

Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning

Amy W. Lim Shih-Wei Liao* Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

{aimee, sliao, lam}@cs.stanford.edu

ABSTRACT

Applicable to arbitrary sequences and nests of loops, affine partitioning is a program transformation framework that unifies many previously proposed loop transformations, including unimodular transforms, fusion, fission, reindexing, scaling and statement reordering. Algorithms based on affine partitioning have been shown to be effective for parallelization and communication minimization. This paper presents algorithms that improve data locality using affine partitioning.

Blocking and array contraction are two important optimizations that have been shown to be useful for data locality. Blocking creates a set of inner loops so that data brought into the faster levels of the memory hierarchy can be reused. Array contraction reduces an array to a scalar variable and thereby reduces the number of memory operations executed and the memory footprint. Loop transforms are often necessary to make blocking and array contraction possible.

By bringing the full generality of affine partitioning to bear on the problem, our locality algorithm can find more contractable arrays than previously possible. This paper also generalizes the concept of blocking and shows that affine partitioning allows the benefits of blocking be realized in arbitrarily nested loops. Experimental results on a number of benchmarks and a complete multigrid application in aeronautics indicates that affine partitioning is effective in practice.

1. INTRODUCTION

Affine partitioning is a program transform framework that is applicable to arbitrarily nested loops and affine array accesses[17, 18]. Constructs such as conditional statements and non-affine accesses are generally handled by treating

*Currently with Intel Research in Santa Clara, California.

them conservatively. In this model, instances of a statement are identified by the loop index values of their surrounding loops, and affine expressions are used to map the original loop index values to new index values in the transformed code. All possible combinations of many previously proposed transforms including unimodular transforms, fusion, fission, reindexing, scaling, and statement reordering[4] can be expressed as affine transforms. We have developed an algorithm that finds the best affine partition that maximizes the degree of parallelism in a program while minimizing the degree of synchronization[17]. We have also shown that affine partitioning is useful for minimizing communication between processors[16].

However, to get scalable performance on a multiprocessor, or even high performance on a uniprocessor, we must optimize for the memory hierarchy on the machine. Many optimizations have been invented to improve an application's memory subsystem performance. Two particularly important techniques are array contraction and blocking.

Array contraction[6, 8, 14, 20] was invented to handle array constructs in languages such as Fortran 90 and HPF effectively. The front end of a compiler would typically translate each array operation into a loop and use work arrays to hold partial terms in array expressions. Direct execution of such programs performs poorly on architectures with caches. Not only are there more memory operations, the working set of the program gets bigger, thus reducing the effectiveness of the cache. By fusing these loops together so that every element of the work array is defined and used within the same iteration, the same scalar variable can be used to hold the values of the different array elements consecutively. Allocating the scalar variable to a register reduces the number of memory operations executed and the working set size of the program.

Blocking is a technique well known to numerical analysts[9]. For data locality, it is preferable to decompose a matrix computation into submatrix calculations, as opposed to row and column operations. The sizes of the submatrices are chosen so that data fetched into a faster level of memory hierarchy are reused before they are displaced. This optimization can be applied recursively to improve the performance of registers, caches, physical memories, or the translation lookaside buffers.

This paper shows that array contraction and blocking can both benefit greatly from affine partitioning. Our parallelization technique based on affine partitioning is directly

applicable to array contraction. The algorithm finds the best affine transform that separates the computation of a program into the largest number of independent threads. As all related operations are captured in the same thread, running the threads sequentially separates the live ranges of the elements in the work arrays, making it possible to reduce the arrays to scalar variables. It also improves temporal locality among related operations that might be scattered over many loops in the source program. As affine partitions are equivalent to all combinations of unimodular transforms, fusion, fission, reindexing, scaling, and statement reordering, our algorithm is more effective than previous techniques which use only a subset of such transforms.

Various attempts have been made to generalize blocking beyond its original domain of perfect loop nests. In the original formulation of blocking, unimodular transforms are first applied to a perfectly nested loop to make it *fully permutable*[23]. A loop nest is fully permutable if all the loops can be arbitrarily permuted without changing the original program semantics. Once a loop nest is made fully permutable, it is trivial to block the loop nest. An n -dimensional fully permutable loop nest also has at least $n-1$ degrees of pipelining parallelism. We have developed an algorithm that can extract from an arbitrarily nested program the largest outermost fully permutable loop nest using affine transforms[17]. Ahmed et al. have also developed a heuristic procedure that finds outermost fully permutable loop nests in two steps: the first turns arbitrary loop nests into a high-dimensional perfect loop nest[1, 2], and then unimodular techniques are used to create fully permutable loop nests. The approach of just blocking fully permutable loop nests, however, is inadequate. Most programs cannot be made into one fully permutable loop nest but can still benefit from the concept of blocking.

By going back to first principles, we generalize the notion of blocking to make it directly applicable to arbitrarily nested loops. Fundamentally, blocking can be viewed as an interleaving of a number of parallel or pipelinable threads to allow reuse of common data. The threads may share common read-only data, or they may read from or write to the same cache line. Perfect loop nests represent the special case where each parallel or pipelinable thread contains inner loops; blocking here means that the iterations of the inner loops from different threads are interleaved. In the case where threads contain multiple statements, blocking interleaves the execution of each statement from different threads. Blocking can thus be generalized as a distribution of a portion of the outer loop containing parallel or pipelinable threads over inner statements or inner loops. With this generalization, it is not necessary to convert loops into perfect nests before the benefits of blocking can be realized.

There is a very interesting interaction between array contraction and blocking. Whereas blocking tries to interleave independent threads to enhance locality, an array contraction algorithm tries to “de-interleave” independent threads so that intermediate data can be consumed immediately. Whereas blocking may require variables containing intermediate data be expanded by the block size, array contraction tries to reduce the dimension of work arrays to minimize the computation’s data footprint. Our algorithm combines the advantages of the two techniques by first separating the computation into threads to enhance the temporal locality of intermediate results, contracting the work arrays where pos-

sible, followed by a judicious interleaving of the independent threads to increase data locality, and expanding variables to small arrays where necessary.

