It,Jt] = C[It,Jt] + A[It,K] * B[K,Jt]
```

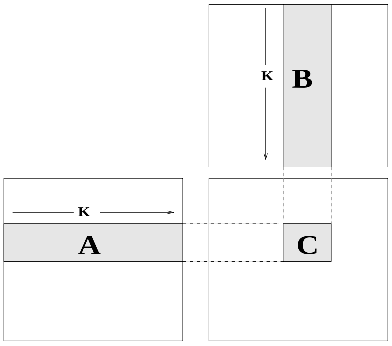


Figure 6: Simplified code produced by blocking C for matrix matrix multiply

4.2 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* [23]. 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 combining shackles as described in Section 6.

The code shown in Figure 7 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. In this regard, it is important to understand the "division of work" between our data-centric transformation and a polyhedral algebra tool like

```
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)

  do t2 = t1+1, (n+63)/64
    /* Apply updates from left to
       off-diagonal block */
    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)
```

Figure 7: Data shackle applied to right-looking Cholesky factorization

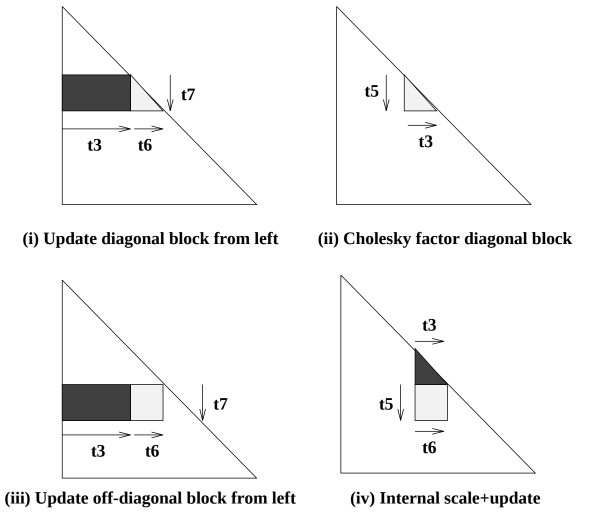


Figure 8: Pictorial View of Code in Figure 7

Omega. Enforcing a desired pattern of data accesses is obtained by choosing the right data shackle - note that the pattern of array accesses made by the code of Figure 5, which is obtained directly from the specification of the data shackle without any use of polyhedral algebra tools, is identical to the pattern of array accesses made by the simplified code of Figure 6. The role of polyhedral algebra tools in our approach is merely to simplify programs, as opposed to producing programs with desired patterns of data accesses.

5 Legality

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 and b_2 must be identical, or b_1 must be touched before b_2. If so, 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 an integer linear programming problem.

5.1 An Example

To understand the general algorithm, it is useful to consider first a simple example: in right-looking Cholesky factorization, we formulate the problem of ensuring that the flow dependence from the assignment of `A[J,J]` in S1 to the use of `A[J,J]` in S2 is respected by the data shackle from which the program of Figure 7 was generated[3]. We first write down a set of integer inequalities that represent 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 J_w of the J loop, and let S2 read from that location in iteration (J_r, I_r) of the J and I loops. A flow dependence exists if the following linear inequalities have an integer solution [25]:

$$\begin{cases} J_r = J_w, I_r = J_w & \text{(same location)} \\ N \geq J_w \geq 1 & \text{(loop bounds)} \\ N \geq J_r \geq 1 & \text{(loop bounds)} \\ N \geq I_r \geq J_r + 1 & \text{(loop bounds)} \\ J_r \geq J_w & \text{(read after write)} \end{cases} \qquad (1)$$

Next, we assume that the instance of S2 is performed when a block (b_{11}, b_{12}) is scheduled, and the instance of S1 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 put in a condition which states that block (b_{11}, b_{12}) is "touched" strictly after (b_{21}, b_{22}). These conditions are represented as:

$$\begin{cases} \text{Writing iteration done in } (b_{11}, b_{12}) \\ b_{11} * 25 - 24 \leq J_w \leq b_{11} * 25 \\ b_{12} * 25 - 24 \leq J_w \leq b_{12} * 25 \\ \text{Reading iteration done in } (b_{21}, b_{22}) \\ b_{21} * 25 - 24 \leq J_r \leq b_{21} * 25 \\ b_{22} * 25 - 24 \leq I_r \leq b_{22} * 25 \\ \text{Blocks visited in bad order} \\ (b_{11} < b_{21}) \vee ((b_{11} = b_{21}) \wedge (b_{12} < b_{22})) \end{cases} \qquad (2)$$

If the conjunction of the two sets of conditions (1) and (2) has an integer solution, it means that there is a dependence, and that dependent instances are performed in the wrong order. Therefore, if the conjunction has an integer solution, the data shackle violates the dependence and is not legal. This problem can be viewed geometrically as asking whether a union of certain polyhedra contains an integer point, and can be solved using standard polyhedral algebra.

