Optimizing Computational and Spatial Overheads in Complex Transformed Loops

Dattatraya Kulkarni[1] and Michael Stumm[2]

[1] IBM Toronto Laboratory, Toronto, CANADA M3C 1V7
dkulki@vnet.ibm.com
[2] Department of Electrical and Computer Engineering, University of Toronto, Toronto, CANADA M5S 3G4
stumm@eecg.toronto.edu

Abstract. In this paper, we stress the need for aggressive loop transformation techniques, such as CDA (Computation Decomposition and Alignment), that have improved ability to optimize nested loops. Unfortunately, these types of aggressive techniques may also generate complex nested loops with relatively higher overheads. In this paper, we demonstrate that the computational and spatial overhead in complex transformed loops can be effectively reduced, often by simple techniques.

1 Introduction

Techniques for linearly transforming nested loops have matured immensely during the past several years [1,5,8,9,11] to the point where today's production compilers can transform arbitrary perfect loop nests with affine references to arrays and pointer structures that have a single level of indirection [4]. However, we believe that more aggressive techniques are necessary in order to exploit the performance of uniprocessor and multiprocessor systems to the fullest degree possible. The need for more aggressive transformation techniques is even more critical when considering the forthcoming processors with over 1GHz clock frequency. On these processors, the miss penalties for memory references will be large enough to warrant program transformations that improve locality of reference at the cost of a higher computational and spatial overhead in the transformed code. In fact, these processors may well require a new family of optimizations, capable of hiding the multiple cycle latency to access cache memory.

Several groups have been involved in research exploring extensions to the linear loop transformation framework [2,3,6,10]. The extended frameworks are considerably more aggressive and more powerful than the linear loop transformation framework, because the transformations in the new frameworks can alter not only the *execution order* of the iterations, but also the *composition* of the new iterations [7]. In particular, they i) examine computation structures in loops at much finer granularity, ii) explore transformation spaces left unexplored by linear loop transformations, and iii) consider loops that are not perfectly nested.

One undesirable side effect of improved transformation capability is increased code complexity of the transformed loops. The transformed loops may have complex loop

D. O'Hallaron (Ed.): LCR'98, LNCS 1511, pp. 367–377, 1998.
© Springer-Verlag Berlin Heidelberg 1998

control structures and higher computational and spatial overhead when compared to linearly transformed loops. The overhead in linearly transformed loop nests tends to be small relative to the execution time of the original loop. Therefore, it has sufficed for the compilers to use only traditional optimization techniques to minimize the overhead of linearly transformed loops. In the extended frameworks, however, the overhead created is much more significant, requiring techniques to reduce the overhead as much as possible. In this paper we demonstrate that simple techniques exist that can substantially reduce the overhead in complex transformed loops.

2 CDA: A Representative Extended Transformation Framework

Computation Decomposition and Alignment (CDA) is a representative example of a framework that extends the linear loop transformation framework [3,7]. We use it here to illustrate our techniques and thus briefly describe it first in this section.

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 doing so, 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. The choice of which sub-expressions 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 computation 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.

Figure 1 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 3 statements in the body:

$$\begin{aligned}
S_{1.1} &: t(i,j) = A(i-1,j) + A(i-1,j-1) + B(i-1,j) \\
S_{1.2} &: A(i,j) = t(i,j) + B(i,j+1) + A(i,j-1) \\
S_2 &: B(i,j-1) = A(i,j-1) + B(i,j)
\end{aligned}$$

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, $S_{1.1}$.

