Linear and Extended Linear Transformations for Shared-Memory Multiprocessors

DATTATRAYA KULKARNI[1] AND MICHAEL STUMM[2]

[1]*Parallel Compiler Development, IBM Toronto Laboratory, Toronto, Canada M3C 1H7*
[2]*Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada M5S 3G4*
Email: dkulki@vnet.ibm.com

Advances in program transformation frameworks have significantly advanced compiler technology over the past few years. Program transformation frameworks provide mathematical abstractions of loop and data structures and formal methods for manipulating these structures. It is these frameworks that have allowed the development of algorithms capable of automatically tailoring an application for a target architecture. In this paper, we focus on the utility of these frameworks in improving the performance of mainly parallel applications on shared-memory multiprocessors. Data locality-oriented program optimizations are a key to good performance on shared-memory multiprocessors, since these optimizations can often improve performance by a factor of 10 or more. In this paper, we show the effectiveness of three key loop and data transformation frameworks in optimizing parallel programs on shared-memory multiprocessors. In particular, we describe our computation decomposition and alignment (CDA) framework, which can modify both the composition and the execution order of the re-composed iterations. We show how fine-grain transformations within the CDA framework enable new optimizations such as local optimizations that are otherwise achieved by global data transformations.

Received October, 1996, 1997; revised September, 1997

1. INTRODUCTION

Shared-memory multiprocessors are becoming increasingly popular and are now widely used in both universities and industry. While most of these systems, notably in industry, are still being used for multiprogramming sequential applications, their use for running parallel programs will increase in the future. Even for executing large-scale parallel applications, shared-memory multiprocessors are becoming the platform of choice over distributed memory message passing systems. The shared memory offers the promise of a simpler programming model, allowing for easier development and porting of parallel applications.

Intuitively, one would expect the availability of shared memory to make the task of parallelizing applications easier, allowing a compiler to focus on identifying and extracting parallelism from the target application rather than worrying about data distribution and placement or the generation of message passing code. In reality, however, it is crucial to focus on data management, at times even more than on extracting parallelism. While not focusing on data management results in correctly working programs when executed on a shared-memory machine, it can result in a significant loss of potential performance.

Three important optimizations aimed at managing data are: improving cache locality, reducing cache conflicts and improving memory locality. Over the years, many program transformation techniques that perform such optimizations have been proposed, and we describe several of them in this paper. Most of the transformations are capable of only modifying loops and arrays: loops, because they are regular, well-defined control structures that are straightforward to manipulate and because they constitute the core of most scientific computing, arrays, also because they have a regular structure.

Focusing on data management adds considerable complexity to the optimization of parallel applications, because the compiler must deal with parallelization and data management in an integrated way—changes to code or changes to data layout both affect the way data is accessed. Moreover, because the layout of the data is visible to all processors, it must be optimized in a global sense. In contrast, distributed memory compilers tend to address data management issues in a less integrated way, since the compiler first distributes and assigns data to the individual processors so as to minimize inter-processor communication. The program is later optimized for cache performance considering local data partitions on individual processors.

To optimize a program, it is often necessary to apply multiple different transformations to the program. This is where loop and data transformation frameworks play an important role—they provide a mathematical abstraction of program structures and formal methods to manipulate them. The same abstraction is used to represent and evaluate many different transformations and compound transformations. The mathematical foundation provides a basis for effectively:

1. representing a set of transformations in a concise and uniform way;
2. reasoning about and comparing transformations and their effects using formal methods;
3. automatically deriving transformations that achieve specific optimization objectives, and
4. applying transformations to the program code in an automated way.

Without a formal framework, reasoning about the effects of transformations tends to be *ad hoc*, which often leaves opportunities for performance improvements unexplored and makes the design of algorithms to automatically optimize applications difficult.

In this paper, we show the effectiveness of three loop and data transformation frameworks based on simple linear algebra. The *linear loop transformation* framework transforms loops to improve parallelism or data access locality, but can only transform perfectly nested loop structures. The *linear array transformation* framework transforms the target program by modifying the layout of the arrays with the objective of improving the data access locality. Finally, *computation decomposition and alignment* is a finer-granularity framework we have developed recently that extends the linear loop transformation framework, and is capable of transforming some non-perfectly nested loops. We illustrate the utility and effectiveness of these frameworks through transformations applied to many specific examples. Each of these examples, as well as several application kernels, were run on a 28-processor KSR-1 multiprocessor, and we include the performance numbers obtained from these experiments.

2. LINEAR LOOP TRANSFORMATION FRAMEWORK

The structure of a loop nest[3] defines the order in which data is accessed from within the loop nest. The access pattern it defines determines the caching behaviour of the loop nest, and in the case of larger systems, the number of remote references. The loop structure also defines the levels at which the loop nest can be parallelized and thus the granularity at which parallelism can be directly extracted. Parallelism in a loop nest is generally restricted by data dependences between statements in the loop, if the parallel execution of the loop nest is to produce the same result as the sequential execution [1]. The objective of a loop transformation is to rewrite the loop nest so that it produces the same result, but has improved parallelism or has an improved data access locality. Without a formal framework, the design of algorithms to automatically transform loops and code generation are generally difficult.

[3]When the context is clear, a loop nest is called a loop for simplicity.

The linear loop transformation framework, introduced in 1990 [2–7], was a major breakthrough that greatly simplified the task for the compiler, partly because it was a formal method based on linear algebra and partly because it provided a unified view of many of the previously proposed loop transformations. With this framework, it became possible to design algorithms that automatically search for transformations for given optimization objectives. However, linear loop transformations can only be applied to perfectly nested loop nests.

2.1. Overview of technique

Each iteration of a nested loop corresponds to a point in the iteration space for the loop, with the loop limits defining the bounds of the iteration space. For example, the loop called Loop 1 on the left-hand side of the top half of Figure 1 has two levels of nesting, so it corresponds to a two-dimensional iteration space as depicted on the right-hand side. Iteration (i, j) is represented as an integer point (i, j) in the iteration space. The iteration space is bounded by the inequalities $1 \le i \le n$ and $1 \le j \le n$, as defined by the loop limits. Precedence constraints between iterations are represented as relations between points in the iteration space [1, 2, 8–11].

A linear loop transformation, represented by a non-singular (i.e. invertible) integer matrix, maps each point in the original iteration space onto its own integer point in the new iteration space. The bounds of the new iteration space define the limits of the transformed loop, and relations between points in the new iteration space correspond to the new precedence constraints.

2.2. Applications of linear loop transformations

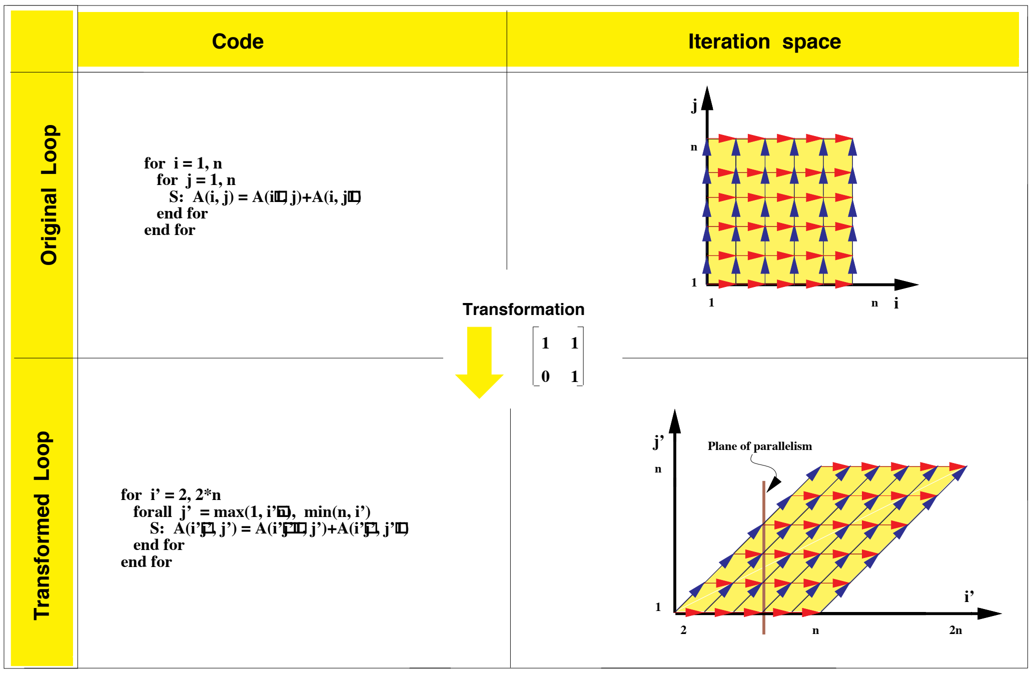
Linear loop transformations can be used to pursue a number of optimization objectives. Here we briefly describe three of them: one that improves parallelism, one that improves cache locality and one that minimizes the number of remote memory accesses.