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

5.2 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 Figure 4, 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 x 25, array element (a_1, a_2) is mapped to the co-ordinate $((a_1\ div\ 25) + 1, (a_2\ div\ 25) + 1)$ in a new two-dimensional space. Note that this map is not one-to-one. The bottom part of Figure 9 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 (V, $\prec$). For the blocking shown in Figure 9, C:(S,I) $\rightarrow A$ maps statement instances to elements of array A through data-centric references, and T:A $\rightarrow$ V maps array elements to block co-ordinates. Concisely, $\mathbf{M}$ = T$\circ$C.

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

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

- *Introduce vectors of unknowns* $\underline{i1}$ *and* $\underline{i2}$ *that represent instances of dependent statements* S1 *and* S2 *respectively.*

[3] The shackle was produced by blocking the matrix A as shown in Figure 4, and choosing the left hand side references of all assignment statements in Figure 1(ii) for shackling.

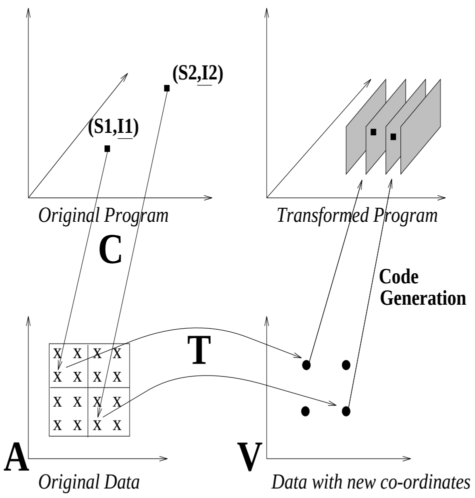


Figure 9: Testing for Legality

- *Formulate the inequalities that must be satisfied for a dependence to exist from instance* $\underline{\mathtt{i1}}$ *of statement* S1 *to instance* $\underline{\mathtt{i2}}$ *of statement* S2. *This is standard [25].*
- *Formulate the predicate* $\mathbf{M}(\mathtt{S2},\underline{\mathtt{i2}}) \prec \mathbf{M}(\mathtt{S1},\underline{\mathtt{i1}})$.
- *The conjunction of these conditions does not have an integer solution.*

Proof: Obvious, hence omitted. □

5.3 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; *the physical array itself is not necessarily reshaped.* For example, in the blocked matrix multiplication code in Figure 6, 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. Physical data reshaping has been explored by other researchers [11, 3].

Upto this point, we have assumed that every statement in the program contains a reference to the array being blocked by the data shackle. Although this assumption in valid for kernels like matrix multiplication and Cholesky factorization, it is obviously not true in general programs. Our current approach to this problem is naive but simple. If a statement does not contain a reference to the array being blocked by the data shackle, we simply add a dummy reference to that array (such as `+ 0*B[I,J]`) to the right hand side of the statement. The dummy reference is of course irrelevant for dependence analysis, and serves only to determine which instances of this statement are performed when elements of B are touched by the data shackle. The precise expression used in the dummy reference is irrelevant for correctness, but a data shackle that is illegal for one choice of this expression may be legal if some other expression is used (since that changes the order in which the statement instances are performed). This is clearly an issue that we need to revisit in the future, and we plan to use tools we developed for automatic data alignment to address this problem more carefully [5].