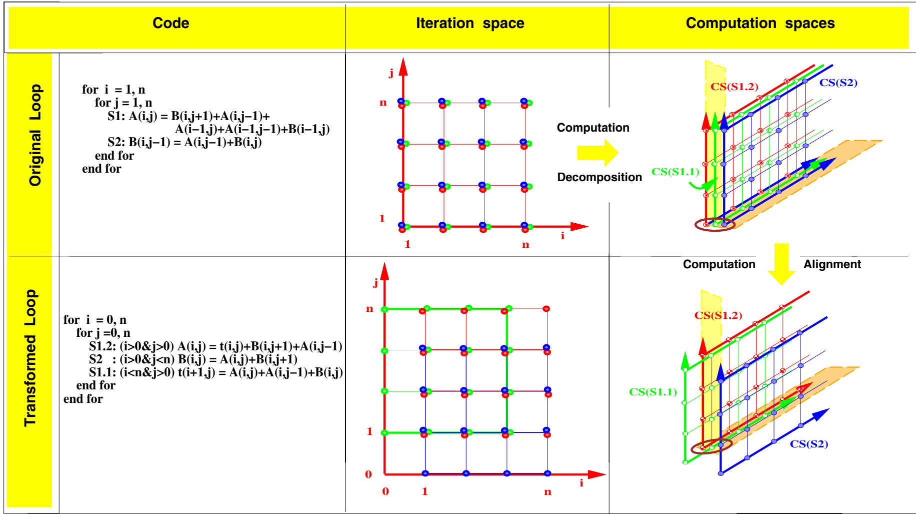


Fig. 1. Application of a simple CDA transformation.

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

$$T_{1.1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad T_{1.2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } T_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

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 1 shows the transformed computation spaces and highlights three computations that are now executed in one iteration. 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. An iteration now no longer necessarily entails the execution of all three statements. The transformed loop thus requires guards that allow a statement to be executed only if appropriate.

```
for i = 0, n
   for j = 0, n
         U(i,j) = c(0) * U(i,j)
         R(i,j) = c(0) * R(i,j)
   end for
end for
```

Fig. 2. The loop used to illustrate the effect of techniques to reduce overheads.

3 Computational and Spatial Overheads

The overheads in complex nested loops, such as those generated by applying CDA transformations, are: i) computational overhead due to empty iterations and guard computations, and ii) spatial overhead for storing temporary variables. In this paper, we briefly outline techniques to reduce both types of overheads. We illustrate the generated overheads and the effectiveness of the techniques to reduce them on the nested loops that result from applying two different CDA transformations on the loop of Figure 2. The loop is deliberately chosen to be simple so that the transformed loops demonstrate only the overheads and none of the benefits.

The transformed iteration space for the first transformation is shown on the left hand side of Figure 3, where one of the computation spaces is applied an offset alignment of (k, k), where k is a positive integer. The left hand side of Figure 4 shows the transformed iteration space for the second transformation, where one of the computation spaces is skewed with respect to the other. We will refer to the CDA transformed loops as *Loop 1* and *Loop 2*, respectively.

The overhead for these two loops was measured on a Sun workstation with hyper-SPARC CPU, and is shown in Figure 3. For the purpose of the experiments, the loop size n was set to 1000 and k was set to 5, unless otherwise specified. The overhead of *Loop 1* with the loop bounds generated by a simplistic algorithm that generates the subsumption of the union of computation spaces is shown as the first five bars on the right hand side of Figure 3. The overhead increases slightly with large k's, due to increasing number of empty iterations and guard computations. For $k = 5$, the overhead is about 22% of the execution time of the original loop. The last five bars on the right hand side of Figure 3 correspond to *Loop 1* optimized using the techniques outlined in the following sections; the overhead is then less than 0.1% of the original loop.

The overheads can also be reduced significantly when the alignments are more general than offsets. The overhead of *Loop 2* with the bounds generated by the simplistic algorithm mentioned above is shown as the first bar on the right hand side of Figure 4. The overhead can be much higher than when using offset alignments (nearly 78% of the original loop in this case), since nearly one quarter of the iterations are empty. However, the overhead is reduced to about 5% of the original loop when *Loop 2* is optimized by removing empty iterations and guards, using the techniques outlined next.

