McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

Collective Loop Fusion for Array Contraction

Guang R. Gao
Russell Olsen
Vivek Sarkar†
Radhika Thekkath‡

ACAPS Technical Memo 41
March 1992

†IBM Palo Alto Scientific Center, 1530 Page Mill Road, Palo Alto, CA 94304
‡Dept. of Computer Science and Eng. FR-35, University of Washington, Seattle, WA 98195

Abstract

In this paper we propose a method for applying the *loop fusion* and *array contraction* optimizations across a collection of loop nests. Loop fusion and array contraction are techniques that can reduce array communication between fused loops by transforming array references into simple scalar references. In 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 [SG91]. This paper extends our previous work by taking into account the data dependence constraints imposed by loop fusion and investigating how the loop nests should be fused together so as to maximize the number of array accesses that are actually optimized by array contraction.

We propose an efficient method called *collective loop fusion* that partitions a collection of loop nests into fusible clusters. After an optimized partition is obtained, loop fusion and array contraction are performed on all loop nests within a cluster so as to reduce the overhead of array accesses and to expose more instruction-level parallelism. Our partitioning method is both novel and efficient; it is novel in that it uses the max-flow min-cut algorithm to minimize the cost of array operations, and it is efficient in that it executes in polynomial-time. This 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, and is generally better than clustering algorithms based upon a greedy approach.

In general, collective loop fusion and array contraction provide an important opportunity to exploit instruction-level parallelism on modern high-performance architectures, such as pipelined and superscalar machines. To illustrate the potential benefit of these transformations, we include preliminary results of a case study showing the improvement that is achieved when collective loop transformations are applied to a sample collection of loops. For this study, the codes for various transformed configurations were benchmarked both on a Sun SPARC Server 4/490 and on an IBM RISC System/6000, Model 550, workstation. The results of these measurements confirm that the two primary transformations, collective loop fusion and array contraction, when applied together, can achieve substantial speedup over the original unclustered code.

Contents

List of Figures

List of Tables

1 Introduction

Loop optimization plays a critical role in compiler optimization of scientific application programs. In such programs, loops are mainly used to read and write large arrays of values. Prior work in the area of loop transformations for cache locality has shown how such loop nests can be transformed so as to reduce the costs of array accesses in a hierarchical memory system, by reusing data in the cache as much as possible [IT88, GJG88, Por89, WL91, FST91]. However, even with an ideal cache hit ratio of nearly 100%, array accesses can incur a large overhead on high-performance processor architectures due to two reasons: First, it is necessary to issue load and store instructions to move the array values between the cache and the register set; these instructions typically comprise about 30% of the total instructions executed, thus wasting many processor cycles that could otherwise have been spent on useful arithmetic operations [HP90]. Second and more importantly, the 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. Even for a large straight-line instruction sequence, instruction parallelism can only be exploited within tiny blocks because of fragmentation caused by the serializations from the load/store instructions. This limitation on exploiting instruction parallelism is a serious problem for modern processor architectures that depend more and more on instruction parallelism to achieve high performance (e.g., pipeline, superscalar, and LIW architectures).

In this paper we use an optimization called *array contraction* to avoid the overheads associated with array accesses [SG91]. 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 [CCK90]. Our approach is based upon performing loop fusion on a collection of loops. *Collective loop fusion* increases instruction-level parallelism primarily by using array contraction to eliminate load/store instructions. A secondary benefit arises from the fact that the number of instructions in a loop body increases after loop fusion is performed. Optimal elimination of load/store instructions by loop fusion is considered to be an "extremely hard problem" in the literature [All83]. Our solution is partially based upon using the *Max-Flow Min-Cut* algorithm from Network-Flow theory [Tar83] to determine how the loop nests should be fused.

Apart from the general approach, our paper offers three main results:

1. a formulation of Collective Loop Fusion as a graph partitioning/clustering problem,

2. a solution to the partitioning/clustering problem based upon a special cluster numbering scheme and on one or more applications of the max-flow min-cut algorithm, (This solution 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 and is generally better than clustering algorithms based upon a greedy approach), and

3. experimental results to show the benefit of our approach of performing array contraction by collective loop fusion.

The rest of the paper is organized as follows: Section 2 describes the *loop dependence graph* which is the program representation assumed for collective loop fusion. Section 3 formally states the optimization problem. Section 4 describes in detail our solution to this problem, based upon the *max-flow, min-cut* network flow algorithm. Section 5 presents a case study and experimental results to validate our approach. Section 6 discusses related work, and finally, Section 7 contains our conclusions.