It can be shown that it is always possible to linearly transform any perfectly nested loop into a new loop that has parallel executable inner iterations (i.e. where data dependences no longer exist between the inner iterations).[4] Figure 1 contains an example of just such a transformation. We have implemented this and all other examples of this paper on a 28-processor KSR-1. The performance improvements obtained are summarized in Table 1. In this case, Loop 1 achieved a speedup of somewhat over 8 with the given transformation. The speedup is due to the parallelism that was exposed by the transformation.

Linear loop transformations can also be used to improve memory access behaviour. For instance, a linear transformation applied to Loop 2[5] permutes the loop levels so that memory is accessed in storage order (which we assume throughout the paper to be row major order), thus

[4]Infinitely many transformations can achieve this.

[5]In the examples illustrating loop and data transformations, the original loop is shown on the left-hand side and the transformed loop is shown on the right-hand side.



```
for  i = 1, n
   for  j = 1, n
      S:  A(i, j) = A(i−1, j)+A(i, j−1)
   end for
end for
```

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

```
for  i' = 2, 2*n
   forall  j'  = max(1, i'−n),  min(n, i')
      S:  A(i'−j', j') = A(i'−j'−1, j')+A(i'−j', j'−1)
   end for
end for
```

FIGURE 1. Linear loop transformation that improves parallelization of Loop 1.

TABLE 1. Comparison of original and transformed example loops on a 28-processor KSR-1. The column of interest is the relative speedup, defined as the ratio of the parallel execution time of the original loop to the parallel execution time of the transformed loop. The original loops all executed in parallel except for Loops 1 and 7. The acronyms are: LLT, linear loop transformation; LAT, linear array transformation; CDA, computation decomposition and alignment; ELAT, extended linear array transformations.

Loop #	*n*	Relative speedup	Transformation	Optimization
1	1600	8.22	LLT	Inner loop parallelization
2	160	9.29	LLT	Spatial locality (cache line and page)
3	1600	16.08	LLT	Remote memory accesses
4	1600	4.71	LAT	Locality (cache and remote memory accesses)
5	1600	3.08	LAT	Remote memory accesses
6	256	3.77	ELAT	Cache conflicts
7	1600	15.43	CDA	Synchronization
8	1600	9.75	CDA	Locality (cache and remote memory accesses)
9	1600	1.45	CDA	Affinity between loops
10	256	3.39	CDA	Cache conflicts

improving the cache hit rate and possibly reducing paging activity.

The original loop has poor cache locality and can cause excessive paging because only one element of each cache line fetched is actually accessed and because each iteration of the innermost loop accesses a different set of pages. By interchanging the i and k loops, the transformed loop accesses elements of A in storage order.

In our final example, a linear loop transformation is used to reduce the number of remote memory accesses. This is still an important objective, even if the cost of accessing non-local data is significantly lower than on a distributed memory system. If we assume that matrix A is stored in row major order and distributed over multiple memory modules, then Loop 3 will cause many remote memory accesses, since each execution of the inner loop will access a different row of the matrix.

The transformed loop accesses the arrays contiguously by row, and row i of A and B can be mapped to memory close to the processor executing the ith iteration to make all array accesses in every iteration local. This type of transformation is referred to as access normalization [5], and achieves a speedup of over 16 on Loop 3 running on the KSR-1.

We have described three important and frequent uses of linear loop transformations: exposing available parallelism, improving cache and paging behaviour, and reducing the number of remote memory accesses. There are of course other applications. Improving processor load balance [3, 12] and enabling loop tiling [13, 14] are two examples. The latter is important, because loop tiling, although not a linear transformation in itself, is an effective technique that improves cache performance, even on uniprocessors. Often loops are not tileable as is, but linear loop transformations exist that can make them tileable.

```
forall $k = 1, n$                               forall $i = 1, n$
   forall $j = 1, n$                               forall $j = 1, n$
      for $i = 1, n$                                  for $k = 1, n$
         $A(i, j, k) = B(i, j, k) \ldots$                 $A(i, j, k) = B(i, j, k) \ldots$
      end for                          $\equiv$              end for
   end for                                         end for
end for                                         end for
```

Loop 2. Linear loop transformation that improves cache locality.

```
forall $i = 1, n$                                        forall $i = 0, n - 1$
   for $j = 1, i$                                           for $j = 1, n - i$
      $A(i - j, j) = A(i - j, j) + B(i - j, j)$                 $A(i, j) = A(i, j) + B(i, j)$
   end for                                      $\equiv$         end for
end for                                                  end for
```

Loop 3. Linear loop transformation that reduces the number of remote memory accesses.

2.3. Automatic derivation of linear loop transformations

The derivation of a linear loop transformation that is optimal for a given optimization objective is, unfortunately, hard in general. The problem is NP-complete for unrestricted loops and even affine loops with non-constant dependence distances [15]. Since the inception of the linear loop transformation framework, researchers have developed heuristic algorithms capable of automatically deriving transformations given specific optimization objectives, such as maximizing parallelism, maximizing cache locality, minimizing communication volume, balancing load etc. [3–5, 7, 12, 16].

The approximate solutions are typically derived by using the desired properties of the transformed loop, such as the structure of the dependence matrix or patterns of expressions in array references, to guide the search for a transformation. For example, an important optimization objective is to bring parallelism to the outermost loops, so that there are no dependences between the outer loop iterations, thus removing the need for synchronization. The dependence matrix can be used to guide the search and to derive a linear transformation to parallelize the outer loops of a perfectly nested affine loop in polynomial time [4, 17]. Such algorithms apply a sequence of matrix operations to transform the original dependence matrix into a new dependence matrix having the desired properties; in this case, the new dependence matrix will have the first row and possibly a few subsequent rows that contain only zero elements.

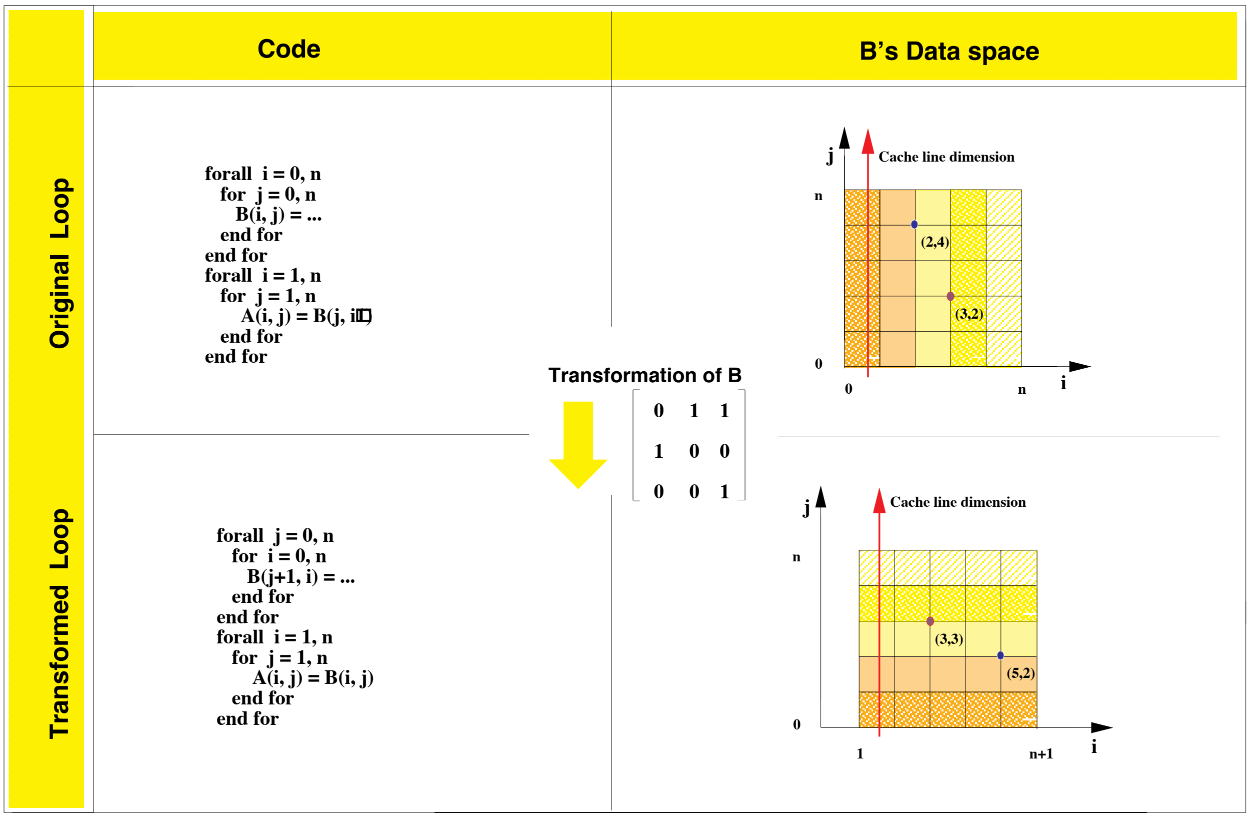
3. LINEAR ARRAY TRANSFORMATION FRAMEWORK

The previous section described how linear loop transformations can change the data access pattern to improve performance. An alternative strategy to pursue the same objective is to change the way data is stored in memory. In this section we describe three such techniques: linear array transformations [18, 19], array padding [20, 21] and data tiling. The linear array transformation is similar to the linear loop transformation in concept and also subsumes numerous data transformations. Array transformations are considered to be global, since they affect all references to the array in the entire program. This is in contrast to loop transformations that only affect individual loops. Because data transformations are global, it can be difficult to find a good one, since each transformation applied must be suitable for each loop that accesses the target array. On the other hand, data transformations do not change data dependences and are thus always legal to apply. Array transformations are used extensively on distributed memory machines to implement data alignments [18] that match references to arrays so that they can be distributed in a way that minimizes communications.