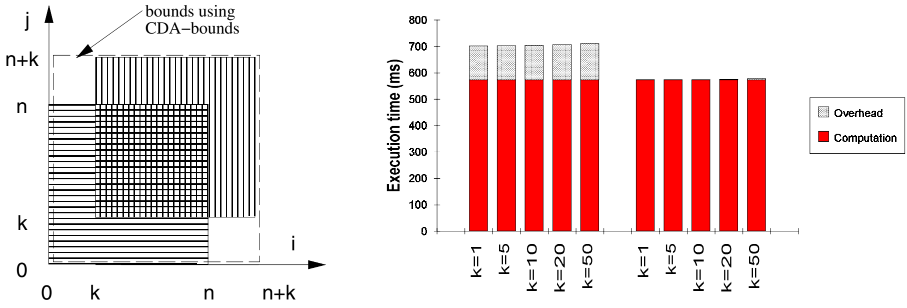


Fig. 3. Overheads in a CDA transformed loop with offset alignments (k, k). The bars on the left correspond to the execution times of *Loop 1* with overheads, whereas the bars on the right correspond to the execution times of *Loop 1* after reducing overheads with the techniques described in this paper.

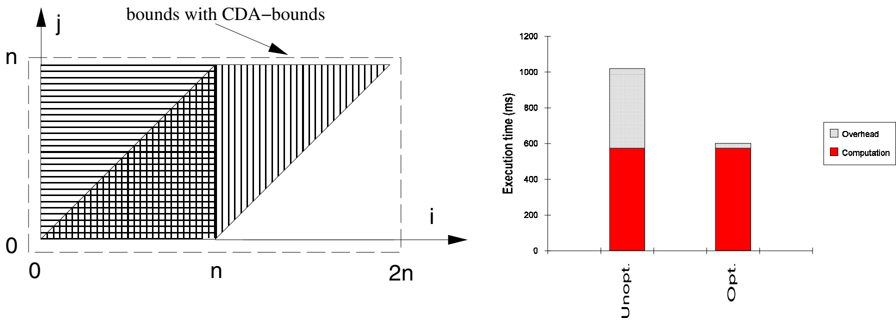


Fig. 4. Overheads in *Loop 2*, CDA-transformed with a linear alignment.

4 Removing Empty Iterations

The iteration space of a CDA transformed loop is the union of the transformed computation spaces projected onto an integer space (which we refer to as the union of computation spaces for conciseness). It is desirable to derive tight loop bounds so that a CDA transformed loop scans integer points in the smallest convex polytope containing the union of computation spaces. With tighter loop bounds, the overhead of empty iterations and the guard computations they contain is reduced.

While deriving tight loop bounds, it is desirable to keep the CDA transformed loop perfectly nested, because it may be necessary to apply other loop transformations in later stages, and most transformations require that the loop be perfectly nested. In order to obtain a perfectly nested CDA transformed loop, the polytope that the loop scans must be convex.[3] Our algorithm removes empty iterations by finding the convex-hull of the union of computation spaces. When the union is a convex polytope itself, then

[3] Only in some cases do non-convex polytopes correspond to perfect nestings.

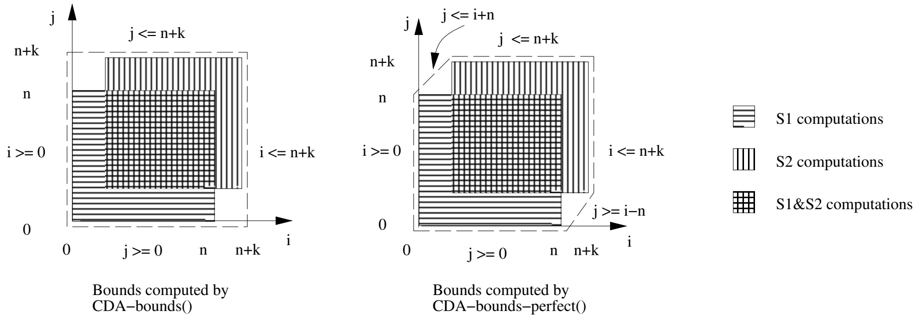


Fig. 5. Empty iterations in an iteration space with tight bounds.

the derived loop bounds are exact in that the transformed loop does not have any empty iterations.

As an example of applying our technique to reduce empty iterations, consider again the transformed computation spaces for *Loop 2* on the left hand side of Figure 4. The algorithm computes the convex-hull as defined by the lines:

$$i = 0, \;\; i = 2n, \;\; j = 0, \;\; j = n, \;\; j = i - n$$

from which the algorithm produces the following inequalities:

$$i \geq 0, \;\; i \leq 2n, \;\; j \geq 0, \;\; j \leq n, \;\; j \geq i - n$$

After variable elimination, these inequalities provide the loop bounds,

$$0 \leq i \leq 2n, \quad max(0, i - n) \leq j \leq n$$

These inequalities bound the shaded area in the figure. The loop bounds are exact in this case, since the union is a convex polygon, so it no longer includes empty iterations.