6 Products of shackles

We now show that there is a natural notion of taking the Cartesian product of a set of shackles. The motivation for this operation comes from the matrix multiplication code of Figure 6, 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 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 in unconstrained references to the other two arrays. To get LAPACK-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 across different partitions of 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 5 that a data shackle for a program P can be viewed as a map $\mathbf{M}{:}(\mathtt{S},\mathtt{I}) \rightarrow \mathtt{V}$, whose domain is the set of statement instances and whose range is a totally ordered set.

Definition 2 *For any program P, let*

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

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

$$(\mathtt{M}_1 \times \mathtt{M}_2)(\mathtt{S},\underline{\mathtt{i}}) \ = \ < \mathtt{M}_1(\mathtt{S},\underline{\mathtt{i}}), \mathtt{M}_2(\mathtt{S},\underline{\mathtt{i}}) >$$

The product domain $\mathtt{V}_1 \times \mathtt{V}_2$ 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 5. It is easy to see that for each point $\underline{v_1} \times \underline{v_2}$ in the product domain $\mathtt{V}_1 \times \mathtt{V}_2$, we perform the statement instances in the set $(\mathtt{M}_1 \times \mathtt{M}_2)^{-1}(\underline{v_1}, \underline{v_2}) = \mathtt{M}_1{}^{-1}(\underline{v_1}) \cap \mathtt{M}_2{}^{-1}(\underline{v_2})$.

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 $\mathtt{M}_1 \times \mathtt{M}_2$ can be legal even if $\mathtt{M}_2$ 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.

6.1 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 $\mathtt{M}_C \times \mathtt{M}_A$ of the C and A shackles produces the code in Figure 3. It is interesting to note that further shackling with the B shackle (that is the product $\mathtt{M}_C \times \mathtt{M}_A \times \mathtt{M}_B$) 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 Figure 1(ii), it is easy to verify that there are six ways to shackle references in the source program to blocks of the matrix (choosing `A[J,J]` from statement `S1`, either `A[I,J]` or `A[J,J]` from statement `S2` and either `A[L,K]`, `A[L,J]` or `A[K,J]` from statement `S3`). Of these, only two are legal: choosing `A[J,J]` from `S1`, `A[I,J]` from `S2` and `A[L,K]` from `S3`, or choosing `A[J,J]` from `S1`, `A[J,J]` from `S2` and `A[L,J]` 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, identical to the blocked Cholesky algorithm in [15], while the other order gives a fully-blocked right-looking Cholesky.

6.2 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. The following obvious result is useful to determine how far to carry the process of taking Cartesian products. We assume that all array access functions are linear functions of loop variables (if the functions are affine, we drop the constant terms); if so, they can be written as $F * \underline{I}$ where F is the **data access matrix** [17] and $\underline{I}$ is the vector of iteration space variables of loops surrounding this data reference.

Theorem 2 *For a given statement S, let $F_1, \ldots, F_n$ 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 $F_1, \ldots, F_n$.*

Stronger versions of this result can be proved, but it suffices for our purpose in this paper. For example, the access matrix for the reference `C[I,J]` is $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$. Shackling this reference does not bound the data accessed by row $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ of the access matrix $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ 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]`.

Although the problem of generating good shackles automatically is beyond the scope of this paper, Cartesian product is obviously viewed as a way of generating new shackles from old ones. The discussion in this section provides some hints for generating good data shackles.

- *How are the data-centric references chosen?* For each statement, the data-centric references should be chosen such that there are no remaining unconstrained references.
- *How big should the Cartesian products be?* If there is no statement left which has an unconstrained reference, then there is no benefit to be obtained from extending the product.
- *What is a correct choice for orientation of cutting planes?* To a first order of approximation, the orientation of the cutting planes is irrelevant as far as performance is concerned, provided the blocks have the same volume. Of course, orientation is important for legality of the data shackle.

These matters and related ones are currently being investigated and will be addressed in the forthcoming thesis of the first author of the paper.