3.1. Linear array transformation

A target array can be regarded as an integer space with each data element of the array corresponding to an integer point. A linear transformation can then be applied to the integer space, effectively changing the storage location of the data elements. Such a transformation requires that each reference to the array be updated to reflect the new location of the target element.

A linear array transformation is illustrated in Figure 2, where the original program has two loops. The first initializes array B, while the second, named Loop 4, reads it and writes to array A. Loop 4 accesses array A in row major order, but accesses B in column major order. It also accesses B at an offset; that is, iteration (i, j) accesses $B(j, i - 1)$ instead of $B(j, i)$. The transformation in Figure 2 transforms array B in two ways. First, B is transposed so that it is accessed in storage (row major) order. The transposition requires a change in references to B from $B(j, i - 1)$ to $B(i - 1, j)$. If data is both stored and accessed in row major order, then the cache hit rate will be significantly higher than if the data is stored in row major order but accessed in column major order. On large systems, the transposition also reduces the number of remote accesses; some of the accesses in the original loop will invariably be to remote memory,



```
forall  i = 0, n
  for  j = 0, n
    B(i, j) = ...
  end for
end for
forall  i = 1, n
  for  j = 1, n
    A(i, j) = B(j, i□
  end for
end for
```

```
forall  j = 0, n
  for  i = 0, n
    B(j+1, i) = ...
  end for
end for
forall  i = 1, n
  for  j = 1, n
    A(i, j) = B(i, j)
  end for
end for
```

FIGURE 2. Illustration of a linear array transformation, improving locality (Loop 4).

whereas accesses in the transposed loop can be made local by mapping B by row to appropriate memory modules. The second effect of the transformation is to shift B right by one so that iteration (i, j) accesses $B(i, j)$ instead of $B(i-1, j)$ to match the access to $A(i, j)$.

The above transformation improved the performance of Loop 4 by a factor of 4.7 on our KSR-1 (Table 1). It should be noted that a linear loop transformation could not have achieved the same result—if the second loop is loop-transformed to modify its accesses to B, then its accesses to A would also change in a similar way, yet here we wish to change only the accesses to one of the arrays and not both. This example also illustrates one of the disadvantages of data transformation. While the array transformation improves the behaviour of the second loop, it makes it worse for the first, because the transformation on B is global. Luckily in this case, it was possible to interchange the loops of the first nest to retain row major order access.

The example transformation above illustrated two uses of linear array transformations: improving cache performance and reducing remote memory accesses. With a proper mapping of rows to memory modules close to the accessing processors, all B accesses can be made local by simplifying array index functions in Loop 5 using transformation similar in effect to access normalization loop transformations [5].

3.2. Other types of data transformations

Other data transformations exist that do not belong directly to the category of linear array transformations. For example, array padding increases the array size (along any dimension) and/or introduces a dummy array between two arrays for the purpose of reducing cache conflicts. The four elements in the original Loop 6, namely $A(i, j, k)$, $B(i-1, j, k)$, $B(i, j, k)$ and $B(i+1, j, k)$ are accessed in each iteration and conflict in the cache on a system with 256 Kb two-way set-associative caches (as is the case on the KSR-1), assuming $256 \times 256 \times 256$ arrays with row major ordering and 8 bytes per array element. Hence, each access will cause a cache miss. The conflicts can be eliminated by increasing the size of array B from $256 \times 256 \times 256$ to $256 \times 258 \times 258$.

One disadvantage of padding and some unimodular array transformations that increase the size of arrays is that they affect the relative storage positions of all arrays that follow in memory. In recent work, we have extended linear array transformations to include nonlinear modulo operators in the transformation. A unimodular array transformation in conjunction with modulo operations wraps the transformed data space around to allow the array transformation without the increase in size. For example, a unimodular array transformation that maps $A(i, j)$ to $A(i', j')$ could be modified to map $A(i, j)$ to $A(i'\%m, j'\%n)$ if the size of the target array is $m \times n$.

Array B of Loop 6 can be transformed so that $B(i, j, k)$ is mapped to $B(i, (j+i)\%n, k)$ to eliminate the cache conflicts described earlier. The transformed loop no longer has cache conflicts since no more than two array accesses map onto a cache set. The result is a speedup of 3.7 on the KSR-1.

Another nonlinear extension to the linear array transformation framework, we have developed, is called array tiling

```
forall  $i = 1, n$                              forall  $i = 1, n$
   for  $j = 1, n$                                 for  $j = 1, n$
      $A(i, j) = B(i + j, j) \ldots$                  $A(i, j) = B(i, j) \ldots$
   end for                            $\equiv$          end for
end for                                         end for
```

Loop 5. Linear array transformation to improve remote memory access pattern.

```
forall  $i = 1, n$                                   forall  $i = 1, n$
   for  $j = 1, n$                                      for  $j = 1, n$
      for  $k = 1, n$                                      for  $k = 1, n$
         $A(i, j, k) = \sum_{c=-1,0,1} B(i \pm c, j, k)$          $A(i, j, k) = \sum_{c=-1,0,1} B(i \pm c, (j + i \pm c)\%n, k)$
      endfor                          $\equiv$                endfor
   endfor                                               endfor
endfor                                               endfor
```

Loop 6. Extended linear array transformation to eliminate cache conflict misses.

and uses both division and modulo operators. Conceptually, the target $m \times n$ array is tiled into $a \times b$ subarrays or data tiles, and the array is then transformed so that all elements of any data tile fit in one cache line of size l. The restrictions on a and b are that l/a divides evenly and that $b = l/a$. In effect, this transformation implements rectangular cache lines, and thus can be viewed as the data equivalent of loop tiling. It can significantly increase the cache hit rate for loops that access elements of the array along both the i and j dimensions.

Both of the above nonlinear extensions introduce more complex and costly array index computations. But with the difference between processor and memory speeds increasing, it becomes increasingly beneficial to trade-off memory access overhead for index computation overhead.

3.3. Automatic derivation of array transformations

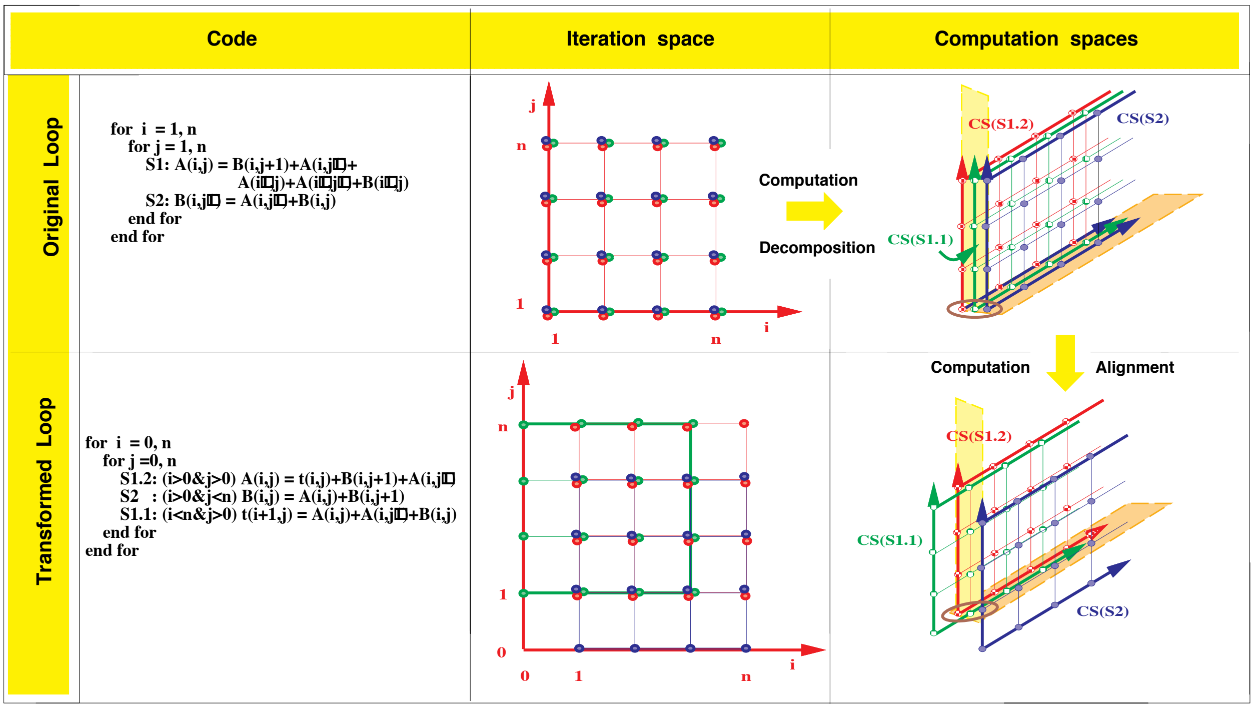
The general problem of deriving linear array transformations is to find the optimal shapes for the arrays considering all nested loops in the program. Finding the optimal shape for arrays is NP-hard [22]. Approximate techniques for automatically deriving data alignments [18, 19, 23], that transform one array onto another, can be used in this case to improve cache and memory locality. Recent work in dependence analysis and optimization techniques suggests that exact techniques may in fact have acceptable execution times in practice [10]. Therefore, we believe that exact techniques that exhaustively search for optimal linear array transformations may not be impractical. This is especially so, because the arrays accessed in programs tend to have only a small number of array dimensions and simple transformations such as transpose or permutation of array dimensions result in good solutions in practice as these transformations have low index computation and spatial overhead.