Our locality optimization reuses the same basic algorithms that have previously been developed for parallelization. By transforming the code to contain the largest number of outermost parallel and pipelinable threads, these algorithms put the code in a canonical form where all the code motion opportunities are exposed. All the clever program transforms are performed by these algorithms. It is rather straightforward to implement array contraction and generalized blocking for programs in the canonical form. The commonality between parallelization and locality optimization reduces the implementation effort.

The organization of the rest of the paper is as follows. We start with a short introduction to affine partitioning. Sections 3 and 4 discuss array contraction and blocking, respectively. Section 5 discusses the interactions between the two optimizations and present an algorithm that integrates array contraction with blocking. Section 6 presents some experimental results to show the effectiveness of data locality optimizations based on affine partitioning. We close with a comparison with related work and some concluding remarks.

2. AFFINE PARTITIONING

Due to space constraints, we will present here only those salient features of affine partitioning that are relevant to this paper. Readers are referred to previous publications for more information[17, 18]. In affine partitioning, an iteration instance is identified by the value of the loop indices in enclosing loops. An affine partitioning scheme consists of affine mappings, one for each operation in the program, from the original index variables to values of index variables in the transformed code. Operations with common enclosing loops share the same loop index variables. It is possible to use affine partitions to express all combinations of unimodular transforms, fusion, fission, reindexing, scaling and statement reordering.

We have developed an algorithm that finds the optimal affine transform that maximizes parallelism in a program while minimizing synchronization. Our algorithm finds two kinds of affine partitions: space partitions and time partitions. The space partitioning algorithm places operations belonging to different independent threads in different space partitions. If the algorithm succeeds in finding n independent solutions to the space partitioning problem, it is possible to create n outermost parallel loops whose iterations represent the partitions found. When a program does not have any independent threads, our algorithm divides the computations into time partitions such that completing the partitions in order will satisfy the data dependences. If there are n independent solutions to the time partitioning problem, we can rewrite the code as having n outermost loops that are fully permutable. An n -deep fully permutable loop nest can be pipelined to create $n-1$ degrees of parallelism. A fully permutable loop nest is also blockable.

The space and time partitioning problems are solved in a similar manner. The partitioning constraints are derived directly from the array access functions in the program and the loop bounds. All affine expressions of loop indices and symbolic constants are represented faithfully in the constraints without any loss of precision. By using the affine form of the Farkas lemma[7, 21], the problem reduces to solving a set of

linear inequalities. The partitioning algorithms can be applied recursively to inner loops to find independent threads at all levels.

3. ARRAY CONTRACTION

Array contraction has been shown to be a useful technique for optimizing array constructs and legacy scientific codes that have been tuned for vector machines. To aid vectorization, programmers often expand scalar variables into arrays so that consecutive iterations in the innermost loop operate on consecutive memory locations. Whereas simple applications of loop fusion might be adequate for array constructs in languages, sophisticated program transforms can expose many more opportunities for array contraction in legacy codes.

Loop fusion has been shown to be useful for array contraction[6, 8, 14, 20]. For example, suppose the source is an array expression $A = B + C + D$ where each array is an N -element array. The front end of a compiler would typically generate the loops shown in the source program below. Fusing the two loops makes it possible to reduce the temporary array T to a scalar variable, t , which can now be register allocated.

(Example 1) Source program	Transformed code
Do I = 1, N	Do I = 1, N
T(I) = B(I) + C(I)	t = B(I) + C(I)
Enddo	A(I) = t + D(I)
Do I = 1, N	Enddo
A(I) = T(I) + D(I)	
Enddo	

This is an important performance optimization. The processor executes fewer memory operations, which are especially costly if they miss in the cache. The working set also becomes smaller, allowing other memory references to enjoy a greater hit rate.

3.1 Our Approach

Our approach to array contraction is to attempt to reverse the array expansion process. Our algorithm starts by extracting independent threads from the source program using affine partitioning. By executing the independent threads one after the other, the live ranges of these temporary variables no longer overlap. In the case where each thread only uses one element of the array, the array can be reduced to a scalar variable.

To illustrate this approach, consider as an example the program *btrix*, a vectorized block tri-diagonal solver in the SPEC92 NASA7 benchmark. An excerpt of the 160-line routine is shown in Figure 1(a). Even though the nesting depth of the code varies from loop to loop, affine partitioning has no problem picking out all the independent threads in this program. Specifically, the algorithm correctly determines that all iterations with the same L index values are related. That is, it is possible to create an outermost loop containing all independent iterations, where the l th iteration includes all the computations in the l th iteration of all the original loops with index L . The fact that the programmer happens to use the same index variable L for all the related iterations in different loops in this example makes it easy to explain the partitioning results, but it is not used by the algorithm in any way. The transformed code, with an outermost loop visiting each of the partitions sequentially, is shown in Figure 1(b).

(a) Source program:

```

DO 100 J = JS,JE
  IF(J.EQ.JS) GO TO 3
  DO 3 M = 1,5
    DO 3 N = 1,5
      DO 3 L = LS,LE
        B(M,N,J,L) = B(M,N,J,L) - B(1,N,J-1,L) ...
3    CONTINUE
  DO 20 L = LS,LE
    L11(L) = 1.DO / B(1,1,J,L)
20  CONTINUE
  DO 25 L = LS,LE
    ... = B(1,2,J,L)*L11(L)
25  CONTINUE
  IF(J.EQ.JS) GO TO 34
  DO 33 M = 1,5
    DO 33 L = LS,LE
      S(J,K,L,M) = S(J,K,L,M) - S(J-1,K,L,1) ...
33  CONTINUE
34  DO 35 L = LS,LE
    D1 = S(J,K,L,1)*L11(L)
    S(J,K,L,1) = D1 - S(J,K,L,2) ...
35  CONTINUE
  IF(J.EQ.JE) GO TO 100
  DO 40 N = 1,5
    DO 40 L = LS,LE
      C1 = C(1,N,J,L)*L11(L)
      B(1,N,J,L) = C1 - B(2,N,J,L) ...
40  CONTINUE
100 CONTINUE