In some cases, the bounds derived using our algorithm may not remove all empty iterations. Consider the union of the computation spaces of *Loop 1* depicted on the left hand side of Figure 3, where the union is a non-convex polygon. The dotted lines on the left hand side of Figure 5 show the loop bounds that are derived by the simplistic algorithm. The dotted lines at the center of the figure show the bounds obtained by our algorithm.

Figure 6 compares the overhead of the unoptimized *Loops 1* and *2* with the overhead of the optimized loops, where the bounds are derived using our algorithm. The reduction in the overhead of *Loop 1* (of Figure 3) is not significant, since it contains only a small number of empty iterations. The application of our algorithm to *Loop 2* (of Figure 4) reduces the overhead by about 45%, since nearly one quarter of its iterations were empty.

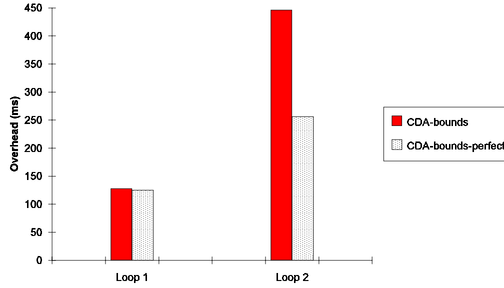
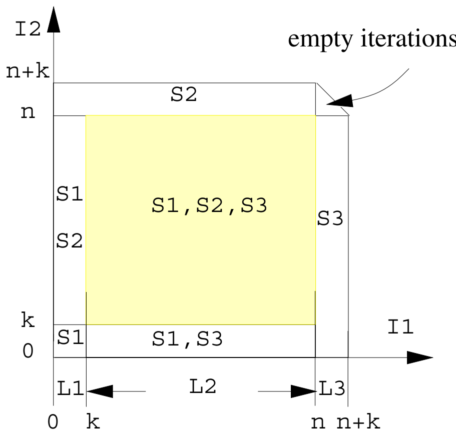


Fig. 6. Performance benefits of eliminating empty iterations.

5 Reducing the Overhead of Guard Computations

Guards are often necessary in CDA transformed loops, both to step off empty iterations and to prevent inappropriate computations from executing in the new iterations. Guards may incur considerable run-time overhead, but in many cases it is possible to remove them. Here, we outline a relatively simple technique that incrementally removes guards from selected regions of the union of computation spaces; however it results in non-perfectly nested loop nests. It is targeted primarily towards CDA transformations, where the intersection of the computation spaces makes up a large portion of the union of the computation spaces. We illustrate the technique with an example, namely the transformed iteration space on the left of Figure 7. The CDA transformations in the figure are such that the computation space of statement S_2 is moved up by k in the I_2 direction with respect to the computation space of statement S_1, and the computation space of statement S_3 is moved right by k in the I_1 direction with respect to the computation space of statement S_1. We refer to the CDA transformed loop corresponding to this iteration space as *Loop 3*. Our algorithm removes guards using the following steps:

1. The bounds of the intersection of the computation spaces are derived. For instance, the shaded area in Figure 7 is the intersection of the three computation spaces. The iterations in the intersection require the execution of all three statements S_1, S_2 and S_3. Therefore, if we partition the new iteration space to separate out the intersection, the code generated for the iterations in the intersection does not require any guards.
2. The iteration space is partitioned along the first dimension I_1 so as to delineate the intersection in that dimension. In our example, the CDA transformed iteration space of Figure 7 is divided into three partitions, namely, L_1, L_2 and L_3, based on the fact that the I_1 bounds for the intersection are k and n. Partition L_1 has iterations with I_1 values between 0 and $k-1$; partition L_2 has iterations with I_1 values between k and n, (the two I_1 bounds for the intersection); and partition L_3 has iterations with the I_1 values between $n+1$ and $n+k$.
3. Code is generated for partition L_2. This code consists of a sequence of subnests. The first subnest includes those iterations with I_2 values that do not belong to the intersection, thus requiring guards. The second subnest includes the iterations that




```
// Code for L1
     ...

// code for L2

for I1 = k, n
   for I2 = 0, k-1
      g(S1)  S1:
      g(S2)  S2:
      g(S3)  S3:
   end for
   for I2 = k, n
             S1:
             S2:
             S3:
   end for
   for I2 = n+1, n+k
      g(S1)  S1:
      g(S2)  S2:
      g(S3)  S3:
   end for
end for

// code for L3
     ...
```