4. EXTENDED LINEAR LOOP TRANSFORMATION FRAMEWORK

The linear loop transformation framework is elegant and can be effective, but it does not always exploit all of the optimization opportunities that are available. Loops are transformed at the granularity of iterations, which only changes the order in which the iterations are executed. It does not transform the composition of the iterations themselves. Recently, several groups have been exploring extensions to the linear loop transformation framework to transform loops at statement [24, 25] and substatement granularity [26, 27]. Here we describe our transformation framework called computation decomposition and alignment (CDA). The CDA transformation framework unifies many existing transformations, including all linear loop transformations, and enables new optimizations which cannot be obtained by linear loop transformations alone. It is also capable of transforming some non-perfectly nested loops. However, heuristics are essential to efficiently derive CDA transformations as the space of candidate transformations tends to be large.

4.1. Basic idea of the CDA transformation technique

A CDA transformation consists of two stages. In the first stage, *computation decomposition* decomposes the loop body initially into its individual statements, and then optionally the individual statements into statements of finer granularity. A statement is decomposed by rewriting it as a sequence of smaller statements that produce the same result as the original statement. In so doing, it is necessary to introduce temporary variables to pass intermediate results between the new statements. For example, the statement $a = b + c + d + e$ can be partitioned into $t = d + e$ and $a = b + c + t$, where t is a temporary variable used to pass the result of the first statement to the second. A statement can be decomposed multiple times into possibly many statements.



```
for  i  = 1, n
   for j = 1, n
      S1: A(i,j) = B(i,j+1)+A(i,j□)+
                      A(i□,j)+A(i□,j□)+B(i□,j)
      S2: B(i,j□) = A(i,j□)+B(i,j)
   end for
end for
```

```
for  i  = 0, n
   for j =0, n
      S1.2: (i>0&j>0) A(i,j) = t(i,j)+B(i,j+1)+A(i,j□)
      S2   : (i>0&j<n) B(i,j) = A(i,j)+B(i,j+1)
      S1.1: (i<n&j>0) t(i+1,j) = A(i,j)+A(i,j□)+B(i,j)
   end for
end for
```

FIGURE 3. Illustration of an extended linear loop transformation.

The choice of which subexpressions to elevate to the status of statements is a key decision in CDA optimization and is determined largely by the specific optimization objective being pursued.

A sequence of decompositions produces a new loop body that can have more statements than the original, but the loop references and loop bounds remain unchanged. For each new statement S, there is a computation space, $CS(S)$, which is an integer space that represents all execution instances of statement S in the loop.

In the second stage of CDA, *computation alignment* applies a separate linear transformation to each of the computational spaces. The set of all transformed computation spaces together defines the new iteration space. Unlike the original iteration space, the new iteration space may be non-convex, so the corresponding new loop may have complex bounds.

4.2. Example transformation

Figure 3 illustrates the application of a simple CDA transformation. Computation decomposition first splits the loop body into two statements S_1 and S_2. Statement S_1 is further decomposed into two smaller statements $S_{1.1}$ and $S_{1.2}$, using a temporary array t to pass the result of $S_{1.1}$ to $S_{1.2}$. The result is a loop with three statements in the body:

$$S_{1.1}: t(i, j) = A(i-1, j) + A(i-1, j-1) + B(i-1, j)$$
$$S_{1.2}: A(i, j) = t(i, j) + B(i, j+1) + A(i, j-1)$$
$$S_2: B(i, j-1) = A(i, j-1) + B(i, j).$$

This computation decomposition effectively partitions the iteration space into three computation spaces, namely $CS(S_{1.1})$, $CS(S_{1.2})$ and $CS(S_2)$. The particular decomposition for S_1 was chosen so that it separates all $(i-1, *)$ references into a new statement, in this case $S_{1.1}$. This will allow a subsequent transformation to modify the $(i-1, *)$ references into $(i, *)$ references, without affecting the other references in S_1 that are now in $S_{1.2}$.

The three computation spaces are computationally aligned by applying transformations

$$T_{1.1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad T_{1.2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and

$$T_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

to $CS(S_{1.1})$, $CS(S_{1.2})$ and $CS(S_2)$ respectively. As a result, computation spaces $CS(S_{1.1})$ and $CS(S_2)$ move relative to $CS(S_{1.2})$, since $T_{1.2}$ is the identity matrix. $CS(S_{1.1})$ moves one stride in direction i so that the $(i-1, *)$ references in $S_{1.1}$ change to $(i, *)$ references. $CS(S_2)$ moves one stride in direction j so that the $B(i, j-1)$ reference changes to $B(i, j)$. The transformations thus align the computation spaces so that most references are aligned to $A(i, j)$. Figure 3 shows the transformed computation spaces and highlights three computations that are now executed in one iteration.[6]

[6]It was necessary to change the order of the statements in the loop so that $S_{1.1}$ is executed after $S_{1.2}$ and S_2 to maintain legality. Before the transformation, $S_{1.1}$ had a loop-carried flow dependence from both $S_{1.2}$ and S_2. These dependences become loop independent after the alignment, thereby necessitating the reordering.

```
for  $i = 1, n$                                  forall  $i = 0, n$
  for  $j = 1, n$                                  for  $j = 1, n$
    $A(i, j) = A(i, j-1)\ldots$                       $(i > 0)$  $A(i, j) = A(i, j-1)\ldots$
    $B(i-1, j) = A(i-1, j)\ldots$        $\equiv$          $(i < n)$  $B(i, j) = A(i, j)\ldots$
  end for                                        end for
end for                                        end for
```

Loop 7. CDA transformation to eliminate synchronizations.

The new iteration space is defined by the projection of the transformed computation spaces onto a plane. Iteration (i, j) in the new iteration space now has new, different instances of $S_{1.2}$, S_2 and $S_{1.1}$ computations, namely those that were originally in iterations (i, j), $(i, j+1)$ and $(i+1, j)$ respectively. The new iteration space is non-convex, and the limits of the new, transformed loop correspond to the convex hull of this new iteration space. It is no longer true that each iteration entails the execution of all three statements. The transformed loop requires guards that allow a statement to be executed only if appropriate. These guards can be removed in most cases to produce a more complex transformed code with imperfect nesting.

5. APPLICATIONS OF CDA TRANSFORMATIONS

In this section, we describe some of the optimization contexts, where CDA's ability to transform loops at statement and subexpression granularity is particularly useful.

5.1. Elimination of synchronizations

Loop 7 does not have any outer loop parallelism due to the $(1, 0)$ and $(0, 1)$ dependences; that is, iterations (i, j) must wait until iterations $(i-1, j)$ and $(i, j-1)$ have completed. Because the loop has only inner loop parallelism, it does not scale due to synchronization overheads. The $(1, 0)$ dependence can be eliminated by linearly transforming the computations associated with the second statement while leaving the first statement as is. In this particular example, we transform the second statement in order to change the $A(i-1, j)$ reference to $A(i, j)$, with the result of bringing parallelism to the outer loop. The result is a speedup of almost 15.5 on our 28-processor KSR-1 (Table 1). These types of CDA transformations are called loop alignments.