```

(b) Transformed code:

```

DO 100 L = LS,LE
  DO 100 J = JS,JE
    IF(J.EQ.JS) GO TO 3
    DO 3 M = 1,5
      DO 3 N = 1,5
        B(M,N,J,L) = B(M,N,J,L) - B(1,N,J-1,L) ...
3    CONTINUE
    L11(L) = 1.DO / B(1,1,J,L)
    ... = B(1,2,J,L)*L11(L)
    IF(J.EQ.JS) GO TO 34
    DO 33 M = 1,5
      S(J,K,L,M) = S(J,K,L,M) - S(J-1,K,L,1) ...
33  CONTINUE
34  D1 = S(J,K,L,1)*L11(L)
    S(J,K,L,1) = D1 - S(J,K,L,2) ...
    IF(J.EQ.JE) GO TO 100
    DO 40 N = 1,5
      C1 = C(1,N,J,L)*L11(L)
      B(1,N,J,L) = C1 - B(2,N,J,L) ...
40  CONTINUE
100 CONTINUE

```

Figure 1: An excerpt of *btrix* from SPEC92 NASA7

This algorithm also illustrates the ability of our algorithm to handle conditional statements. Because the predicates in the program are affine expressions of loop indices and symbolic constants, they can be captured precisely in our transformation constraints without any loss of precision. Once the code has been transformed as shown in Figure 1(b), it is easy to see that array $L11$ can now be contracted since only one element of the array is defined and used in each iteration. With this approach, 25 arrays were contracted in the entire *btrix* routine.

3.2 The Array Contraction Algorithm

By using affine partitioning to restructure the code, our array contraction algorithm appears simple yet powerful. In finding all the independent threads in the program, array partitioning “fuses” together loops with different nesting levels, applying unimodular transforms, fission, reindexing, scaling and statement reordering where necessary. As a re-

sult, it discovers many more opportunities for array contraction than were previously possible.

Algorithm 3.1 Array Contraction

1. Find independent threads in a program using affine partitioning.
2. Identify those arrays whose regions of data accessed in each iteration have less dimensions than the original arrays. This can be done easily by examining the access patterns of the arrays.
3. For each array identified in Step 2, determine if the live range of each array element is confined to a single iteration using an interprocedural context-sensitive array liveness algorithm[15].
4. Generate code to execute the independent threads sequentially and contract arrays where appropriate.

4. BLOCKING

We first explain the need to generalize the concept of blocking to handle imperfect loop nests by way of an example. Next we discuss the concept of generalized blocking for programs with independent partitions. Finally, we show how programs without independent partitions are handled.

4.1 Motivation of Generalized Blocking

The essence of blocking is to create a set of inner loops, also known as a tile, whose iteration counts are carefully controlled to ensure that data reused in the tile fit into the level of memory hierarchy of interest. A loop nest is blockable if it is fully permutable. Blocking turns an n -deep loop nest into a $2n$ -deep nest; the outer n loops control the order in which the tiles are visited, and the inner n loops control the order in which the iterations belonging to a tile are visited. The outer loops can be arbitrarily permuted and so can the inner ones.

There have been various attempts to reduce the problem of blocking for arbitrary loop nests to the problem of blocking for perfect loop nests. While it is sometimes possible to turn loops into one fully permutable loop nest, such an approach is too limited for the general case. Many programs cannot be reduced to a single loop nest but can benefit from the concept of blocking. Consider the following example:

```
(Example 2) Do I = 1, M [L1]
             X(I,1) = Y(I) [S1]
             Do J = 2, N [L2]
               X(I,J) = f( X(I,J-1) ) [S2]
             Enddo
             Do K = 1, N-1 [L3]
               X(I,N-K) = g( X(I,N-K+1) ) [S3]
             Enddo
             Enddo
```

Loop L2 computes the values of a row of X from left to right, then loop L3 takes the results from L2 and updates the same row of data from right to left. In other words, loop L2 must finish executing on a row of data before loop L3 can be executed.

Assuming that the data are organized in column major order, as is the case in Fortran, every iteration in the inner loops fetches a different cache line. The code executes more efficiently if the computation is re-ordered so that the inner loops operate on all the data in the cache lines as they are brought in. This can be achieved with the following code:

```
Do II = 1, M, B
  Do I = II, II + B-1
    X(I,1) = Y(I) [S1]
  Enddo
  Do J = 2, N [L2]
    Do I = II, II + B-1
      X(I,J) = f( X(I,J-1) ) [S2]
    Enddo
  Enddo
  Do K = 1, N-1 [L3]
    Do I = II, II + B-1
      X(I,N-K) = g( X(I,N-K+1) ) [S3]
    Enddo
  Enddo
Enddo
```

The code above represents a simple extension of blocking, but none of the existing blocking techniques can produce this code. Blocking with unimodular transforms does not apply because the loops are not perfectly nested. Algorithms that only block fully permutable loop nest would also fail as loops J and K cannot be combined into one fully permutable loop nest[1, 2]. Data shackling cannot change the order of the execution[13]. Iteration space slicing cannot get the blocking effect to improve spatial locality[19].

4.2 Handling Sets of Independent Threads

In this section, we focus on arbitrarily nested loops that can be broken up into independent threads. It is useful to consider this subset separately even if we are not interested in parallelism because such loops have more degrees of freedom in code motion. Specifically, any interleavings of the independent threads are legal as long as the relative ordering of the operations in each thread is maintained.

THEOREM 4.1. *Let P be a program consisting of n independent threads T_1, \dots, T_n , where thread T_i is a sequence of operations $T_{i1}, T_{i2}, \dots, T_{im_i}$. Let $\rho(o)$ denote the time when operation o is executed. Schedule ρ is legal if*

$$\forall j < k \quad \rho(T_{ij}) < \rho(T_{ik})$$

4.2.1 Two Thread-Interleaving Primitives

Blocking is a stylized form of thread interleaving. Threads nested under the same outer loops are interleaved to exploit the fact that they often share the same data or the same cache line. Just as there are many ways to block a computation in a perfect loop nest, there are many different ways to block independent threads.