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 [FOW87], with conformable loop bounds. A node in the LDG represents one of the k *perfect loop nests* [Wol89]; an edge in the LDG represents a loop-independent data dependence from the source loop nest to the destination loop nest [All83]. Each edge in the LDG is marked as being *fusible* or *nonfusible*. The source and destination loop nests of a nonfusible LDG edge are not allowed to be fused because doing so would violate the data dependence test for loop fusion [Wol89]. Fusible edges are further classified as being either *contractable* or *noncontractable*. A contractable edge is one for which loop fusion will yield a savings in memory cost because the data computed by the statement from the source loop nest will be available in the processor's registers for use by the statement from the destination loop nest, and a noncontractable edge is one for which loop fusion will not yield such a savings in memory cost. An LDG's edges correspond to *flow*, *anti*, or *output* data dependences [Wol89]; however, only flow edges can possibly be marked as being contractable.

Figure 1 contains a simple example with six FORTRAN loop nests. The Loop Dependence Graph for Figure 1 is shown in Figure 2a in Section 4, where the nonfusible edges are marked with X and the fusible-noncontractable edge 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 orders; similarly, the nonfusible edge from v_4 to v_6 occurs due to array `E` being produced and consumed in reverse orders (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 LDG is a directed acyclic multigraph. Nodes $n_1, \ldots, n_k$ are numbered according to the execution ordering of the loop nests in the original program; this ordering must be according to a topological sort of the partial order defined by the LDG's dependence edges. Since the loop nests corresponding to $n_1, \ldots, n_k$ are assumed to be identically control dependent, there can be no control branching among them; however, each loop nest can contain arbitrary control

```
         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

flow within its body, and the region corresponding to $n_1, \ldots, n_k$ can also be embedded within an arbitrary control flow.

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 in a sequence that is according to a topological sort of the partial order defined by the edges in the reduced graph.

The assumption that all loop nests corresponding to $n_1, \ldots, n_k$ have conformable loop bounds was made to simplify presentation, though this assumption is satisfied by loop nests obtained from loop distribution or from the Fortran-90 `WHERE` statement (both important candidates for loop fusion). The results described in this paper are also applicable to any LDG in which all nodes are identically control dependent, but 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

The goal of collective loop fusion is to 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. The motivation for maximizing the number of fused contractable edges is as follows: Each contractable edge represents an opportunity for savings in memory costs, and the opportunity is realized when the source and destination loop nests of a contractable edge are fused.

The goal of maximizing the number of fused contractable edges implicitly assumes that the memory cost savings from each fused contractable edge is the same, say, equal to 1 unit. Here, 1 unit essentially denotes the cost of loading a data value from memory. For simplicity we ignore the cost of storing a data value because *1)* whether or not that cost can be eliminated depends upon whether or not the value is needed later during program execution and *2)* because a store-buffer often hides storing cost, in modern processor architectures. 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 [Sar90b]. For the sake of simplicity, we approximate this case by a unit cost as well. In future work we will extend the cost model to take into account conditional probabilities.

In previous work we showed how to perform loop reversal and loop interchange on individual loop nests so as to collectively reduce the number of contractable edges [SG91]. There has also been a lot of prior work on loop transformations for reducing hierarchical memory access costs of individual loop nests [IT88, GJG88, Por89, WL91, FST91]. In this paper we focus on the loop-fusion problem and assume that suitable loop configurations have already been selected for the individual loop nests, using methods such as those referenced above.

4 A Network-Flow Formulation of the Partitioning Problem

In this section we present our solution to the graph partitioning/clustering problem described in Section 3. The solution described is quite general and works on any acyclic graph having an unrestricted number of nonfusible edges and any number of contractable and noncontractable fusible edges. We suspect that the general problem of collective loop fusion is NP–Complete due to the objective of finding a cluster partition that minimizes the number of inter-cluster contractable edges; however, we do not have a satisfactory NP-Completeness proof, as yet.