In general, dependences in a loop limit the availability of coarse-grained parallelism. When the rank of the dependence matrix is not less than the loop dimension, then the loop nest does not have any parallel outer loops (although all the inner loops can be made parallel) [4]. Unfortunately, the performance of parallel inner loops is not scalable, since the dependences in the outer loops manifest themselves as barrier synchronizations. CDA transformations can be used to modify the dependence matrix of some loops, so that the rank becomes less than the loop dimension; this effectively eliminates the need for synchronization and thus makes available coarse-grained parallelism.

Here, we show how to derive CDA transformations that eliminate synchronization by modifying loop-carried dependences between statements into loop-independent dependences. The technique is general in that it can be used to modify a loop-carried dependence $\mathtt{d}$ between two statements S_w and S_r into any desired dependence $\mathtt{d}'$, although in this case we would modify $\mathtt{d}$ to be $\mathtt{0}$. Let A_w and A_r be the reference matrices of the write reference w to an array in statement S_w and a read reference r to the array in statement S_r respectively.[7] We can derive a matrix f, which transforms the computations of S_r so that the dependence between w and the modified reference r' becomes $\mathtt{d}'$, as follows. Let A_w, A_r and f^{-1} be

$$A_w = \begin{bmatrix} U_w & 0 \\ 0 & 1 \end{bmatrix} \quad A_r = \begin{bmatrix} U_r & \mathtt{d_r} \\ 0 & 1 \end{bmatrix} \quad f^{-1} = \begin{bmatrix} T & \mathtt{t} \\ 0 & 1 \end{bmatrix}.$$

Transforming the computations of S_r by matrix f modifies the reference matrix A_r to become $A_r f^{-1}$. If the dependence between statements S_w and S_r in the original loop was $\mathtt{d}$, then reference matrices $A_w\mathtt{I}$ and $A_r\mathtt{I} + A_r\mathtt{d}$ both access the same array element in the original loop. We would like $A_w\mathtt{I}$ and $A_r\mathtt{I} + A_r\mathtt{d}'$ to access the same array element after the transformation; i.e.

$$A_w\mathtt{I} = A_r f^{-1}(\mathtt{I} + \mathtt{d}').$$

By substituting for A_w, A_r and f^{-1} in this equation, we obtain

$$\begin{bmatrix} U_w & 0 \\ 0 & 1 \end{bmatrix}\mathtt{I} = \begin{bmatrix} U_r T & U_r\mathtt{t} + \mathtt{d_r} \\ 0 & 1 \end{bmatrix}(\mathtt{I} + \mathtt{d}').$$

After expanding the matrix-vector multiplication on the right-hand side we have

$$U_w\mathtt{I} = U_r T(\mathtt{I} + \mathtt{d}') + U_r\mathtt{t} + \mathtt{d_r}$$

which can be expanded into

$$U_w\mathtt{I} = U_r T\mathtt{I} + U_r T\mathtt{d}' + U_r\mathtt{t} + \mathtt{d_r}.$$

For this equation to be true for all iterations $\mathtt{I}$, it is necessary for T to be equal to $U_r^{-1}U_w$, and for $U_r\mathtt{t}$ to be equal to $-U_r T\mathtt{d}' - \mathtt{d_r}$. By substituting $T = U_r^{-1}U_w$ in this latter equation, we obtain

$$U_r\mathtt{t} = -U_w\mathtt{d}' - \mathtt{d_r}$$

which we can solve for $\mathtt{t}$ by pre-multiplying both sides of the equation by U_r^{-1}:

$$\mathtt{t} = -U_r^{-1}U_w\mathtt{d}' - U_r^{-1}\mathtt{d_r}.$$

[7]The matrices are suitably padded when the array and loop dimensions are not the same.

```
forall  $i = 0, n/2$
   for  $j = 0, n/2$
      $A(2*i-1, 2*j) = A(2*i, 2*j) + A(2*i-2, 2*j)$
      $A(2*j, 2*i-1) = A(2*j, 2*i) + A(2*j, 2*i-2)$
   end for
end for
```

$\equiv$

```
forall  $i = 0, n/2$
   for  $j = 0, n/2$
      $A(2*i-1, 2*j) = A(2*i, 2*j) + A(2*i-2, 2*j)$
      $A(2*i, 2*j-1) = A(2*i, 2*j) + A(2*I, 2*J-2)$
   end for
end for
```

Loop 8. CDA transformation to improve cache locality.

Therefore, matrix f is defined such that

$$f^{-1} = \begin{bmatrix} U_r^{-1}U_w & -U_r^{-1}U_w\mathtt{d}' - U_r^{-1}\mathtt{d}_\mathtt{r} \\ 0 & 1 \end{bmatrix}$$

which modifies the dependence $\mathtt{d}$ between statements S_w and S_r to be $\mathtt{d}'$. In particular, when the desired dependence $\mathtt{d}'$ is $\mathtt{0}$, then the transformation f is defined such that

$$f^{-1} = \begin{bmatrix} U_r^{-1}U_w & -U_r^{-1}\mathtt{d}_\mathtt{r} \\ 0 & 1 \end{bmatrix}.$$

The technique we have just described is a generalization of loop alignment. While loop alignment considers only offset alignments between statements, CDA transformations can be any non-singular integer matrices. Moreover, CDA can align subexpressions, statements or subnests in the loop body. Thus, CDA can be viewed as unifying loop alignment and its generalizations into a linear algebraic framework.

5.2. Improving locality and parallelism

CDA can achieve improvement in locality and parallelism that is traditionally achieved with loop distribution and fusion. Consider the improvement of locality in Loop 8. Loop levels i and j need to be interchanged for the second statement, but not the first, in order to ensure row major order access for array A. Traditionally, this is done by distributing the statements in the loop into two separate loop nests, interchanging the i and j loops in the second nest and then fusing the two loop nests back together again. CDA can directly transpose the computation space of the second statement alone, improving the execution time by a factor of 9.7.

In particular, the CDA framework generalizes loop distribution in three ways. First, any loop distribution can be represented by a CDA transformation that decomposes the loop body and applies an appropriate offset alignment along the outermost loop dimension to each group of statements (to be distributed) so that the computation space for each group does not overlap with the computation space for any other group. The first group is aligned by an offset of 0, whereas the ith group is aligned by an offset of $ni - n$, where n is the size of the outermost loop. Note that the loops resulting from loop distribution and the loops resulting from the CDA transformation differ in subscript functions. However, most compilers can normalize loop bounds to simplify the subscript functions.

Second, with CDA, loop distribution can be performed at the granularity of subexpressions, and not just entire statements. Computation decomposition can store the results of subexpressions in temporaries to isolate dependence cycles. Thus, appropriate computation decomposition can be considered as a form of node splitting, which introduces temporaries to break dependence cycles. In contrast to node splitting, CDA can also move computations of the portions of the split statement apart from each other in the iteration space.

Third, CDA makes partial loop distributions possible. A loop distribution separates all instances of a statement from the instances of another statement in the loop body. A partial loop distribution separates only some instances of a statement or subexpression from the instances of other statements or subexpressions in the loop body. A partial loop distribution would be beneficial in a situation where a dependence cycle prevents loop distribution (and where node splitting does not help break the dependence cycle).

As an example, consider the loop on the left-hand side of Figure 4, which has a dependence cycle between statements S_1 and S_2. The dependence cycle cannot be isolated with computation decomposition, since the dependences in the cycle are flow dependences.[8] However, S_2 can be aligned by an offset of $-k$ to obtain the transformed loop shown in the centre of the figure. The CDA transformed loop after eliminating the guards is shown on the right-hand side of Figure 4. The transformation effected a partial loop distribution where only k computations (out of n) of the statements S_1 and S_2 are separated from each other.[9] When the dependent iterations are far apart (i.e. k is large or k is an offset in the outermost loop index), then partial loop distribution can separate a substantial number of statement instances.

As another example, consider the loop on the left-hand side of Figure 5 which cannot be distributed due to a dependence cycle that cannot be broken. CDA can align S_2 by an offset along the j dimension in this case so that the statements are distributed only with respect to the j and k

[8]That is, all statements of the decomposed loop will be in the dependence cycle.

[9]The dependence cycle prevents the distributions of the computations of the statements in the remaining iterations.

```
L: for  $i = k + 1, n$
        for  $j = 1, n$
            $S_1: B(i, j) = A(i - 1, j)$
            $S_2: A(i, j) = B(i - k, j)$                    $\Longrightarrow$
        end for
    end for

                    L: for  $i = 1, n$
                            for  $j = 1, 2 * n$
                                $S_1: (k + 1 \le i \le n)\ B(i, j) = A(i - 1, j)$
                                $S_2: (1 \le j \le n - k)\ A(i + k, j) = B(i, j)$          $\Longrightarrow$
                            end for
                        end for

                                        $L_1$: for  $i = 1, k$
                                                for  $j = 1, n$
                                                    $S_1: A(i + k, j) = B(i, j)$
                                                end for
                                            end for
                                        $L_2$: for  $i = k + 1, n - k$
                                                for  $j = 1, n$
                                                    $S_2: B(i, j) = A(i - 1, j)$
                                                    $S_1: A(i + k, j) = B(i, j)$
                                                end for
                                            end for
                                        $L_3$: for  $i = n - k + 1, n$
                                                for  $j = 1, n$
                                                    $S_2: B(i, j) = A(i - 1, j)$
                                                end for
                                            end for
```

FIGURE 4. CDA transformation for partial loop distribution.

dimensions. The CDA transformed loops with and without guards are shown at the centre and right of Figure 5.