To support the full generality of blocking, we model every instruction nested in n loops by its own n -dimensional iteration space; instructions sharing an outer loop share the same axes. For example, Figure 2(a) shows the iteration spaces for each of the three instructions in Example 2, assuming for the sake of simplicity that the variables M and N are 8 and 5, respectively. Each dynamic instance of an instruction is represented by a node in the figure; statement S1 is represented by a column of nodes, whereas statements S2 and S3 are each represented by a 2-dimensional space. The original sequential execution order is shown by the arrows in Figure 2(a). It is easy to see from the code that all the iterations of the outermost loop are independent of each other, as they operate on separate rows of the matrix X. By Theorem 4.1, it is legal to execute the program in any order as long as all the operations belonging to the same row in the figure retain their relative ordering. Figures 2(b-d) represent examples of legal interleavings for the code.

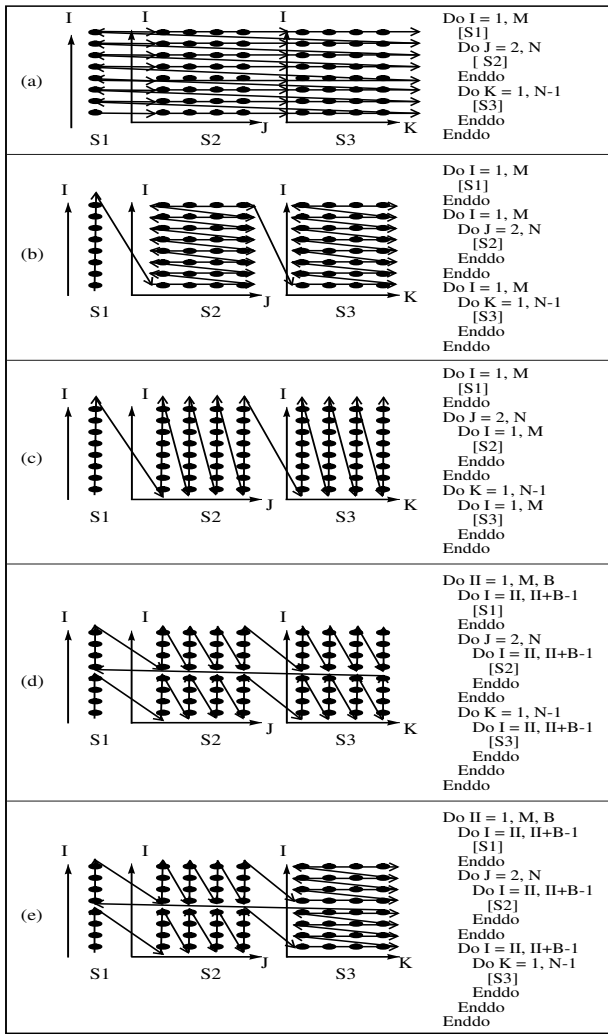


Figure 2: Legal interleavings and transformed codes for Example 2 with $M=8, N=5$. Each row of operations forms an independent thread; arrows connecting the operations show their execution order. (a) Sequential execution. (b) Statement interleaving. (c) Iteration interleaving of (b). (d-e) Interleavings with stripmining.

A program with a set of independent partitions can be represented by an outer parallel loop whose loop body can consist of an arbitrary nesting of loops and sequences of loops and statements. All the legal interleavings can be derived by applying two primitives recursively: statement interleaving or iteration interleaving. These primitives are special cases of Theorem 4.1.

COROLLARY 4.2. Statement Interleaving.

Let T_1, T_2, \dots, T_n be n independent threads whose code consists of statements S_1, S_2 . Denoting the computation of statement S_j by thread T_i as $T_{i,j}$, the execution sequence $T_{1,1}, T_{2,1}, \dots, T_{n,1}, T_{1,2}, T_{2,2}, \dots, T_{n,2}$ is a legal interleaving.

COROLLARY 4.3. Iteration Interleaving.

Let T_1, T_2, \dots, T_n be n independent threads whose code consists of a loop with iterations I_1, I_2, \dots, I_m , where m is the number of iterations in the loop. Denoting the computation

of iteration I_j by thread T_i as $T_{i,j}$, the execution sequence $T_{1,1}, T_{2,1}, \dots, T_{n,1}, T_{1,2}, \dots, T_{n,2}, \dots, T_{1,m}, \dots, T_{n,m}$ is a legal interleaving.

It is interesting to note that these two corollaries from the same theorem correspond to two well-known loop transformations: loop distribution and loop permutation. The code transformation schema are shown below:

1. Statement interleaving

(a) Source program (b) Transformed code

```

Do I = 1, N
[S1]
[S2]
Enddo
Do I = 1, N
[S1]
[S2]
Enddo

```

2. Iteration interleaving

(a) Source program (b) Transformed code

```

Do I = 1, N
Do J = 1, M
[S]
Enddo
Do I = 1, M
Do J = 1, N
[S]
Enddo
Enddo

```

Figures 2(b) shows an application of statement interleaving. Once interleaving is applied to the outermost loop, it creates in the inner scope a set of independent partitions, which can be further interleaved. Applying iteration interleaving to the result in Figure 2(b) yields Figure 2(c). Stripmining is often used with interleaving so as to limit the size of the tile. Figures 2(d) and 2(e) show combinations of stripmining and interleaving. They also illustrate that interleaving decisions can be made independently for each statement or loop. For simplicity, M is assumed to be divisible by B .

4.2.2 A Generalized Blocking Algorithm

We define *generalized blocking* as a combination of stripmining a loop with independent iterations and applying one or more interleaving primitives recursively to the loop body. Our generalized blocking algorithm to improve data locality first creates loops with independent iterations, then identifies the source of reuse, and applies interleaving primitives selectively to exploit the identified reuse.

Algorithm 4.1 Generalized Blocking for Programs with Independent Partitions

1. Apply affine partitioning recursively to find all the independent partitions at all levels in the program.

This step puts the code in a canonical form and minimizes the algorithm's dependence on how the code was written. This step creates outermost parallel loops, but they will be stripmined and moved inwards if they carry locality in Step 3.

2. Identify data locality in the transformed program.

Use Wolf and Lam's algorithm to analyze the affine array index expressions in the program to derive the dimensions in the iteration space that carry reuse[23]. The algorithm determines the iteration subspace that may share the same data or cache lines by finding the null space of the access functions. For each statement, identify those outer parallel loop dimensions that have data reuse.

