18 Collective Loop Fusion for Array Contraction

G. Gao and R. Olsen
McGill University, Montréal, Québec, Canada
V. Sarkar
IBM Palo Alto Scientific Center, Palo Alto, California
R. Thekkath
University of Washington at Seattle

Abstract

In this paper we propose a loop fusion algorithm specifically designed to increase opportunities for array contraction. Array contraction is an optimization that transforms array variables into scalar variables within a loop nest. In contrast to array elements, scalar variables have better cache behavior and can be allocated to registers. In past work we investigated loop interchange and loop reversal as optimizations that increase opportunities for array contraction [13]. This paper extends this work by including the loop fusion optimization. The fusion method discussed in this paper uses the maxflow-mincut algorithm to do loop clustering. Our collective loop fusion algorithm is efficient, and we demonstrate its usefulness for array contraction with a simple example.

1 Introduction

Loop optimization plays a critical role in the compiler optimization of scientific programs. Loops are primarily used to read and write large arrays of values. The cost of array accesses can be reduced by transforming loop nests for cache locality [11, 16, 6]. However, inspite of high cache hit rates, array accesses can incur a large overhead. Because array variables cannot be allocated to registers, load and store instructions must be issued to move the array values between the cache and the register set (these instructions typically comprise about 30% of the total instructions executed). These load and store instructions interfere with instruction parallelism because of serializations imposed by the memory system and by the pessimistic memory aliasing assumptions that must be made by the instruction scheduler. This limitation on instruction parallelism is a serious problem for modern processor architectures that depend heavily on instruction parallelism to achieve high performance.

Our approach is based upon performing loop fusion to increase the opportunities for array contraction [13]. Array contraction replaces an array variable by a scalar variable or by a buffer containing a small number of scalar variables. This replacement usually eliminates many load/store instructions because scalar variables are good candidates for register allocation [4]. A secondary benefit arises from the fact that loop fusion increases the number of instructions in a loop body.

The algorithm for collective loop fusion presented in this paper is novel in its use of a cluster numbering scheme followed by one or more applications of the maxflow-mincut algorithm [14]. Unlike other loop fusion algorithms that are based upon fusing pairs of loops, our algorithm operates collectively on all loop nests. Our approach is provably optimal for the special case of two-cluster partitioning for which the max-flow algorithm is known to yield a cut set of minimum size. We believe that our approach is generally better than the pairwise greedy approach, though this can only be confirmed by performance comparisons on realistic test cases.

The rest of the paper is organized as follows: Section 2 describes the *loop dependence graph* which is our program representation. Section 3 formally states the optimization problem, and Section 4 describes our solution. Section 5 presents a case study and experimental results. Section 6 discusses related work, and Section 7 the conclusion and future work.

2 Program Representation

The program representation assumed for performing collective loop fusion is a *Loop Dependence Graph* (LDG). The loop dependence graph represents a single-entry, single-exit region consisting of k *identically control dependent* perfect loop nests [5], with conformable loop bounds. A node in the LDG represents a loop nest [17]; an edge in the LDG represents a loop-independent data dependence from the source loop nest to the destination loop nest [2]. Each edge in the LDG is marked as being *fusible* or *nonfusible*. The source and destination loop nests of a nonfusible LDG edge cannot be fused because this would violate the data dependence test for loop fusion [17]. Fusible edges are further classified as *contractable* and *noncontractable*. A contractable edge is one where loop fusion will yield a savings in memory cost because a data computed by a source loop nest will be available in the processor's registers for use by a destination loop nest. A noncontractable edge is one for which loop fusion will not yield such a savings in memory cost. Only flow edges can be marked contractable. Figure 1 contains a simple example with six FORTRAN loop nests. The Loop Dependence Graph for Figure 1 is shown in Figure 2(a) in Section 4, where the nonfusible edges are marked with X and the fusible-noncontractable edge

```
      DO 10 I=1, N
10       A(I) = E(I)
      DO 20 I=1, N
         B(I) = A(I) * 2 + 3
20       C(I) = B(I) + 99
      DO 30 I=1, N
30       D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40       E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
         F(I) = B(I) * 4 + 2
50       G(I) = E(I) * 8 - F(I)
      DO 60 I=1, N
60       H(I) = F(I) + G(I) * E(N-I+1)
```