Fig. 7. Transformed computations spaces to illustrate steps in algorithm to remove guards. The transformed loop corresponding to the transformed computation spaces is called *Loop 3*.

belong to the intersection. This code constitutes most of the iterations of the loop that need to be executed and require no guards. The final subnest includes those iterations with I_2 values higher than those of the intersection, thus requiring guards again. The three subnests for our example are shown on the right hand side of Figure 7. Note that the subnest corresponding to the intersection does not have any guard computations.

4. The algorithm is applied recursively to remove guards from partitions L_1 and L_3. The iterations in these partitions contain only a subset of the statements of the original loop body. Thus, only a subset of the computation spaces participate in the intersections of these partitions. Recursive application of the algorithm to partition L_1, does not partition it further along I_1, since the intersection of computation spaces for S_1 and S_2 spans the entire I_1 bounds of L_1. The intersection in L_1 has I_2 bounds of k and n, and guards can be similarly removed from L_1.

The result of applying the guard removal algorithm is thus a sequence of loop nests, which typically are imperfectly nested. The right hand side of Figure 7 shows a template of the code generated for the transformed computation spaces on the left hand side.

Figure 8 shows the effectiveness of the guard removal algorithm. The dark bars correspond to the overhead of CDA transformed code with guards, where the algorithm to remove empty iterations was applied to remove as many empty iterations as possible. The grey bars correspond to code for which the guard removal algorithm was applied. The figure shows that additional removal of guards can reduce the overhead substantially, when the loops are transformed by offset alignments. The reduction in overhead for *Loop 2* was not as large as for *Loops 1* and *3*, since the code for *Loop 2* continues to have

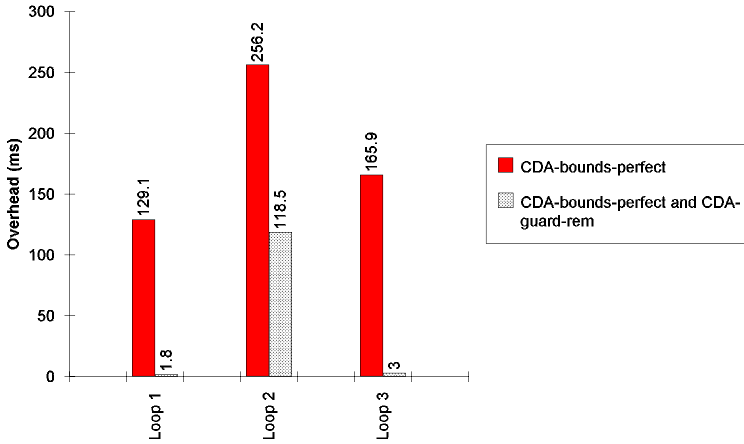


Fig. 8. Performance benefits of removing guards.

guards in nearly one quarter of the iterations, but the benefits of applying the guard removal algorithm is still significant.[4]

6 Optimization of Spatial Overhead for Temporaries

The temporary variables introduced during Computation Decomposition may increase the number of references to memory and may add to space requirements and the cache footprint. A number of optimizations can reduce some of these overheads.