In general, any sequence of loop distribution, linear transformation of the new loop nests and fusion of the transformed loop nests can be represented by a single CDA transformation. A linear transformation applied to a loop nest after loop distribution is essentially an alignment transformation applied to the computation space of the corresponding statement(s). The loop fusion is effected while generating code for the CDA transformed loop, where the computation spaces are coalesced by projecting them onto a grid and finding their union.

5.3. Inter-loop computation alignments

CDA transformations can be used to improve memory locality across loop nests by tailoring the loop to the way iterations are assigned to parallel threads. Consider a situation where threads are assigned individual iterations for execution. For instance, in the original loop, Loop 9, assume that the first iteration of the first loop is assigned to the same thread as the first iteration of the second loop, and that the following iterations are assigned to subsequent threads in a similar way. With this assignment, the thread executing iteration i of both loops will read and write $A(i, *)$ and $A(i - 1, *)$. Because this data set overlaps with the data set being accessed by the thread executing iteration $(i - 1)$, there will be an excessive amount of consistency traffic. This situation can be alleviated by CDA transforming the second statement of the second loop so as to align the accesses to A in the second statement to the corresponding accesses in the first loop. The first statement of the second loop is kept unchanged to keep the accesses to B local. This results in a speedup of 1.45 over the original Loop 9.

5.4. CDA as a control dual to array padding

Many CDA transformations behave as duals to data transformations. This duality allows for additional opportunities for optimization, because CDA is an optimization local to the loop, while data transformations are global. For instance, a CDA transformation can eliminate cache conflict misses in Loop 6 in place of array padding or an extended linear array transformation. Loop 6, which is reproduced as Loop 10, can be CDA transformed by decomposing the statement into two statements and computationally aligning one with respect to the other along the j dimension. As a result, at most two of the references map to a cache set so that the CDA transformed loop behaves as though the arrays had been suitably padded without any changes to data layout. The speedup achieved on this example is 3.4 (Table 1). The CDA transformed loops in general may run somewhat slower than array-padded versions because of the computational and spatial overheads in the CDA transformed loops.

```
L: for  $i = 1, n$
      for  $j = 1, n$
        for  $k = 1, n$
          $S_1$ : $B(i, j, k) = A(i-1, j, k)$
          $S_2$ : $A(i, j, k) = B(i-1, j, k)$              $\Longrightarrow$
        end for
      end for
    end for

    L: for  $i = 1, n$
          for  $j = 1, n$
            for  $k = 1, n$
              $S_1$ : $(1 \le j \le n)$ $B(i, j, k) = A(i-1, j, k)$
              $S_2$ : $(n+1 \le i \le 2*n)$ $A(i, j-n, k) = B(i-1, j-n, k)$        $\Longrightarrow$
            end for
          end for
        end for

          L: for  $i = 1, n$
                $L_1$: for  $j = 1, n$
                      for  $k = 1, n$
                        $S_1$ : $B(i, j, k) = A(i-1, j, k)$
                      end for
                end for
                $L_2$ : for  $j = n+1, 2*n$
                      for  $k = 1, n$
                        $S_2$ : $A(i, j-n, k) = B(i-1, j-n, k)$
                      end for
                end for
              end for
```

FIGURE 5. CDA transformation for partial loop distribution.

```
forall  $i = 1, n$                               forall  $i = 1, n$
  for  $j = 1, n$                                  for  $j = 1, n$
    $A(i, j) = A(i, j-1) + B(i, j) \dots$              $A(i, j) = A(i, j-1) + B(i, j) \dots$
  end for                                        end for
end for                                        end for
forall  $i = 1, n$                               forall  $i = 0, n$
  for  $j = 1, n$                      $\equiv$           for  $j = 1, n$
    $B(i, j) = \dots$                                 $(i > 0)$  $B(i, j) = \dots$
    $A(i-1, j) = A(i-1, j-2) \dots$                  $(i < n)$  $A(i, j) = A(i, j-2) \dots$
  end for                                        end for
end for                                        end for
```

Loop 9. CDA transformation to improve access affinity across loops.

However, the advantage of CDA is that it does not change the layout of the data globally and only affects this one loop; i.e. it is a local optimization. Note that array padding is often avoided by robust compilers for the sake of program correctness, especially when the program accesses the same array in different shapes. In such situations, CDA becomes an effective alternative transformation.

In order to show the duality between CDA and array padding transformations with respect to improving cache efficiency, consider conflicts in a direct-mapped cache as depicted in Figure 6a.

We represent the number of elements in an array between the first element and a chosen element of the array using an integer vector, which we call a mapping vector. A mapping vector, $\mathtt{V} = (v_1, \dots, v_m)^{\mathrm{T}}$, is such that v_i is the size of the m-dimensional array along array dimension $i+1$, and v_m is 1. Then, the array element accessed using reference matrix r in iteration $\mathtt{I}$ is $r\mathtt{I}\cdot\mathtt{V}$ array elements away from the first element of the array. As an instance, the mapping vector for a two-dimensional $n \times n$ array is $V = (n, 1)^{\mathrm{T}}$ and reference $A(i, j)$ in iteration $(10, 5)$ accesses an element which is $10n + 5$ elements away from element $A(1, 1)$.

We can now use the mapping vector to represent the mapping of array accesses onto cache and memory. An array access in iteration $\mathtt{I}$ with reference matrix R_1 maps to memory location $M_1 = C_1 + R_1\mathtt{I} \cdot \mathtt{V}$, where C_1 is a

```
forall $i = 1, n$
    for $j = 1, n$
        for $k = 1, n$
            $A(i, j, k) = \sum_{c=-1,0,1} B(i \pm c, j, k)$
        end for
    end for
end for
```

$\equiv$

```
forall $i = 1, n$
    for $j = 0, n$
        for $k = 1, n$
            $(i < n)\quad T(i, j + 3, k) = \sum_{c=-1,0} B(i \pm c, j + 1, k)$
            $(i > 0)\quad A(i, j, k) = T(i, j + 2, k) + B(i + 1, j, k)$
        end for
    end for
end for
```

Loop 10. CDA transformation to eliminate cache conflict misses.

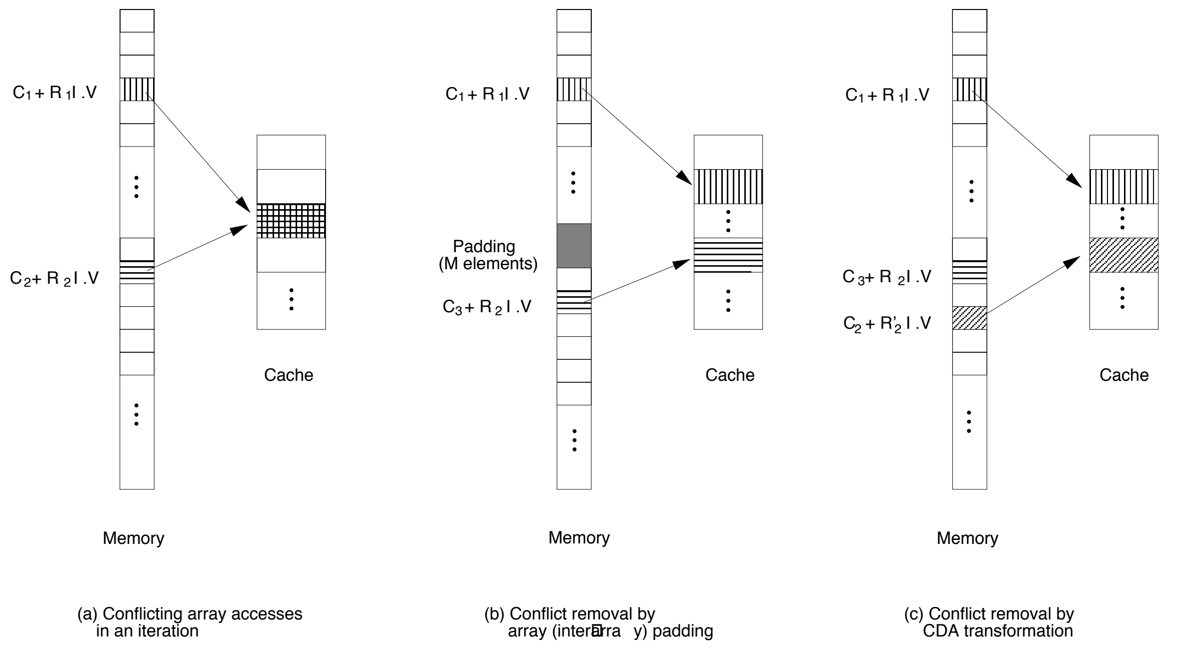

C1+ R 1I .V
C2+ R 2I .V
Cache
Memory
(a) Conflicting array accesses in an iteration
C1+ R 1I .V
Padding (M elements)
C3+ R 2 I .V
Cache
Memory
(b) Conflict removal by array (interarray) padding
C1+ R 1I .V
C3+ R 2I .V
C2 + R'2 I .V
Cache
Memory
(c) Conflict removal by CDA transformation

FIGURE 6. Reducing cache conflicts with modification to array layout and CDA.

constant and V is the mapping vector. A second array access in iteration I with reference matrix R_2 maps to memory location $M_2 = C_2 + R_2\mathtt{I} \cdot \mathtt{V}$.[10] A cache conflict occurs for these two accesses if the cache geometry is such that both M_1 and M_2 map to the same cache line.