Figure 1: A FORTRAN Code Fragment

is marked with an asterisk. The nonfusible edge from v_1 to v_3 occurs due to array A being produced and consumed in reverse order; similarly, the nonfusible edge from v_4 to v_6 occurs due to array E being produced and consumed in reverse order (in both cases, the data dependence test for loop fusion is used to determine that the edges are nonfusible). There is a fusible noncontractable edge from v_1 to v_4 due to an anti-dependence on array E. All other edges are fusible and contractable. The nodes of the LDG are numbered according to a topological sort of the partial order defined by the LDG's dependence edges.

A *fusion partition* of an LDG is a partition of the set of nodes into disjoint *fusible clusters*; each fusible cluster represents a set of loop nests to be fused. A fusion partition is *legal* if and only if *1)* for each nonfusible edge, the source and destination nodes belong to distinct fusible clusters, and *2)* the reduced graph defined by the fusion partition is acyclic. Given an LDG and a legal fusion partition, the output code configuration can be obtained by fusing all loops that belong to the same fusible cluster and by ordering the fused loops according to some topological sort defined by the edges in the reduced graph.

The assumption that all loop nests have conformable loop bounds is a valid assumption satisfied by loop nests obtained from loop distribution or from the Fortran-90 WHERE statement. However, the results described in this paper are also applicable to any LDG in which some nodes correspond to loop nests of size and shape that are nonconformable with the remaining loop nests (this includes the degenerate but common case in which a "nonconformable loop nest" is just a simple statement). This more general case can be handled by marking as nonfusible any LDG node corresponding to a non-conformable loop nest and by placing such a node in a cluster by itself (with all edges adjacent to a nonfusible node also being marked as nonfusible).

3 Problem Statement

In this paper we are interested in the following optimization problem:

Problem. *Given a Loop Dependence Graph (LDG), find a legal fusion partition that maximizes the number of fused contractable edges, i.e., the contractable edges for which the source and destination loop nests are placed in the same fusible cluster.*

We assume that the memory cost savings from each fused contractable edge is the same, say, equal to 1 unit, which denotes the cost of loading a data value from memory. Thus, maximizing the fused contractable edges maximizes the savings in memory cost. For simplicity we ignore the cost of storing a data value because *1)* the elimination of this cost depends upon whether this value is needed later during program execution and *2)* because a store-buffer often hides the cost of stores. Since all loops have conformable loop bounds, the number of iterations need not appear as a factor in the edge cost. However, it is possible for either a store or load operation to be contained within a conditional. This case can be more accurately modeled by using an edge cost <1, based upon the execution probabilities for the conditionals [12]. For the sake of simplicity, we approximate this case by a unit cost as well. The cost model can be extended to take into account conditional probabilities. We also implicitly assume that suitable loop transformations have already been performed on the individual loop nests before applying the fusion algorithm (see [13] or other related work).

4 A Solution based upon Network-Flow

A graph that has no nonfusible nodes or edges can be trivially fused into a single cluster. Let $G = [V, E]$ be an acyclic Loop Dependence Graph with at least one nonfusible edge or node. The algorithm consists of two steps: 1) perform initial cluster assignment to nodes satisfying all fusion legality constraints and build a reduced graph, and 2) apply the maxflow algorithm repeatedly on the reduced graph to determine a final cluster assignment. These two steps are summarized in Figures 3 and 4 and discussed in detail in subsections 4.1 and 4.2.

Figure 2(a) shows a graph $G = [V, E]$ with six vertices, $v_1, \ldots, v_6$. The nonfusible edges are marked with an X, and the fusible-noncontractable edge is marked with an asterisk. All unmarked edges are both fusible and contractable.

4.1 Construction of the Reduced Graph

Step 1 makes tentative cluster assignments to the vertices of the graph. An assignment of a vertex to a single cluster is considered to be permanent. The decision for all other vertices is done in Step 2.