6.3 Multi-level Blocking

Cartesian products can be used in an interesting manner to block programs for multiple levels of memory hierarchy. For lack of space, we explain only the high level

```
do t1 = 1, (n+63)/64
do t2 = 1, (n+63)/64
do t3 = 1, (n+63)/64
  do t7 = 8*t1-7, min(8*t1,(n+7)/8)
  do t8 = 8*t2-7, min(8*t2,(n+7)/8)
  do t9 = 8*t3-7, min(8*t3,(n+7)/8)
    do t13 = 8*t9-7, min(n,8*t9)
    do t14 = 8*t8-7, min(n,8*t8)
    do t15 = 8*t7-7, min(8*t7,n)
      C[t13,t14]+= A[t13,t15]*B[t15, t14]
```

Figure 10: Matrix multiply blocked for two levels of memory hierarchy

idea here. For a multi-level memory hierarchy, we generate a Cartesian product of products of shackles where each factor in the outer Cartesian product determines blocking for one level of the memory hierarchy.

The first term in the outer Cartesian product corresponds to blocking for the slowest (and largest) level of the memory hierarchy and corresponds to largest block size. Subsequent terms correspond to blocking for faster (and usually smaller) levels of the memory hierarchy. We have applied this idea to obtain multiply blocked versions of our running examples in a straightforward fashion. It is unclear to us that tiling can be generalized to multiple levels of memory hierarchy in such a straightforward manner.

Figure 10 demonstrates this idea for matrix multiplication. The outer Cartesian product for this example has two factors: the first factor is itself a product of two shackles (on C[I,J] and A[I,K] with block sizes of 64), and the second factor is also a product of two shackles (once again, on C[I,J] and A[I,K], but block sizes of 8). As can be seen from the code, the first term of the outer Cartesian product performs a 64-by-64 matrix multiplication, which is broken down into several 8-by-8 matrix multiplications by the second term in this product. By choosing smaller inner blocks (like 2-by-2) and unrolling the resulting loops, we can block for registers.

7 Performance

We present performance results on a thin node of the IBM SP-2 for the following applications: ordinary and banded Cholesky factorizations, QR factorization, the ADI kernel and the GMTRY benchmark from the NAS suite. All compiler generated codes were compiled on the SP-2 using xlf -O3.

Figure 11 shows the performance of Cholesky factorization. The lines labeled *Input right-looking code* show the performance of the right-looking Cholesky factorization code in Figure 1(ii). This code runs at roughly 8 MFlops. The lines labeled *Compiler generated code* show the performance of the fully blocked left-looking Cholesky code produced by the Cartesian product of data shackles discussed in Section 6. While there is a dramatic improvement in performance from the initial code, this blocked code still does not get close to peak performance because the compiler back-end does not perform necessary optimizations like scalar replacement in innermost loops. A large portion of execution time

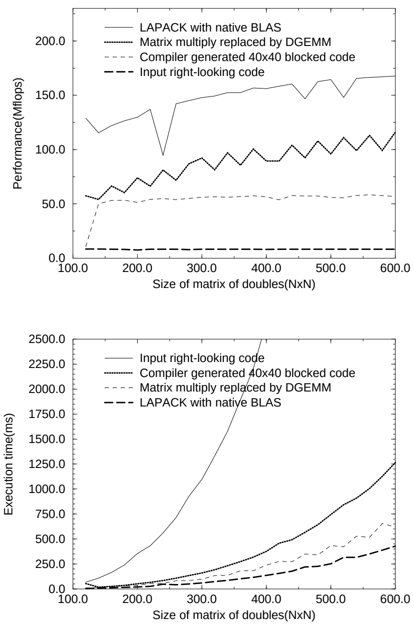


Figure 11: Cholesky factorization on the IBM SP-2

is spent in a few lines of code which implement matrix multiplication, but which are optimized poorly by the IBM compiler. Replacing these lines manually by a call to the ESSL BLAS-3 matrix multiplication routine improves performance considerably, as is shown by the lines labeled *Matrix Multiply replaced by DGEMM*. Finally, the line labeled *LAPACK with native BLAS* is the performance of the Cholesky factorization routine in LAPACK running on the native BLAS routines in ESSL. The MFlops graph provides "truth in advertising" — although the execution times of LAPACK and compiler-generated code with DGEMM are comparable, LAPACK achieves higher Mflops. This is because we replaced only one of several matrix multiplications in the blocked code by a call to DGEMM. On the positive side, these results, coupled with careful analysis of the compiler-generated code, show that the compiler-generated code has the right block structure. What remains is to make the compilation of inner loops (by unrolling inner loops, prefetching data and doing scalar replacement [1, 25]) more effective in the IBM compiler.