3. Create the innermost tile to include all parallelizable loops with reuse.

Stripmine all parallel loops that carry any array reuse for any statement. Distribute the stripmined loops into its components recursively until they enclose the statements that carry reuse. In the case where the inner loops carry no reuse, do not stripmine the innermost parallel loop but simply distribute that entire loop into its components.

For Example 2, Step 1 determines that the outermost loop is parallelizable without any further optimization. Step 2 determines that all the loop dimensions carry reuse. The outer loop also carries reuse since iterations (I, J) , $(I+1, J)$ write to contiguous locations $X(I, J)$ and $X(I+1, J)$, given the assumption of a column major layout. The inner loop L2 carries reuse because iteration (I, J) uses the data produced in iteration $(I, J-1)$. Step 3 stripmines the outer loop and distributes it recursively over all the statements and inner loops. The new execution ordering and transformed code are shown in Figure 2(d).

All the sophisticated loop transformations are performed in Step 1 of this algorithm; Steps 2 and 3 are straightforward. This is particularly attractive from a software engineering perspective since the implemented algorithm can be used for both parallelization and locality optimization.

4.3 Blocking Fully Permutable Loop Nests

As discussed in Section 2, we have developed two affine partitioning algorithms. The space partitioning algorithm finds independent threads and the time partitioning algorithm finds pipelinable parallelism. To achieve the latter, the affine partitioning transforms arbitrary loop nests and sequences to create the largest outermost fully permutable loop nest. Fully permutable loop nests have pipelinable parallelism and are also blockable.

To optimize for data locality, we wish to create an *innermost* fully permutable loop nest that carries data reuse. Our affine partitioning technique, on the other hand, produces the largest *outermost* fully permutable loop nest. In the case where the outermost fully permutable loop nest encompasses all loops, the outermost nest is also the innermost. For such codes, our time partitioning algorithm succeeds in finding the best transform for locality.

We will illustrate with an example how our time partitioning algorithm can create fully permutable loop nests out of imperfectly nested loops; readers are referred to our previous publication for further information[17]. The original code for cholesky decomposition (row version) is shown below:

```

Do I = 1, N
  Do J = 1, I-1
    Do K = 1, J-1
      A(I, J) = A(I, J) - A(I, K) * A(J, K)
    Enddo
    A(I, J) = A(I, J) / A(J, J)
  Enddo
  Do L = 1, I-1
    A(I, L) = A(I, L) - A(I, L) * A(I, L)
  Enddo
  A(I, I) = SQRT( A(I, I) )
Enddo

```

The above imperfectly nested loop consists of 4 statements, all nested in different loops. Our affine time partitioning algorithm puts all the statements, including those

originally nested in 1 and 2-deep loops, into a 3-deep fully permutable loop nest shown below. The algorithm finds, for each statement, an affine mapping from the original iteration space to the new iteration space such that all the dependence constraints are satisfied. The affine mappings found are included in the code below. The code generation routine guards the execution of the operations with the original loop bounds to ensure that the new programs execute only operations that are in the original code.

```

Do I2 = 1, N
  Do J2 = 1, I2
    Do K2 = 1, I2
      // Mapping: I2 = I, J2 = J, K2 = K
      if ((J2<I2) && (K2<J2))
        A(I2, J2) = A(I2, J2) - A(I2, K2) * A(J2, K2)
      // Mapping: I2 = I, J2 = J, K2 = J
      if ((J2==K2) && (J2<I2))
        A(I2, J2) = A(I2, J2) / A(J2, J2)
      // Mapping: I2 = I, J2 = I, K2 = L
      if ((I2==J2) && (K2<I2))
        A(I2, I2) = A(I2, I2) - A(I2, K2) * A(I2, K2)
      // Mapping: I2 = I, J2 = I, K2 = I
      if ((I2==J2) && (J2==K2))
        A(K2, K2) = SQRT( A(K2, K2) )
    Enddo
  Enddo
Enddo

```

Given a fully permutable loop nest like the one above, it is trivial to apply blocking to improve the code's data locality. Index-set splitting is applied at the end to eliminate the conditional guard calculations from the loops.

5. PUTTING IT ALL TOGETHER

There is an interesting interaction between blocking and array contraction. Array contraction is the optimization of pulling independent threads apart so as to minimize the live ranges of work arrays. Blocking is the technique of interleaving independent threads to take advantage of the data reuse between them. In fact, when we interleave multiple threads, we may have to turn scalar variables into array variables. Fortunately, blocking only requires scalar variables be expanded by the size of the block. It is often sufficient to get most of the benefit of blocking by using a small block size. Thus, we can combine these techniques in the following steps: find and serialize the independent threads, contract the arrays, interleave the threads by blocking, and expand the scalar variables where necessary by the block size.

Algorithm 5.1 Optimize data locality using affine partitioning, array contraction, and generalized blocking.

1. Apply the following to the whole program, then repeat by applying to the computation inside the parallelizable and fully permutable loops found in the last iteration:
 - (a) Separate statements that operate on different data into different equivalence classes. Apply the following steps to each set of statements.
 - (b) Use affine partitioning to extract the independent partitions in the computation. Transform the code to execute the partitions sequentially as iterations of an outermost parallelizable loop nest. In the case where there are multiple parallel loops, place the one with read-only data reuse in the innermost position of the nest. If the algorithm is

	Program Description				Integrated Affine Algorithm			
	Problem Size	# of Lines	# of Loops	# of Arrays	Number of Arrays		Number of Loops	
					Contracted	Expanded	After Partitioning	Blocked
cholesky	1000 × 1000	25	4	1	0	0	3	3
btrix (nasa7)	30 × 30 × 30 × 5	160	14	29	25	0	8	0
cholsky (nasa7)	250 × 4 × 40	55	18	3	1	1	10	2

Table 1: Characteristics of the benchmark kernels and their transformed codes.

successful in finding independent partitions, skip the next step.