– Temporaries needed in a loop may be replaced by dead variables, which are not used in the later flow of the program.
– While decomposing a statement, it is possible to eliminate the need for temporary variables altogether by using the lhs array elements to store the intermediate results. Such a replacement is legal if the dependence relations remain legal. Even though a Computation Decomposition does not modify dependences, eliminating the temporary variable this way can modify dependences. For example, it is legal to replace $t(i,j)$ by $a(i,j)$ in the following decomposition,

$$a(i,j) = a(i,j) + a(i-1,j) + a(i,j-1) \quad \Rightarrow \quad \begin{array}{l} \overset{a(i,j)}{\cancel{t(i,j)}} = a(i,j) + a(i-1,j) \\ a(i,j) = \overset{a(i,j)}{\cancel{t(i,j)}} + a(i,j-1) \end{array}$$

However, such a replacement would be illegal in the following decomposition, because $a(i,j)$ would be modified before it is used in the second statement so the temporary variable needs to be retained.

[4] These iterations correspond to the region bounded by $0 \leq i \leq n$ and $i+1 \leq j \leq n$ on the left of Figure 4.

$$a(i,j) = a(i,j) + a(i-1,j) + a(i,j-1) \qquad \begin{array}{l} t(i,j) = a(i-1,j) + a(i,j-1) \\ \Rightarrow a(i,j) = t(i,j) + a(i,j) \end{array}$$

Hence, storage requirements can be reduced in this way for only some decompositions. Moreover, note that the dependences introduced by replacing the temporary variable can constrain later opportunities for Computation Alignment. It is therefore better to replace the references to the temporary by references to the lhs after the CDA transformation.

- Temporary variables that were introduced in one loop can be reused in subsequent loops. This is possible since the temporaries are intended to store only the results inside a loop, and these results are not needed outside the loop.
- Temporary arrays are typically initially chosen to have the same dimension and size as the iteration space, since the subexpressions that generate values for the temporaries potentially have a new value in each iteration. The dimension and size of the temporary arrays can be reduced following the CDA transformation. It is only necessary to have as many storage locations as there are iterations between when the temporary is defined and when it is used. For simple offset alignments, the size of the temporary arrays can be just a fraction of the size of the iteration space. For example, consider the decomposition of a statement S in a two dimensional loop into statements S_1 and S_2. The results of S_1 are stored in a temporary array t. When statement S_1 is aligned to statement S_2 along the outer loop level by an offset c, then t need only be of size $c \times n$, assuming n iterations in the inner loop.

7 Concluding Remarks

In this paper, we observed the need for aggressive transformation techniques that have improved ability to optimize nested loops, but that may also generate complex nested loops with attendant overhead. We demonstrated that the computational and spatial overhead in these complex transformed loops can be effectively reduced, often by simple techniques. We believe that with the reduction in overhead achieved using the techniques we have described, complex loop transformations become suitable for integration into production compilers.

References

1. Banerjee, U. (1990) Unimodular transformations of double loops. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.
2. Kelly, W. and Pugh, W. (1992) A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126, University of Maryland, 1992.
3. Kulkarni, D. (1997) CDA: Computation Decomposition and Alignment. PhD thesis, Department of Computer Science, University of Toronto, 1997.
4. Kulkarni, D. et al. (1997) XL Fortran Compiler for IBM SMP Systems. *AIXpert Magazine*, December 1997.
5. Kulkarni, D., Kumar, K.G., Basu, A., and Paulraj, A. (1991) Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

6. Kulkarni, D. and Stumm, M. (1995) CDA loop transformations. In *Chapter 3, Languages, compilers and run-time systems for scalable computers, B.K. Szymanski and B. Sinharoy (eds)*, pages 29–42, Boston, May 1995. Kluwer Academic Publishers.
7. Kulkarni, D. and Stumm, M. (1997) Linear and Extended Linear Transformations for Shared Memory Multiprocessors. *The Computer Journal*, 40(6), pp. 373-387, December 1997.
8. Kumar, K.G., Kulkarni, D., and Basu, A. (1992) Deriving good transformations for mapping nested loops on hierarchical parallel machines in polynomial time. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, July 1992.
9. Li, W. and Pingali, K. (1994) A singular loop transformation framework based on non-singular matrice. *International Journal of Parallel Programming*, 22(2), 1994.
10. Torres, J. and Ayguade, E. (1993) Partitioning the statement per iteration space using non-singular matrices. In *Proceedings of 1993 International Conference on Supercomputing, Tokyo, Japan, July 1993.*, 1993.
11. Wolf, M and Lam, M. (1991) A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, 1991.