The cache conflict shown can be eliminated by suitably modifying the number of elements between M_1 and M_2, so that the accessed data elements map to different cache lines (Figure 6b). Modifying V changes the array sizes, and is referred to as intra-array padding; this is achieved by changing the declaration of the arrays to become larger along one or more dimensions. Modifying the C_i changes the placement of the arrays in memory and is referred to as inter-array padding; this is achieved by inserting dummy variables between the array declarations.

The main idea behind using CDA to reduce the number of cache conflicts is to spread the conflicting accesses of an iteration into different iterations. While modification to array layout moves conflicting array accesses apart in space, CDA moves conflicting array accesses apart in time.

[10]Without loss of generalization we can assume here that arrays have the same size.

In the example we are considering, the time (in number of iterations) between the access to M_1 and access to M_2 can be changed by aligning the statement containing R_2 relative to the statement containing R_1. In other words, the statements containing R_1 and R_2 can be aligned so that R_1 and the aligned R_2, R_2', do not access M_1 and M_2 in the same iteration; in iteration I, R_1 would continue to access M_1, whereas R_2' would access a new location M_2'. In Figure 6c, R_2 is changed to R_2' so that the new memory location $M_2' = C_2 + R_2'\mathtt{I} \cdot \mathtt{V}$ and M_1 map to different cache lines.

5.5. Transforming imperfect loop nests

Although transformation of unconstrained imperfectly nested loops is still an open issue, some imperfectly nested loops can be successfully transformed using linear loop and CDA transformations.

Simple cases of imperfect nests are created, for example, in order to perform boundary computations or initializations. The cause of imperfectness is often a single assignment statement interspersed between the loop statements of an otherwise perfectly nested loop. In this case, the imperfect nest can be transformed into a perfect nest by moving the single assignment statement S_1 into the loop and using guards to prevent the execution of S_1 in some of the iterations [28, 29]. In the transformed loop, S_1 is executed only when $j = 1$ because of the guard. This new loop can now be applied to a linear loop transformation. An example transformation of this type is shown as Loop 11.

Another imperfectly nested loop structure that occurs frequently is illustrated by Loop 12, where the loop body of a perfectly nested loop contains a sequence of perfect subnests. Nested loops similar to Loop 12 can be linearly transformed only in a hierarchical fashion—each of the perfect subnests can be linearly transformed, and the loop nest containing the sequence of subnests can be linearly transformed. However, the entire loop nest cannot be transformed by a linear transformation.

With CDA, it is possible to transform this type of imperfect loop nest. For an example, the imperfect nest of Loop 12 can be CDA transformed to the loop nest on the right. CDA treats all computations of S_1 (in the i and j loops) as one computation space, while all computations of S_2 in i and j loops are treated as the second computation space. The transformation first aligns the computation space of S_2 by an offset of -1 along the j dimension and then skews both computation spaces so that the dependences are internalized to the inner loop. In the transformed loop, the flow dependence from S_1 to S_2 is loop independent, and the flow dependence from S_2 to S_1 is carried only by the j iterations. Therefore, the iterations of the outer loop are independent, so that the outer loop can be executed in parallel. Transformations of this type can expose additional optimization opportunities that linear loop transformations alone cannot.[11]

5.6. Automatic derivation of CDA transformations

The search space for legal CDA transformations of a nested loop is considerably larger than that for legal linear loop transformations. The increased difficulty in deriving CDA transformations is because one has to derive both a decomposition of the iteration space into computation spaces and a separate linear transformation for each of the computation spaces. Therefore, good heuristics are the key to efficient derivation of CDA transformations. Fortunately, we find that it is possible to design efficient algorithms for deriving CDA transformations in many optimization contexts using the knowledge of the optimization context in hand. In our recent work, we have designed two algorithms: one for automatically deriving CDA transformations to reduce the number of cache conflicts, and another to reduce the number of remote memory accesses and ownership tests [26]. The algorithms use the duality between CDA transformations and array padding, and CDA transformations and data alignment respectively, to derive the transformations automatically. The derivation of CDA transformations for other optimization contexts is an open problem.

An undesirable effect of applying CDA transformations is that the transformed loops may have a computational and spatial overhead, which a simple linearly transformed loop may not have. In our recent work we have designed techniques to reduce these overheads significantly and improve the efficiency of the CDA transformed loops [26]. These techniques are especially effective for CDA transformations, where a significant number of the transformed iterations execute all statements in the loop body.

6. EFFECTIVENESS OF THE TRANSFORMATIONS

Transformations for improving data access behaviour and parallelism of applications are obviously important for achieving good performance, and the frameworks we have presented play a crucial role in automating the search for and the application of suitable transformations. The examples we presented demonstrate the power of the transformations, achieving substantial improvements on a KSR-1 with 28 processors. While these examples may seem somewhat artificial—the codes were chosen specifically to isolate a particular feature of a transformation—comparable performance improvements can be obtained, in practice, when applying these same transformations on real application code. This is shown in Table 2 which lists the speedup obtained on a number of well-known application kernels. Here again, the speedup is measured as the ratio of the transformed code to the original code, both running fully parallel (except for `Mva`).

The loop and data transformation frameworks and their recent extensions presented here are effective on their own, but they are still separate frameworks. We believe they would be even more effective if they could be unified into a single meta-framework. Unifying them would offer additional opportunities for automatic program optimization, because of the duality that exists among the frameworks. In many cases, an objective can be achieved through either a data transformation or a loop transformation. For example, cache performance can be improved or the number of remote memory accesses can be reduced with both loop and data transformations.[12] Similarly, either array padding, extended linear array transformation or CDA transformations can be used to eliminate cache conflicts. This duality is exhibited in Table 2 where four of the five kernels are optimized within two frameworks. In a unified meta-framework, duality could be exploited to resolve conflicting optimization objectives, and to compensate for the limitations of constituent frameworks.

[11]A linear loop transformation cannot be applied to internalize the dependence and expose parallelism in the outer loop, because the loop is imperfectly nested. Linearly transforming the i loop or the j loops alone cannot expose parallelism either, since the dependences are across j subnests.

[12]Cierniak and Li used this property for an improved locality model of loops [30].

```
for  $i = 1, n$                                  for  $i = 1, n$
   $S_1$ :  $A(i, 0) = C$                           for  $j = 1, n$
   for  $j = 1, n$                                      $(j = 1)$ $S_1$ :  $A(i, 0) = C$
      $S_2$ :  $A(i, j) = A(i, j - 1) + D$    $\equiv$          $S_2$ :  $A(i, j) = A(i, j - 1) + D$
   end for                                         end for
end for                                         end for
```

Loop 11. Converting a simple imperfectly nested loop into a perfectly nested loop.

```
for  $i = 0, n$                                   for  $i = 0, 2n + 1$
   for  $j = 0, n$                                   for  $j = max(0, i - n - 1), min(n, i)$
      $S_1$ :  $A(i, j) = A(i - 1, j) + T_1$             $S_1$ :  $(i \le 2n, max(0, i - n) \le j \le min(i, n))$
   end for                                                    $A(j, i - j) = A(j - 1, i - j) + T_1$
   for  $j = 0, n$                        $\equiv$          $S_2$: $(i \ge 1, max(0, i - n - 1) \le j \le min(i - 1, n))$
      $S_2$ :  $A(i, j) = A(i, j + 1) + T_2$                       $A(j, i - j - 1) = A(j, i - j) + T_2$
   end for                                         end for
end for                                         end for
```

Loop 12. CDA transformation of an imperfectly nested loop.