- (c) Divide the program dependence graph into strongly connected components. Use the affine space and time partitioning algorithms to find, respectively, the independent and pipelinable partitions in each component. We combine the components greedily if they share reuse and if the combination does not eliminate all parallelism.
2. Use interprocedural array liveness results to identify contractable arrays and replace them with scalar variables.
3. Apply generalized blocking to improve locality across parallel or pipelinable threads. We place into the innermost tile the stripmined portions of all the outer parallelizable loops and all the loops in the innermost fully permutable nest that carry reuse. Blocking an outer fully permutable nest is advisable to exploit higher levels of memory hierarchies, such as the third-level cache or the physical memory. Scalar variables may be expanded by the block size to implement blocking.

The data locality algorithm described above can easily be combined with an affine-transform based parallelization algorithm to create multiprocessor codes with good data locality.

Algorithm 5.2 Optimize parallelism and data locality for multiprocessors.

1. Use affine partitioning to find parallelism in a sequential program and create an SPMD program that exploits the coarsest granularity of parallelism[16]. Note that pipelined partitions found in the parallelization step would already be blocked by the parallelization algorithm to reduce interprocessor communications.
2. Apply the locality optimization algorithm (Algorithm 5.1) to the SPMD program produced in Step 1.

6. EXPERIMENTAL RESULTS

To evaluate the effectiveness of our parallelization and locality optimizations, we applied our algorithm to a number of well-known kernels and a 5000-line application that solves three-dimensional Euler equations using a multigrid method. We have implemented the affine partitioning algorithm in the SUIF2 compiler[3]. We used the algorithm to derive the desired affine transforms for the programs automatically and applied array contraction, blocking, and code generation algorithms manually to produce the final output. The kernels were run on a Compaq Turbolaser with 8 300-MHz 21164 processors; the larger application was run on an SGI Origin machine with 32 195-MHz MIPS R10000 processors.

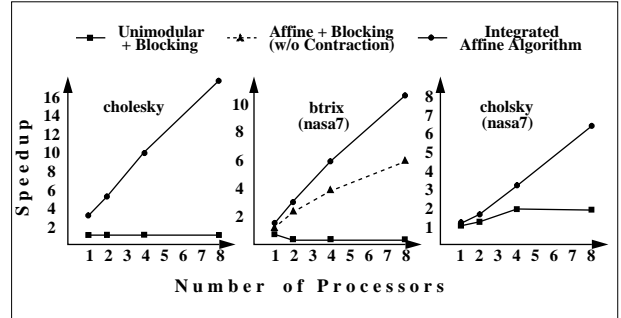


Figure 3: Speedups for the benchmark kernels.

6.1 Performance of Kernel Codes

We have chosen for this study a set of kernels that cannot be optimized well using unimodular transformation techniques[22, 23]. (Programs that can be optimized by unimodular transforms will also benefit from affine partitioning, since the former is subsumed by the latter). Table 1 lists the Fortran kernels used in our experiment together with some of their characteristics before and after optimization. Figure 3 shows the performance obtained using our algorithm and an algorithm based on unimodular transforms and blocking[22]. Performance is measured as speedup over the sequential version without any code transformation for locality or parallelism. Because unimodular transforms and blocking are applicable only to perfect loop nests[22, 23], they fail to improve these kernels for either uniprocessors or multiprocessors. Our algorithm achieves between 1.1 and 3.1 times speedups on a uniprocessor across the three kernels tested, and between 6.3 to 17.5 times speedups on a multiprocessor with 8 processors.

Cholesky. None of the loops in Cholesky, shown in Section 4.3, are parallelizable. Unlike unimodular transforms, affine partitioning succeeds in transforming the imperfect loop nest into one fully permutable loop nest. Operating on $O(n^2)$ data, this $O(n^3)$ loop nest has significant reuse along all loop dimensions. Our locality algorithm triples the speed of the program by blocking all three dimensions of the fully permutable loop nest. This algorithm is also effective for multiprocessors. Even though the program has only pipelined parallelism, our optimized code achieves a super-linear speedup of 17.5 on 8 processors, partially due to better data locality on the individual processors. Processors need only to synchronize around block boundaries.

Btrix. Btrix is a vectorized block tri-diagonal solver in the SPEC92 NASA7 benchmark[5]. As discussed in Section 3, affine partitions separate the program into completely independent threads, exposing many array contraction opportunities. Our optimization algorithm speeds btrix up by 50% on a single processor, and the performance scales almost linearly across 8 processors. The contraction of 25

	<i>Description</i>				<i>Integrated Affine Algorithm</i>			
	Calculation Performed	# of Lines	# of Loops	# of Arrays	<i>Number of Arrays</i>		<i>Number of Loops</i>	
					Contracted	Expanded	After Partitioning	Blocked
PSMOO	Smoothing in 3-D	185	30	5	3	2	11	2
DFLUX	Dissipation in 3-D	160	48	15	6	5	20	2
EFLUX*	Compute Euler flux	155	10	29	0	0	10	0
STEP	Compute spectral radii	180	33	13	2	0	24	0

*with code duplication

Table 2: Characteristics of four important subroutines in FLO87 and their transformed codes.

arrays improves the speedup on 8 processors from a factor of 5.9 to 10.6.

Cholsky. Cholsky is another SPEC92 NASA7 benchmark which computes multiple cholsky decompositions in a vector processing style[5]. Data locality for two of the three arrays referenced in the sequential code is already near optimal. By improving the locality of the third array, our generalized blocking algorithm improves the uniprocessor performance by 10%. It achieves a speedup of 6.3 on 8 processors by extracting from the same loops in the source program both coarse-grain parallelism for the outer loops and spatial locality for the inner loops.

6.2 A Case Study: Analysis of Transonic Flow

Jameson's transonic flow solver, known as FLO87, is a 5000-line Fortran program that uses a multigrid algorithm to solve three-dimensional Euler equations[10]. The input data is a model of an ONERA M6 wing, a well-known test case in the aeronautics community. Our first experiment provides an analysis of how the compiler optimizes the code; the second experiment shows how the optimized code scales to a larger number of processors and how the performance varies with different multigrid levels.

The original FLO87 program has already been tuned for locality so that most of the data accessed in inner loops are contiguous. We did not expect to improve the program's uniprocessor performance by much; the challenge was whether the aggressive transformations applied to parallelize the code would ruin the locality in the original program.