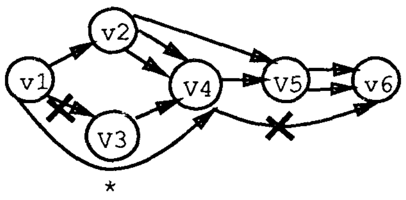

Figure 2(a)

Cluster Set of vertices

STEP	v1	v2	v3	v4	v5	v6
1	{1}	{1}	{2}	{2}	{2}	{3}
2	{1}	{1,2}	{2}	{2}	{2,3}	{3}

Figure 2(b)

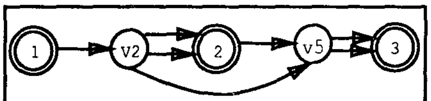

Figure 2(c)

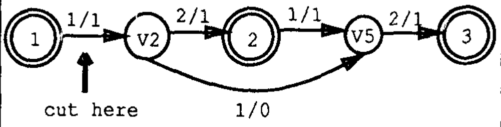

Figure 2(d)

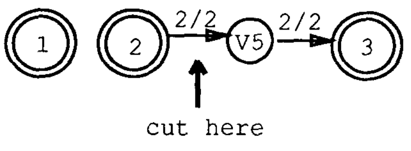

Figure 2(e)

member(1)={v1}
member(2)={v2,v3,v4}
member(3)={v5,v6}

figure 2(f)

Figure 2: Cluster Assignment and Maxflow Application

Each vertex v has a associated cluster set, $Cluster(v) = \{a \mid v \in \text{cluster } a\}$. Initially all source vertices are assigned a cluster number of one. Then the vertices of G are visited in topological order and assigned the maximum cluster number of all preceding vertices, unless there is a nonfusible edge.

The vertices are then traversed in reverse topological order to find all higher-numbered clusters to which each vertex can belong without violating any legality constraints. Figure 2(b) shows the cluster sets created for each vertex. Three clusters are created in this assignment.

From this cluster assignment, a reduced graph $G' = [V', E']$ is built. The key idea here is to collapse vertices that *must* belong to the same cluster and to eliminate all redundant edges. The vertices V' of the reduced graph consist of two types of nodes: *1)* **Cluster Nodes**, created by collapsing vertices that have a singleton $Cluster()$ set, and *2)* **Vertex Nodes**, an original vertex from V, where the cardinality of its $Cluster()$ set is greater than 1 and whose final cluster assignment is therefore still undetermined.

Purpose: compute the minimum number of clusters possible and assign tentative cluster numbers to all the vertices. Then use this numbering to construct a reduced graph.
Input: an acyclic graph $G = [V, E]$
Output: an acyclic reduced graph $G' = [V', E']$ of G
Procedure:

1. Traverse the vertices of G in topological order to find the minimum cluster number for each vertex;
   For each vertex $v \in V$,
   (a) if v has no incoming edges, assign $Cluster(v) = 1$.
   (b) if v has incoming edges,
   $$Cluster(v) = \texttt{MAX}\left(\forall(u,v) \in E, \begin{cases} Cluster(u) + 1, & \text{if } (u,v) \text{ is nonfusible} \\ Cluster(u), & \text{otherwise} \end{cases}\right)$$
2. Traverse the nodes of the graph in reverse topological order to find all the higher-numbered clusters to which each vertex may belong.
   For each vertex $v \in V$ that has at least one out edge,
   for each out edge $(v, w) \in E$,
   (a) if $Cluster(w) = Cluster(v)$, i.e., the sets are identical, then go to the next out edge,
   (b) if $Cluster(w) > Cluster(v)$, consider every $a \in Cluster(w)$ such that $a >$ every element in $Cluster(v)$; add a to $Cluster(v)$ only if both conditions given below are false:
      i. there is some vertex y such that $a \in Cluster(y)$ and there is a path from v to y such that this path has a nonfusible edge (in this case, adding a to $Cluster(v)$ might violate the constraint that prohibits nonfusible edges within the same cluster).
      ii. there is some vertex y such that $a \in Cluster(y)$ and there is a path from v to y such that, this path goes through another vertex t such that, $Cluster(u) \leq Cluster(t) < Cluster(y)$, (in this case, adding a to $Cluster(v)$ might violate the acyclic constraint).
3. Now construct the reduced graph $G' = [V', E']$ by defining V' and E' as follows:
   For a cluster a, define $\texttt{member}(a) = \{v \mid v \text{ is a vertex assigned to this cluster}\}$
   (a) For each vertex $v \in V$,
      i. if $|Cluster(v)| > 1$, then $v \in V'$,
      ii. if $|Cluster(v)| = 1$ and $Cluster(v) = \{a\}$, then add a vertex a to V', if it doesn't already exist. Also add v to $\texttt{member}(a)$.
   (b) For each edge $(u, v) \in E$, if $Cluster(u) \cap Cluster(v) \neq \phi$ and (u, v) is contractable, then $(u, v) \in E'$.