Figure 12 shows the performance of QR factorization using Householder reflections. The input code has poor performance, and it is improved somewhat by blocking. The blocked code was generated by blocking only columns of the matrix, since dependences prevent complete two-dimensional blocking of the array being factored. As in the case of Cholesky factorization, replacing loops that perform matrix multiplication with calls to DGEMM results in significant improvement, as shown by the line labeled *Matrix Multiply replaced by DGEMM*. This code beats the fully blocked code in LAPACK for matrices smaller than 200-by-200. The compiler-generated code uses the same algorithm as the

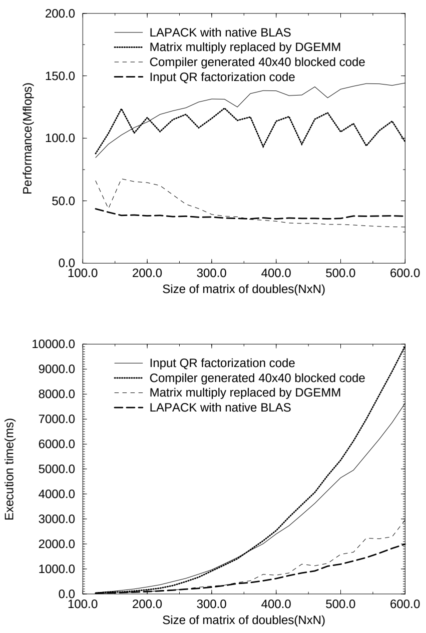


Figure 12: QR factorization on the IBM SP-2

"pointwise" algorithm for this problem; the LAPACK code on the other hand uses domain-specific information about the associativity of Householder reflections to generate a fully-blocked version of this algorithm [9].

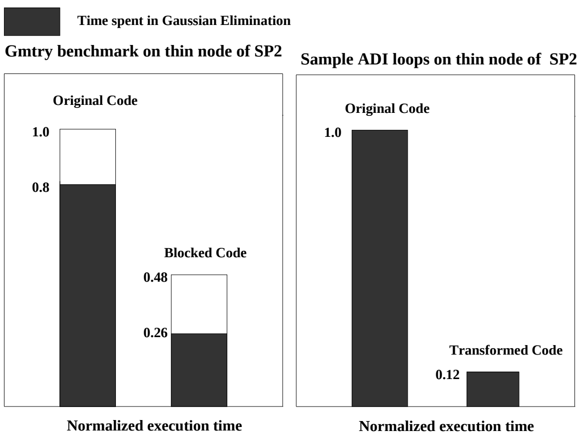


Figure 13: (i) Gmtry & (ii) ADI benchmarks on the IBM SP-2

Figure 13(i) shows the results of data shackling for the **Gmtry** kernel which is a SPEC benchmark kernel from *Dnasa7*. This code performs Gaussian elimination across rows, without pivoting. Data shackling blocked the array in both dimensions, and produced code similar to what we obtained in Cholesky factorization. As can be seen in this figure, Gaussian elimination itself was speeded up by a factor of 3; the entire benchmark was

```
do i = 2, n
  do k = 1, n
    S1: X(i,k) -= X(i-1,k)*A(i,k)/B(i-1,k)
  enddo
  do k = 1, n
    S2: B(i, k) -= A(i,k)*A(i,k)/B(i-1,k)
  enddo
enddo
```

(i) Input code

```
do t1 = 1, n
 do t2 = 1, n-1
  S1: X(t2+1,t1)-=X(t2,t1)*A(t2+1,t1)/B(t2,t1)
  S2: B(t2+1,t1)-=A(t2+1,t1)*A(t2+1,t1)/B(t2,t1)
 enddo
enddo
```

(ii) Transformed Code

Figure 14: Effect of fusion + interchange on ADI

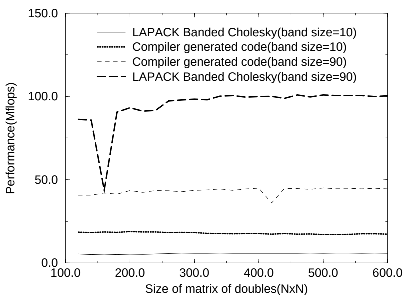


Figure 15: Banded Cholesky factorization on the SP-2

speeded up by a factor of 2.