6.2.1 Analysis of the Program

In the first experiment, the application was run with five multigrid levels on a $192 \times 32 \times 48$ cell mesh. To gain insight into how our algorithm speeds up the code, we analyzed the four most important subroutines in the program, which together take up about 70% of the execution time. Table 2 includes a basic description and some characteristics of these subroutines. The performance results of the individual routines as well as the entire application are shown in Figure 4. The figure also includes a comparison between the speedup achieved using our integrated algorithm versus a direct parallelization of the loops in the source program without transformations.

PSMOO. Collecting together all the related operations from a set of 30 irregularly nested loops in the source program, our algorithm succeeds in putting almost all the computation in the program in two nested outer parallel loops. The number of loops is reduced from 30 to 11, indicating the extent to which the code has been transformed. With this transformation, the algorithm finds 3 out of the 5 arrays to be contractable. The generalized blocking algorithm restores the spatial locality in the innermost loops by inter-

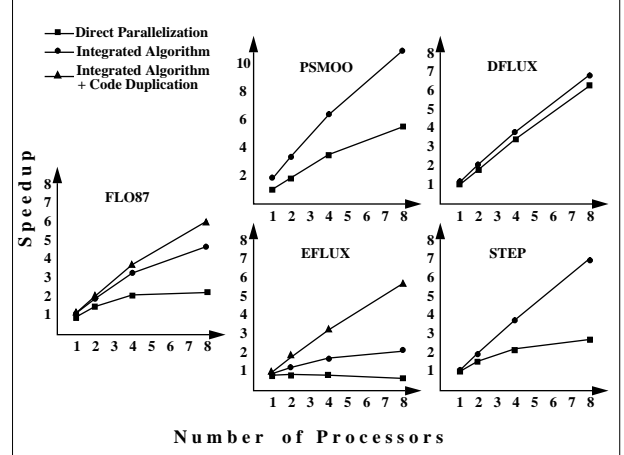


Figure 4: Speedups for FLO87 and four of its subroutines.

leaving the execution of the iterations in the outer parallel loops and expanding two out of the three contracted arrays by the size of the block. The uniprocessor performance speeds up by 81%; the multiprocessor version runs 6.1 and 11.1 times faster on an 8-processor machine when compared to the optimized uniprocessor code and the original uniprocessor code, respectively. In contrast, direct parallelization achieves only a 5.6 times speedup on 8 processors.

DFLUX. Our algorithm transforms the 48 loops in this program into 20 loops, contracts 6 out of the 15 arrays and expands 5 of them by the block size for spatial locality. This results in 10% speedup on a uniprocessor and a 6.9 times speedup on 8 processors compared to the original uniprocessor code and the original uniprocessor code, respectively. The performance obtained represents a small improvement over a direct parallelization of the original code.

EFLUX. Unlike all the other subroutines described so far, our transformation model can parallelize EFLUX only by expanding five linear arrays into large 3-dimensional arrays. This parallelization scheme degrades the uniprocessor performance by 10% and only yields a 2.1 times speedup on 8 processors. It is possible to parallelize this code without array expansion if a small amount of the computation is duplicated to eliminate a number of data dependences in the program. If code duplication is applied, our locality algorithm achieves a 5.7 speedup on 8 processors. This illustrates that other kinds of transformations, such as code duplication, can complement affine partitioning.

STEP. Our algorithm converts an imperfect loop nest in the program, consisting originally of eight loops, into one 3-deep fully permutable loop nest. Array contraction is applied to two arrays and no blocking is necessary because the

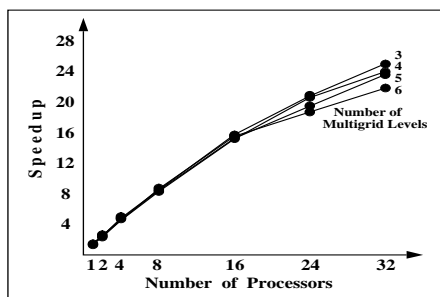


Figure 5: Speedups for FLO87 with different multigrid levels.

inner loops of the parallel threads already carry all the spatial locality in the program. The uniprocessor code speeds up slightly; the multiprocessor enjoys a 7.1 times speedup on 8 processors. Direct parallelization also speeds up the program on a multiprocessor, but to a lesser extent (2.8 times).

6.2.2 Performance of the Application

Our locality optimization speeds up the performance of the entire application by 14% on a uniprocessor. Because of Amdahl’s Law, the poor speedup of the EFLUX subroutine leads to a speedup of only 4.8 on 8 processors. If code duplication is applied, a speedup of 6.1 is obtained.

In a multigrid method, the mesh is coarsened by halving each of its dimensions at each level. With an input mesh size of $192 \times 32 \times 48$, the use of five multigrid levels means that the coarsest mesh has only $12 \times 2 \times 3$ cells. When distributed over 8 processors, the amount of computation associated with the coarsest mesh is small compared to the communication overhead involved. In general, as the multigrid level increases, the mesh gets coarser, the ratio of communication to computation increases, and a slower parallel speedup is expected. The slower speedup, however, may be offset by the fact a program with more multigrid levels may take less iterations to converge.

To experiment with the number of multigrid levels and to evaluate the scalability of the transformed code, we scaled up the mesh by 8 times to $384 \times 64 \times 96$ and varied the multigrid levels from 3 to 6. The performance was measured on the code where manual code duplication has been applied to the EFLUX routine. As shown in Figure 5, the performance results for different multigrid levels are almost identical up to 16 processors. When compared to the code produced by the native sequential F77 compiler, the achieved speedups on 32 processors vary from 25 to 22 as the multigrid levels are increased from 3 to 6.

7. RELATED WORK

A large number of data locality optimizations have been proposed in the past. Many of these algorithms use unimodular transformation and blocking, which are only applicable to perfectly nested loops. Unimodular transforms are used to create fully permutable loop nests that can be trivially blocked to increase locality. Some algorithms attempt to create the largest innermost tiles of loops that carry reuse, but can only be applied to individual perfect loop nests[23]. Others proposed to apply additional loop transformations, such as fusion, fission and code sinking, to create perfect loop nests from imperfect ones so that unimodular trans-

formations and blocking can be applied[6, 11, 24]. One of the most aggressive techniques of this class is proposed by Ahmed[1, 2]. The idea is to embed all the statements in a program into a high-dimensional iteration space, which is the Cartesian product of the iteration spaces of individual statements. In their algorithm, the multi-dimensional embedding functions are chosen with the goal of creating the largest nested layers of permutable loop nests.