Figure 3: Step 1 of the Partitioning Algorithm

Input: the reduced graph $G' = [V', E']$
Output: a fully partitioned graph
Procedure: To each connected component of G', apply the following:

1. If the component has a single cluster node, add any vertex node in this component to this cluster.
2. If the component has multiple cluster nodes, further partition the component by applying the maxflow algorithm:
   - (a) Assign an edge capacity value of 1 to all edges in this component. Replace multiple edges by a single edge whose capacity is the sum of the individual capacities.
   - (b) Apply maxflow to the graph. Traverse the graph finding all edges where $flow = capacity$; the graph can potentially be cut in all of these places, and these cut edges must disconnect the graph. Ignore those cut edges that do not disconnect the graph. Do not disconnect a vertex node from all cluster nodes.
   - (c) Repeat the above steps until no more vertex nodes are left.

Figure 4: Step 2 of the Partitioning Algorithm

The set E' is constructed from E by 1) eliminating all edges between vertices that can *never* belong to the same cluster, i.e., when the intersection of the $Cluster()$ sets for these vertices is empty, and 2) eliminating all edges having `cost` $= 0$, since such edges do not determine the total cost of a fusion partition.

Figure 2(c) shows the reduced graph for the example in Figure 2(a). The cluster nodes are shown as double-circled nodes. The `member` set of each cluster is: `member(1)` $= \{v_1\}$, `member(2)` $= \{v_3, v_4\}$, and `member(3)` $= \{v_6\}$. The vertex nodes are v_2 and v_5. Note that all nonfusible edges have been deleted in Figure 2(c), as well as the fusible noncontractable edge (v_1, v_4), since this is a zero-cost edge that will not contribute to further partitioning.

4.2 Application of the `maxflow` Algorithm

In this step, each vertex node $v \in V'$, is assigned some element from its cluster set $Cluster(v)$, so as to minimize the number of inter-cluster edges. We use the maxflow algorithm to make this determination. Recall that the maxflow algorithm is applied to a directed graph that has two distinguished vertices, a *source* s and a *sink* t, and each edge in the graph has an associated positive capacity. A *flow* through the graph is an assignment of flow values to each edge such that 1) the flow value never exceeds the capacity of the edge, and 2) for all vertices except the source and the sink, the total flow into a vertex is equal to the total flow out of the vertex. A *cut* is a set of edges that separate s from t, that is, a cut partitions the vertex set of the graph into two components, S and T, such that $s \in S$ and $t \in T$, where $V = S \cup T$ and $S \cap T = \phi$. The capacity of a cut is the sum of the capacities of all the edges in the cut. A flow through the graph can never exceed the capacity of any cut. And, for a given maximum flow through the graph, there must exist a cut whose capacity is equal to the maximum flow; such a cut is called the *minimum cut* [14].

A reduced graph with only one cluster node can assign all its vertex nodes to this cluster. But when there are multiple cluster nodes, then the maxflow algorithm is applied. When a capacity of 1 is assigned to all edges, the minimum cut obtained from the maximum flow gives the minimum number of edges which partition the vertices of the graph into S and T. Given a maximum flow, the set of edges that comprise the minimum cut is easily found [14]. The maxflow algorithm is applied repeatedly until there remain no unassigned vertex node.