Figure 14(i) shows the ADI kernel used by McKinley et al in their study of locality improving transformations [18]. This code was produced from FORTRAN-90 by a scalarizer. The traditional iteration-centric approach to obtain good locality in this code is to fuse the two k loops first, after which the outer i loop and the fused k loop are interchanged to obtain unit-stride accesses in the inner loop. The resulting code is shown in Figure 14(ii). In our approach, this final code is obtained by simply choosing B(i-1,k) as the data-centric reference in both S1 and S2 and blocking B into blocks of size 1x1. When an element of B is touched, all statement instances from both loop nests that touch this element must be performed; this achieves the effect of loop jamming. Traversing the blocks in storage order achieves perfect spatial locality, which achieves the effect of loop interchange after jamming. As shown in Figure 13(ii), the transformed code runs 8.9 times faster than the input code, when n is 1000.

Since shackling takes no position on how the remapped data is stored, the techniques described in Section 4 can be used to generate code even when the underlying data structure is reshaped. A good example of this is banded Cholesky factorization [15]. The banded Cholesky factorization in LAPACK is essentially the same as regular Cholesky factorization with two caveats: (i) only those statement instances are performed which touch data within a band of the input matrix, and (ii) only the bands in the matrix are stored (in column order), rather than the entire input matrix. In our framework, the initial point code is regular Cholesky factorization restricted to accessing data in the band, and the data shackling we used with regular Cholesky factorization is applied to this restricted code. To obtain the blocked code that walks over a data structure that stores only the bands, a data transformation is applied to the compiler generated code as a post processing step. As seen in Figure 15, the compiler generated code actually outperforms LAPACK for small band sizes. As the band size increases however, LAPACK performs much better than the compiler generated code. This is because, for large band sizes, LAPACK starts reaping the benefits of level 3 BLAS operations, whereas the xlf back-end (on the IBM SP-2) is once again not able to perform equivalent optimizations for loops in our compiler generated code which implement matrix multiply operations.

8 Ongoing Work

There are several unresolved issues with our approach. In this section, we discuss these open issues and suggest plausible solutions when possible.

As discussed in Section 4, a data shackle has three components - sets of cutting planes, shackled references and order of enumeration over blocks. In this paper, we have assumed that the specification of these components is given to the compiler. Automating our data-centric approach fully requires the compiler to determine these components. We are working on this problem, basing our approach on the following observations.

One approach is to implement a search method that enumerates over plausible data shackles, evaluates each one and picks the best. When there are multiple data shackles that are legal for a program, we need a way to determine the best one. This requires accurate cost models for the memory hierarchy, such as the ones developed by other researchers in this area [18, 22].