Our affine partitioning algorithm can find the best affine partition that maximizes the largest outermost fully permutable loop nest. This work subsumes previous heuristic approaches as the best affine partition is equivalent to the best combination of unimodular transform, fusion, fission, reindexing, scaling and statement reordering. Our data locality approach, however, is not limited to just applying blocking to fully permutable loop nests. By generalizing the concept of blocking as thread interleaving, our algorithm can directly bring the benefits of blocking to arbitrarily nested loops. This generality is important because only one out of all the programs discussed in this paper can be transformed into one permutable loop nest.

Kodukula et al. proposed the idea of data shackling to improve data locality[13]. The idea is to divide the data arrays in a program into multiple blocks. The code is transformed to operate on a block of data at a time and thereby achieve data locality. Their algorithm does not specify how to divide the array into blocks nor how to schedule the computation if it is illegal to process the data a block at a time in one pass. In contrast, our algorithm automatically finds a legal desirable reordering of the operations to enhance locality.

Rosser and Pugh apply iteration space slicing techniques to optimize data locality[19]. The key parameters in their algorithm are the array to be “sliced” and the dimension along which slicing takes place. Their algorithm derives from the data dependences the set of operations that contributes to the calculation of all the elements belonging to a slice of the given array. By executing the slices one at a time, temporal locality is improved. As stated in their paper, it is critical to choose the right array and dimension to slice, but the methods for finding the right choices are beyond the scope of their work. In comparison, this paper has presented a fully automatic algorithm for this problem. Another difference is that their algorithm does not address blocking.

The problem of applying loop fusion to improve data locality has been shown to be NP-hard[12]. Most previous approaches fuse loops that read and write the same arrays heuristically to improve locality[12] and to create opportunities for array contraction[8, 14, 20]. As demonstrated by the examples in this paper, array contraction often requires global analysis and code restructuring, and cannot be performed effectively using pairwise loop fusion primitives. While our algorithm is not guaranteed to be optimal, its ability to optimize across all the arbitrarily nested loops globally makes it effective in enhancing locality and finding contractable arrays in practice.

8. CONCLUSION

This paper presents an integrated data locality algorithm that uses affine partitioning to support both array contraction and blocking. The technique is applicable to arbitrarily nested loops. The key idea is to find the best affine partition, which is equivalent to the best combination of unimodular transforms, fusion, fission, reindexing, scaling and statement

reordering, that separates the program into the largest number of independent threads. Computation within a thread exhibits a high degree of temporal locality. Executing the threads one at a time also minimizes the live ranges of the elements in work arrays, thus exposing opportunities for array contraction. The algorithm interleaves the threads to achieve a blocking effect for imperfect loop nests; this action in turn may require the work arrays be expanded by a small factor.

Because of the power and generality of the affine framework, our algorithm can find many more contractable arrays than previously possible. Many have attempted to extend blocking by converting programs into perfect loop nests so that unimodular transforms and blocking can be applied. We generalize the notion of blocking as an interleaving of threads so it is applicable to arbitrary loop nestings directly. Combining this general notion of blocking with affine partitioning, our algorithm is a powerful technique that can restructure a program with arbitrary sequences and nests of loops thoroughly to achieve good data locality for uniprocessors and multiprocessors.

Our experimental results suggest that our algorithm is effective. On a set of three kernels, the speedups achieved on a uniprocessor are 1.1, 1.5 and 3.1, whereas the speedups on an 8-processor machine are 6.3, 10.6, and 17.5, respectively. With the aid of a manual code duplication optimization, the large multigrid application achieves a speedup of 25 on 32 processors, if three multigrid levels are used.

In conclusion, affine partitioning is an effective program transformation framework. The same basic primitives for finding space and time partitions are useful for both parallelism and locality. In the past, compiler optimizations tend to use many different heuristics and their behaviors are hard to predict. Our compiler, on the other hand, can consistently improve the performance of programs whose array accesses are mostly affine expressions of loop indices and symbolic constants.

9. ACKNOWLEDGMENTS

This research is funded in part by the Department of Energy contract LLL-B341491. We would also like to thank our colleagues in the ASCI project, Juan Alonso, Antony Jameson, Massimiliano Fatica and William Reynolds, for their help with our experiments on the FLO87 application.

10. REFERENCES

- [1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, pages 141–152, May 2000.
- [2] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Proceedings of Supercomputing 2000*, November 2000.
- [3] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Technical report, Stanford University, 2000.
- [4] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4):345–420, December 1994.
- [5] D. Bailey and J. Barton. The NAS kernel benchmark program. Technical Report 86711, NASA, 1985.
- [6] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.
- [7] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.
- [8] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, pages 171–181, August 1992.
- [9] G. H. Golub and C. F. Van Loan. *Matrix Computations - 2nd Edition*. The John Hopkins University Press, Baltimore, Maryland, 1989.
- [10] A. Jameson. Solution of the Euler equations by a multigrid method. *Applied Mathematics and Computation*, 13:327–356, 1983.
- [11] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 323–334, July 1992.
- [12] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, August 1993.
- [13] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 346–357, June 1997.
- [14] E. C. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and array contraction in array languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 50–59, June 1998.
- [15] S.-W. Liao. *SUIF Explorer: an Interactive and Interprocedural Parallelizer*. PhD thesis, Stanford University, August 2000. Published as CSL-TR-00-807.
- [16] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greece, June 1999.
- [17] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.
- [18] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.
- [19] W. Pugh and E. Rosser. Iteration space slicing for locality. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1999.
- [20] V. Sarkar and G. R. Gao. Optimization of array accesses by collective loop transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 194–204, June 1991.
- [21] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, Chichester, 1986.
- [22] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, August 1992. Published as CSL-TR-92-538.
- [23] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.
- [24] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.