This algorithm terminates since each application of the maxflow algorithm partitions the graph into two components that are each smaller than the original. Sometimes the graph can be partitioned into more than two components. This is done by attempting to cut the graph at every edge where flow equals capacity except 1) when a vertex node is disconnected from all cluster nodes in the component, and 2) when this edge does not disconnect the graph.

Figures 2(d) , 2(e), and 2(f) show the application of the maxflow algorithm to the reduced graph in Figure 2(c). Figure 2(d) shows the capacity and the flow on each edge of the graph. Note that the two edges between the vertex node v_2 and the cluster node 2 have been replaced by a single edge of capacity 2. A similar replacement was made between nodes v_5 and 3. The maximum flow specifies the cut at the edge between vertex 1 and vertex v_2; this cut allows us to add v_2 to cluster 2, since this is the only cluster number left in v_2's cluster set (see Figure 2(b)). In the remaining graph, the edge between v_2 and v_5 creates another edge between the cluster node 3 and vertex node v_5. Figure 2(e) shows the application of maxflow to this new graph. This places v_5 into cluster 3, and the graph is fully partitioned.

4.3 Special Cases

For the sake of conciseness, the basic algorithm described above omits special-case details. For example, if the graph contains nonfusible nodes, then these nodes contribute to the acyclic partitioning of the reduced graph but they must be assigned to distinct and separate clusters. Hence they are used in the initial step of assigning cluster numbers, and are thereafter deleted along with their cluster numbers and edges.

A second situation is one where the input graph G is disconnected. The algorithm is applied once to each component of G. Once all components are partitioned, all equal-numbered clusters from each component are merged to obtain a complete partition of the graph. Care is taken to avoid the merger with a nonfusible node.

Thirdly, the graph G', or one of its subcomponents may have multiple source or sink nodes. In such situations, before applying the maxflow algorithm, we create a *pseudo source* vertex and create edges from this pseudo source to all real sources. Similarly, we create a *pseudo sink* vertex with edges from all real sinks to the pseudo sink vertex. These pseudo edges are assigned a large capacity, far greater than the maximum flow possible through the graph to ensure that a pseudo edge will never be a minimum cut edge.

4.4 Algorithm Complexity

We assume that the graph has n vertices and m edges. The three parts of Step 1 of the algorithm take $O(n+m) + O(nm) + O(n+m) = O(nm)$.

The second step of the algorithm operates on the reduced graph, G', assume that the number of its vertices and edges are n_r and m_r, respectively. In the worst case, each application of the maxflow algorithm isolates only one vertex, therefore requiring $n_r - 1$ applications of maxflow. If we assume that the maxflow algorithm on a graph takes time $O(n_r m_r \log n_r)$ [14], the total time taken for Step 2 is $O(n_r^2 m_r \log n_r)$. Thus the total time for the algorithm is $O(nm) + O(n_r^2 m_r \log n_r)$. Note that the second term of this expression will dominate when the first step fails to reduce the size of the graph by a significant amount.

5 A Case Study

In this section we examine the effectiveness of alternate partitioning strategies. Our purpose is merely to illustrate the performance penalty of a naive partitioning, as opposed to effective partitioning done using collective loop fusion and array contraction. We use an example program shown in Figure 5(a). This program is similar to the one used in the previous section. The corresponding loop dependence graph is very similar to the graph shown in Figure 2(a), without the the edges between nodes 1 and 4 and between nodes 2 and 5.

The loop nests have two nonfusible data dependences which can be detected by the compiler, from loop 1 to loop 3 and from loop 4 to loop 6 [13, 10]. During the compilation process, a compiler might transform the collection in any of the following ways: *a)* perform no transformation, as shown in Figure 5(a), *b)* perform array contraction on the original cluster without performing any fusion, as shown in Figure 5(b), *c)* naively partition the LDG, possibly cutting more edges than necessary, and perform the corresponding loop fusion, as shown in Figure 5(c), *d)* naively partition the loops and perform array contraction within each partition, as in Figure 5(d), or *e)* efficiently partition the graph and stop there, as shown in Figure 5(e), or *f)* efficiently partition the graph and perform array contraction within the loop bodies of the respective partitions, as in Figure 5(f). Notice that in some instances the code was rewritten to assist the compiler with array-element recognition. The particular modification is shown in the array-contracted original collection, Figure 5(b), where scalar references are used for array elements to avoid redundant array accesses.[1]

The greedy algorithms described in [1] and [7] would result in the naive partitioning depicted by Figure 5(c). The random method of picking loops to be fused, as described by [15], would yield an unpredictable clustering. The second algorithm described in [7] is more sophisticated, and generates a clustering close to optimal, as shown in Figure 5(e). This partitioning has one less fused contractable edge.