If the search space becomes large, heuristics may be useful to cut down the size of the search. Theorem 2 and the subsequent discussion suggests that to a first order of approximation, the orientation of cutting planes has little impact on performance, and can be selected to satisfy legality considerations alone. For all the benchmarks in the paper, we found that walking over the blocked array in top to bottom, left to right order was adequate. This order of enumerating blocks has great appeal because it is simple and because we believe that this is the natural order used by programmers. In general, of course, this order of traversing blocks may not be legal (triangular back-solve is an example), but we believe that in most of those cases, traversing the blocks bottom to top or right to left will be legal[4]. Another assumption in our approach so far is that a shackled reference makes a single sweep through the corresponding array. This is adequate for problems like matrix factorizations in which there is a definite direction to the underlying data flow of the algorithm, but it is obviously not adequate for problems like relaxation codes in which an array element is eventually affected by every other element. To solve this problem, we must make multiple passes over the blocked array. One possibility is the following: rather than perform all shackled statement instances when we touch a block, we can perform only those instances for which dependences have been satisfied. The array is traversed repeatedly till all instances are performed. Determination of good block sizes can also be tricky, especially for a multi-level memory hierarchy [10]. This problem arises even in handwritten code — in this context, library writers are exploring the use of training sets to help library code determine good block sizes [13]. We can adopt this solution if it proves to be successful.

We have presented data shackling as an alternative restructuring technology that avoids some of the problems of current control-centric approaches. However, it is unclear to us whether our approach can fully subsume loop transformation techniques. In the event that both these approaches are required for program restructuring, an important open question is to determine the interaction between them.

Finally, we note that there are programs for which handwritten blocked codes exploit algebraic properties of matrices. QR-factorization using Householder reflections is an example [15]. It is unclear to us whether a compiler could or even should attempt to restructure programs using this kind of domain-specific information. It is likely that the most plausible scenario is compiler blocking augmented with programmer directives for blocking such codes [9].

Acknowledgments: We would like to thank Rob Schreiber, Charlie van Loan, Vladimir Kotlyar, Paul Feautrier and Sanjay Rajopadhye for stimulating discussions on block matrix algorithms and restructuring compilers. This paper was much improved by the feedback we received from an anonymous "good shepherd" on the PLDI committee.

[4] This is similar to loop reversal.

References

[1] Ramesh C. Agarwal and Fred G. Gustavson. *Algorithm and Architecture Aspects of Producing ESSL BLAS on POWER2.*

[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, editors. *LAPACK Users' Guide. Second Edition.* SIAM, Philadelphia, 1995.

[3] Jennifer Anderson, Saman Amarsinghe, and Monica Lam. Data and computation transformations for multiprocessors. In *ACM Symposium on Principles and Practice of Parallel Programming*, Jun 1995.

[4] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

[5] David Bau, Induprakas Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghil. Solving alignment using elementary linear algebra. In *Proceedings of the 7th LCPC Workshop*, August 1994. Also available as Cornell Computer Science Dept. tech report TR95-1478.

[6] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? In *INTEGRATION, the VLSI Journal*, volume 17, pages 33–51. 1994.

[7] Steve Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing*, 1992.

[8] Steve Carr and R. B. Lehoucq. Compiler blockability of dense matrix factorizations. Technical report, Argonne National Laboratory, Oct 1996.

[9] Steven Carr and R. B. Lehoucq. A compiler-blockable algorithm for QR decomposition, 1994.

[10] L. Carter, J. Ferrante, and S. Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, April 1995.

[11] Michael Cierniak and Wei Li. Unifying data and control transformations for distributed shared memory machines. In *SIGPLAN 1995 conference on Programming Languages Design and Implementation*, Jun 1995.

[12] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In David W. Wall, editor, *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, volume 30(6) of *ACM SIGPLAN Notices*, pages 279–290, New York, NY, USA, June 1995. ACM Press.

[13] Jim Demmel. Personal communication, Sep 1996.

[14] Jack Dongarra and Robert Schreiber. Automatic blocking of nested loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee, May 1990.

[15] Gene Golub and Charles Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[16] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society.

[17] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 1993.

[18] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. In *ACM Transactions on Programming Languages and Systems*, volume 18, pages 424–453. July 1996.

[19] W. Pugh. A practical algorithm for exact array dependency analysis. *Comm. of the ACM*, 35(8):102, August 1992.

[20] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, October 1992.

[21] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *SIGPLAN89 conference on Programming Languages, Design and Implementation*, Jun 1989.

[22] Vivek 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.

[23] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *SIGPLAN 1991 conference on Programming Languages Design and Implementation*, Jun 1991.

[24] M. Wolfe. Iteration space tiling for memory hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[25] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.