TABLE 2. Speedup due to transformations on application kernels running on a 28-processor KSR-1. The speedup is measured as the ratio of the execution time of the original loop to that of the transformed loop. Except for `Mva`, all original loops are fully parallelized. The optimization performed is similar to that applied to the loop listed in the last column. `Wanal` in the last row was data tiled. The kernels are: `Mva`, mean value analysis; `Mm`, matrix multiplication; `Mg`, the main loop in a multigrid solver; `Wanal`, a perfect club benchmark loop and `Syr2k`, a BLAS3 loop that manipulates banded matrices (of width 160 here).

Kernel	n	Relative speedup	Transformation	Optn. as in
`Mva`	1600	20.25	LLT	Loop 1
`Mm`	320	4.62	LLT	Loop 2
`Mm`	320	5.47	LAT	Loop 4
`Syr2k`	1600	3.65	LLT	Loop 3
`Syr2k`	1600	10.74	LAT + LLT	Loop 3 & 5
`Mg`	256	3.15	ELAT	Loop 6
`Mg`	256	2.90	CDA	Loop 10
`Wanal`	1600	7.22	CDA	Loop 8
`Wanal`	1600	4.84	ELAT	

7. CONCLUDING REMARKS

The transformation frameworks are powerful because they are based on a mathematical foundation. This foundation provides a basis for effectively representing sets of transformations in a concise and uniform way, reasoning about and comparing transformations and their effects using formal methods, automatically deriving transformations that achieve specific optimization objectives and applying transformations to the program code in an automated way. Without formal frameworks, reasoning about the effects of transformations is *ad hoc*, which not only may leave opportunities for performance improvements unexplored, but also makes the design of algorithms to automatically derive transformations that optimize applications difficult.

The frameworks presented in this paper do, however, have limitations. Although algorithms exist capable of deriving optimal transformations given a single optimization objective,[13] it is not yet known how to find the optimal transformation given a set of multiple optimization objectives. In general, the search space of possible transformations is very large, and the techniques will have to rely on heuristics to find good, near-optimal approximations within a reasonable time [3–5, 7]. We believe that algorithms which derive optimal transformations for multiple objectives will ultimately require the use of an abstract model of the target hardware to realistically evaluate the performance tradeoffs between candidate transformations. This remains a challenging goal.

Furthermore, the transformation frameworks assume extremely regular loop and data structures. In fact, it is the assumption of regular structure that has allowed a relatively simple mathematical formulation of the problem and indirectly the development of algorithms for automatically deriving transformations. The frameworks are thus suitable for optimizing scientific numerical applications with regular loop and array structures, but it is unreasonable to expect these frameworks to carry over to the optimization of non-numeric applications. Because non-numeric applications tend to use dynamic data structures and have irregular loop structures (or other dynamic control flows), they are less amenable to mathematical formalization at compile time and are much more challenging to optimize.

While there have been some advances in deriving dependence information on pointer-based dynamic structures (i.e. memory disambiguation), it is still not clear how to use this information to transform irregular codes. Run-

[13]In some limited cases, optimal transformations can be derived that optimize a pair of objectives such as outer loop parallelism and cache locality.

time optimization techniques may be more promising. For example, speculative dependence analysis and code parallelization, where either the sequential or a parallel version is conditionally chosen at run time, is a possibility. A related research direction is designing run-time strategies that are directly embedded in the program, so that the dependence information could be collected during execution. Non-numerical applications also have optimization opportunities from techniques orthogonal to frameworks discussed here. For instance, prefetching techniques can sometimes hide the remote memory access latencies [31].

Although we have listed a number of limitations and much work is clearly still needed, the transformation frameworks we presented here have enormously advanced compiler technology over the past few years. These advances have brought us close to the point where parallelizing compilers can automatically generate efficient parallel code, at least within the domain of numerical applications. The technology described here is starting to appear in commercial compiler implementations by companies such as IBM and SGI.

REFERENCES

[1] Banerjee, U. (1988) *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Dordrecht.

[2] Banerjee, U. (1990) Unimodular transformations of double loops. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August.

[3] Kulkarni, D., Kumar, K. G., Basu, A. and Paulraj, A. (1991) Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proc. 1991 ACM Int. Conf. on Supercomputing*, Cologne, June.

[4] Kumar, K. G., Kulkarni, D. and Basu, A. (1992) Deriving good transformations for mapping nested loops on hierarchical parallel machines in polynomial time. In *Proc. 1992 ACM Int. Conf. on Supercomputing*, Washington, DC, July.

[5] Li, W. and Pingali, K. (1994) A singular loop transformation framework based on non-singular matrices. *Int. J. Parallel Program.*, **22**.

[6] O'Boyle, M. and Hedayat, G. (1992) A transformational approach to compiling SISAL for distributed memory architectures. In *Proc. 1992 ACM Int. Conf. on Supercomputing*, Washington, DC.

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

[8] Feautrier, P. (1991) Dataflow analysis of array and scalar references. *Int. J. Parallel Program.*, **20**, 23–53.

[9] Maydan, D., Hennessy, J. and Lam, M. (1991) Efficient and exact data dependence analysis. *SIGPLAN Notices*, **26**, 1–14.

[10] Pugh, W. (1992) A practical algorithm for exact array dependence analysis. *Commun. ACM*, **35**, 102–114.

[11] Pugh, W. and Wonnacott, D. (1993) *An Exact Method for Analysis of Value-based Array Data Dependences*. Technical Report CS-TR-3196, University of Maryland, MD.

[12] O'Boyle, M. and Hedayat, G. (1992) Load balancing of parallel affine loops by unimodular transformations. In *Proc. Eur. Workshop on Parallel Computing*, Barcelona.

[13] Lam, M., Rothberg, E. and Wolf, M. (1991) The cache performance and optimizations of block algorithms. In *4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April, pp. 63–74.

[14] Ramanujam, J. and Sadayappan, P. (1990) Tiling of iteration spaces for multicomputers. In *Proc. 1990 Int. Conf. on Parallel Processing*, pp. 179–186.

[15] Dowling, M. (1990) Optimum code parallelization using unimodular transformations. *Parallel Comput.*, **16**, 155–171.

[16] O'Boyle, M. and Hedayat, G. (1992) A new program transformation to minimise communication on distributed memory architectures. In *Proc. PARLE'92 Parallel Architectures and Languages Europe*.

[17] Kumar, K. G., Kulkarni, D. and Basu, A. (1991) Generalized unimodular loop transformations for distributed memory multiprocessors. In *Proc. Int. Conf. on Parallel Processing*, Chicago, MI, July.

[18] Li, J. and Chen, M. (1991) The data alignment phase in compiling programs for distributed memory machines. *J. Parallel Distrib. Comput.*, **13**, 213–221.

[19] O'Boyle, M. and Hedayat, G. (1992) Data alignment: transformations to reduce communication on distributed memory architectures. In *Proc. Scalable High Performance Computing Conf.* IEE Press, Williamsburg.

[20] Bacon, D., Chow, J., Ju, D., Muthukumar, K. and Sarkar, V. (1994) A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proc. CASCON'94*, Toronto, November.

[21] Bacon, D., Graham, S. and Sharp, O. (1994) Compiler transformations for high-performance computing. *Comput. Surveys*, **26**, 345–420.

[22] Mace, M. (1987) *Memory Storage Patterns in Parallel Processing*. Kluwer Academic Publishers, Dordrecht.

[23] Knobe, K. Lucas, J. and Dally, W. (1992) Dynamic alignment on distributed memory systems. In *Proc. 3rd Workshop on Compilers for Parallel Computers*, Vienna, pp. 394–404.

[24] Kelly, W. and Pugh, W. (1992) *A Framework for Unifying Reordering Transformations*. Technical Report UMIACS-TR-92-126, University of Maryland.

[25] Torres, J. and Ayguade, E. (1993) Partitioning the statement per iteration space using non-singular matrices. In *Proc. 1993 Int. Conf. on Supercomputing*, Tokyo, July.

[26] Kulkarni, D. (1997) *CDA: Computation Decomposition and Alignment*. Ph.D. Thesis, Department of Computer Science, University of Toronto, Toronto, Canada.

[27] Kulkarni, D. and Stumm, M. (1995) CDA loop transformations. In Szymanski, B. K. and Sinharoy, B. (eds), *Languages, Compilers and Run-time Systems for Scalable Computers*, Boston, May, Chapter 3, pp. 29–42. Kluwer Academic Publishers, Dordrecht.

[28] Abu-Sufah, W. (1978) *Improving the Performance of Virtual Memory Computers*. Ph.D. Thesis, University of Illinois at Urbana-Champaign.

[29] Wolfe, M. (1990) *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA.

[30] Cierniak, M. and Li, W. (1995) Unifying data and control transformations for distributed shared memory machines. In *Proc. ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, La Jolla, CA, June, Vol. 30, pp. 205–217.

[31] Mowry, T. and Gupta, A. (1991) Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, **12**, 87–106.