The results of partitioning based upon timings taken on the SPARC server are shown in Table 1(a). The table shows both optimized and nonoptimized versions for integer, single-precision floating point, and double-precision floating-point arrays. The same test codes were timed on an IBM RISC System/6000 (RS/6000) and the results are shown in Table 1(b). Because the test code segments were short, they were each executed five-hundred times within an outer loop to obtain improved timing-function resolution.

The statistics reported in Table 1 use the metric *eps*, or output array elements generated per second, i.e., how quickly elements of the output array H are produced. The eps number is derived from the slope of the line generated by regressing data points over a range of vector sizes between 60 elements and 3000 elements, at 60-element intervals. The advantage of this metric over a simple average is that it tends to smooth away system anomalies. To assist in interpreting these results, we show the speedup due to the various loop transformations in Table 2

In general, little improvement occurred as a result of applying array contraction to the original loop (refer to Figure 5(b) and the top row of Table 2). This was because no arrays were actually eliminated by the contraction. Instead, a few memory references were replaced by register references, and these register references represented only a small portion of the overall loop-cluster computation. In the case of the RS/6000, it actually resulted in lower performance, cf. Table 1(b). The transformation does, however, provide a basis for comparing the other loop-transformation alternatives since it allows us to isolate partitioning effects.

When the loop cluster was naively partitioned as in Figure 5(c), the benefit was again small, if at all, because only a little loop-control overhead was eliminated, which was small when compared to the overall computation, cf. second row of Table 2. Likewise, the efficiently partitioned loops in Figure 5(e) showed little improvement for the same

[1]The compilers we used failed to recognize such references as scalar identities and therefore unnecessarily reloaded the previously written array element again from memory, rather than using the value which is still in register, see Callahan et al. [4].

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         B(I) = A(I) * 2 + 3
20       C(I) = B(I) + 99
      DO 30 I=1, N
30       D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40       E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
         F(I) = E(I) * 4 + 2
50       G(I) = E(I) * 8 - 3
      DO 60 I=1, N
60       H(I) = F(I) + G(I) * E(N-I+1)
```

(a) **Original Loop Collection**

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         b = A(I) * 2 + 3
         B(I) = b
20       C(I) = b + 99
      DO 30 I=1, N
30       D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40       E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
         e = E(I)
         F(I) = e * 4 + 2
50       G(I) = e * 8 - 3
      DO 60 I=1, N
60       H(I) = F(I) + G(I) * E(N-I+1)
```

(b) **Array-Contracted Original Collection**

```
      DO 10 I=1, N
         A(I) = I
         B(I) = A(I) * 2 + 3
10       C(I) = B(I) + 99
      DO 20 I=1, N
         D(I) = A(N-I+1) + 6
         E(I) = B(I) + C(I) * D(I)
         F(I) = E(I) * 4 + 2
20       G(I) = E(I) * 8 - 3
      DO 30 I=1, N
30       H(I) = F(I) + G(I) * E(N-I+1)
```

(c) **Fused Loops from Naive Partitioning**

```
      DO 10 I=1, N
         a = I
         b = a * 2 + 3
         A(I) = a
         B(I) = b
10       C(I) = b + 99
      DO 20 I=1, N
         d = A(N-I+1) + 6
         e = B(I) + C(I) * d
         E(I) = e
         F(I) = e * 4 + 2
20       G(I) = e * 8 - 3
      DO 30 I=1, N
30       H(I) = F(I) + G(I) * E(N-I+1)
```

(d) **Array-Contracted Naive Partitioning**

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         B(I) = A(I) * 2 + 3
         C(I) = B(I) + 99
         D(I) = A(N-I+1) + 6
20       E(I) = B(I) + C(I) * D(I)
      DO 30 I=1, N
         F(I) = E(I) * 4 + 2
         G(I) = E(I) * 8 - 3
30       H(I) = F(I) + G(I) * E(N-I+1)
```

(e) **Fused Loops from Efficient Partitioning**

```
      DO 10 I=1, N
10       A(I) = I
      DO 20 I=1, N
         b = A(I) * 2 + 3
         c = b + 99
         d = A(N-I+1) + 6
20       E(I) = b + c * d
      DO 30 I=1, N
         e = E(I)
         f = e * 4 + 2
         g = e * 8 - 3
30       H(I) = f + g * E(N-I+1)
```

(f) **Array-Contracted Efficient Partitioning**

Figure 5: Sample Collection of Loops

(a) Performance of Collective Loop Transformations on a Sun SPARC Server 4/490†

Phase of	integer		single precision		double precision	
Transformation	f77	f77 -O	f77	f77 -O	f77	f77 -O
original loop collection	90	165	117	201	86	134
array-contracted original	97	168	117	216	82	136
fused naive partition	103	171	137	209	96	133
array-contracted naive part.	124	191	136	239	95	154
fused efficient partition	113	178	137	214	97	135
array-contracted efficient part.	149	225	151	330	116	220

(b) Performance of Collective Loop Transformations on an IBM RISC System/6000†

Phase of	integer		single precision		double precision	
Transformation	xlf	xlf -O	xlf	xlf -O	xlf	xlf -O
original loop collection	195	939	201	1154	195	1202
array-contracted original	191	933	197	1142	194	1201
fused naive partition	245	1089	253	1104	239	1239
array-contracted naive part.	259	1109	247	1139	248	1325
fused efficient partition	247	1198	248	1306	252	1346
array-contracted efficient part.	322	1541	332	1363	329	2494

† Results are in thousands of output elements produced per second (eps), based upon regression of vectors within the range 60–3000.

Table 1: Effect of Loop Partitioning

reason, cf. fourth row of Table 2. Another potential benefit from fusion is the increase in the size of the loop bodies, which increases the pool of instructions available for instruction scheduling. However, compared with the small loop bodies of the original loops, the larger fused loops made little difference.

The improvements obtained by array contraction after fusion were marginally better than that of the other cases discussed so far (see Figure 5(d) and row three of Table 2). The main reason was the elimination of an array. The array-contracted efficient partitioning (Figure 5(f)), showed much greater improvement—up to a sixty percent over the original code for floating-point arrays, on the Sun, and a two-fold performance improvement for double-precision floating point arrays, on the RS/6000 (see the last row of Table 2). The reason is obvious: five of the eight arrays in the original loop collection were eliminated and replaced by scalar variables, and of the three remaining arrays, one was the output array H and the other two (A and E) represent the nonfusible edges in the LDG.

Table 2: Speedup due to Collective Loop Transformation over Performance of the Original Code

Phase of Transformation†	Sun 4/490			IBM RS/6000		
	int.	sgl.	dbl.	int.	sgl.	dbl.
array-contracted original	1.0	1.1	1.0	1.0	1.0	1.0
fused naive partition	1.0	1.0	1.0	1.2	1.0	1.0
array-contracted naive part.	1.2	1.2	1.1	1.2	1.0	1.1
fused efficient partition	1.1	1.1	1.0	1.3	1.1	1.1
array-contracted efficient part.	1.4	1.6	1.6	1.6	1.2	2.1

These tests demonstrate the importance of efficient partitioning. The benefit derives not from loop fusion or array contraction alone, but from a combination of the two transformations.

6 Related Work

A lot of the past work in optimizing the performance of loops has focused on individual loops rather than on collections of loops [9, 3, 17]. Loop distribution and loop fusion are two well known loop transformations which deal with multiple loop nests [17]. Loop fusion is useful in 1) reducing loop overhead, 2) increasing the size of loop bodies, which can affect instruction-scheduling, and 3) for better register allocation and memory performance. The effects of loop fusion upon virtual memory management and register assignment has been studied by Kuck et al. [9]. Loop fusion for vector machines has been studied in [2, 1].

Warren describes a fusion algorithm which merges a collection of loops, allowing the replacement of temporary arrays by scalars [15]. His method however, does not indicate how to perform fusion to minimize the cost of array accesses using scalar replacement. Goldberg and Paige describe loop fusion as an optimization technique to find an "optimal schedule" by minimizing the number of fusible loop clusters [7]. The loop fusion problem studied in this paper goes beyond simple loop-cluster minimization and instead tries to minimize the data-access cost function.

A recent study [8], uses loop fusion to maximize loop parallelism and to improve data locality. In this independent study, they use the maxflow algorithm to do graph partitioning for loop fusion. They use edge weights, similar to our approach, to denote the reuse of arrays across different loop nests. In our approach, we perform a pre-processing step that pre-assigns some nodes to clusters based on the fusion-preventing edges. This pre-assignment allows us reduce the size of the input graph to the maxflow algorithm. We believe that this size reduction has a good payoff in the running time, since the complexity of the overall algorithm is dependent upon the complexity of the maxflow algorithm.

Wolfe uses the term "array contraction" to denote reducing the size of a compiler-generated temporary array created by scalar expansion [17]. Allen discusses the problem of minimizing temporary array storage in a single loop nest by a technique called *sectioning* [2]. Register allocation of subscripted variables (arrays) has been recognized as a challenging optimization problem and studied by Callahan et al., but their work focused on single loop nests as well [4].

In our previous work we investigated the problem of performing loop interchange and loop reversal on individual loop nests so as to maximize the number of array accesses that are potential candidates for array contraction [13]. This paper extends our previous work by performing collective loop fusion Thus, our overall framework considers loop optimizations both at the individual loop-nest level and at a collective level.

7 Conclusions and Future Work

We have presented a method for performing collective loop fusion to enhance opportunities for array contraction. We represent a set of loop nests by a loop dependence graph and use the maxflow algorithm to partition this graph into clusters. We have demonstrated the effectiveness of this method via a simple example. These results suggest that the method can be useful in the optimization of scientific programs with loops nests.

As future work we plan to investigate the use of our partitioning technique together with advanced loop scheduling techniques like loop unrolling and software pipelining. These techniques facilitate a greater use of instruction-level parallelism within fused loops. We also plan to study the effectiveness of our algorithm on large benchmarks. Based upon these results, we intend to implement the partitioning method in the compiler testbed currently under development at McGill University.

Acknowledgments

We gratefully acknowledge the support from IBM Corporation, the National Science and Engineering Research Council (NSERC). Radhika Thekkath was supported in part by NSF PYI Award #MIP-9058-439.

References

[1] J. R. Allen and K. Kennedy. Vector register allocation. Technical Report TR86-45, Rice University, Houston, TX, December 1986.

[2] John R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation.* PhD thesis, Rice University, 1983.

[3] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, 1987.

[4] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990. White Plains, NY.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[6] Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash. On estimating and enhancing cache effectiveness. *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991. To appear in Springer Verlag's Lecture Notes in Computer Science series.

[7] Allen Goldberg and Robert Paige. Stream processing. In *1984 ACM Symposium on Lisp and Functional Programming*, pages 53–62, Austin, TX, August 1984.

[8] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. Technical report, Rice University, August 1992. Rice COMP TR92-189.

[9] D. J. Kuck, Kuhn R., D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[10] Russell Olsen. Analysis and transformation of loop clusters. Master's thesis, McGill University, Montreal, May 1992.

[11] Allan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989. COMP TR89-93.

[12] Vivek Sarkar. The PTRAN parallel programming system. In B. Szymanski, editor, *Parallel Functional Programming Languages and Environments.* McGraw-Hill Series in Supercomputing and Parallel Processing, 1990.

[13] Vivek Sarkar and Guang R. Gao. Optimization of array accesses by collective loop transformations. *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 194–205, June 1991.

[14] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[15] Joe Warren. A hierarchical basis for reordering transformations. *Eleventh ACM Principles of Programming Languages Symposium*, pages 272–282, January 1984. Salt Lake City, UT.

[16] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 26–28 1991.

[17] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.