Nonetheless, the solution presented in this section is provably optimal for the special case of two-cluster partitioning for which the *maxflow* algorithm is known to yield a cut set of minimum size, and it is generally better than clustering algorithms based upon a greedy approach.

Given a Loop Dependence Graph, as previously described, the key idea behind our algorithm is to *find the minimum number of clusters* that satisfy the two problem constraints: *1)* separating all nonfusible edges across multiple clusters and *2)* ensuring that the reduced graph is acyclic. This minimum-cluster partitioning is then used to satisfy the cost requirement of maximizing the number of contractable edges within a cluster.

A graph that has no nonfusible nodes or edges can be trivially fused into a single cluster; thus, the algorithm is only applied to acyclic graphs which have at least one nonfusible edge or node. Let $G = [V, E]$ be such a graph. The algorithm consists of two steps:

Step 1: Find the minimum number of clusters required to satisfy the problem constraints. This is done by assigning cluster numbers to each vertex $v \in V$. A vertex may be assigned to more than one cluster so a set called the partition-cluster set, $Part(v)$, is associated with each vertex v, and this set contains all of the possible cluster numbers for the given vertex. If the partition-cluster set of a vertex is a singleton set, it directly provides the output cluster number for the vertex. The subset of vertices that have singleton partition-cluster sets is denoted $V^o \subset V$. The vertices that do not belong to V^o are denoted by the subset $V^s = V - V^o$. Vertex $v \in V^s$ will be assigned to one of the clusters in $Part(v)$, and the final assignment is made in Step 2 of the algorithm. The partition-cluster sets are then used to construct an acyclic reduced graph $G' = [V', E']$ from graph G. The reduced graph is constructed by

1. collapsing some vertices of G into a single vertex in G' and
2. by deleting the nonfusible edges and other irrelevant edges from G such that $E' \subset E$.

Step 2: Assign a suitable cluster number to each $v \in V^s$. This cluster assignment is made from one of the elements in the partition-cluster set created for this vertex, in Step 1. This assignment is designed to maximize the number of contractable edges within a cluster. Implementation is based upon using repeated application of the maxflow algorithm to G', until the graph is fully partitioned.

Section 4.1 and 4.2 contain detailed descriptions of the two parts of the algorithm. Section 4.3 discusses a few special situations that do not appear in the general algorithm description, and Section 4.4 gives the time complexity of the algorithm along with a sketch of a correctness proof for the algorithm.

For the purpose of illustration we use the example in Figure 2. Figure 2a contains an input graph $G = [V, E]$ with six vertices, $v_1, \ldots, v_6$. The nonfusible edges are marked with an

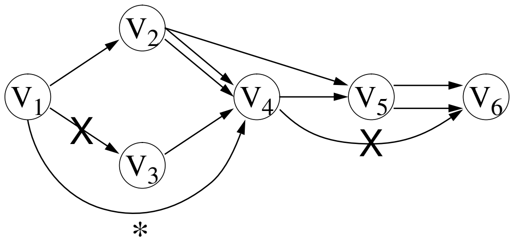

Figure 2a.

	Partition Cluster Sets					
Step	P_{v_1}	P_{v_2}	P_{v_3}	P_{v_4}	P_{v_5}	P_{v_6}
1	$\{1\}$	$\{1\}$	$\{2\}$	$\{2\}$	$\{2\}$	$\{3\}$
2	$\{1\}$	$\{1, 2\}$	$\{2\}$	$\{2\}$	$\{2, 3\}$	$\{3\}$

Figure 2b.

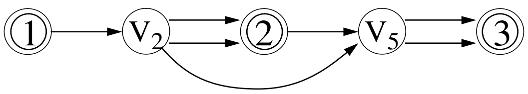

Figure 2c.

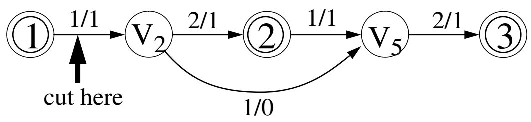

Figure 2d.

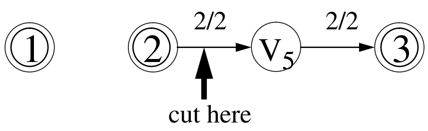

Figure 2e.

member(1) $= \{v_1\}$
member(2) $= \{v_2, v_3, v_4\}$
member(3) $= \{v_5, v_6\}$

Figure 2f.

Figure 2: An Example of Cluster Partitioning

X, and the fusible-noncontractable edge is marked with an $*$. All unmarked edges are assumed to be both fusible and contractable. This example illustrates as many features of the algorithm as possible.

4.1 Construction of the Reduced Graph

In this section we discuss how vertices are assigned cluster numbers and how this information is used to construct the reduced graph. Associated with each vertex v is a partition-cluster set $Part(v)$, where $Part(v) = \{x \mid v \text{ can belong to cluster } x\}$. We now define what it means for one partition-cluster set to be equal, less than, or greater than another partition-cluster set. These definitions are used in the algorithm of Figure 3.

Definition 4.1 $Part(u) = Part(v)$ when the sets are identical.

Definition 4.2 $Part(u) < Part(v)$ when $\min(Part(u)) < \min(Part(v))$. (This implies that there is at least one cluster number in $Part(u)$ that has a value less than all cluster numbers in $Part(v)$.)

Definition 4.3 $Part(u) > Part(v)$ when $\max(Part(u)) > \max(Part(v))$. (Intuitively there is at least one cluster number in $Part(u)$ that has a value greater than all cluster numbers in $Part(v)$.)

We outline below the steps leading up to the construction of the reduced graph (see Figure 3 for details):

1. Visit the vertices of G in topological order to find the lowest-numbered cluster to which they can belong. All source vertices of the graph are initially assigned a cluster number of 1. If the maximum value of a cluster number assigned to any vertex is k, then k is the minimum number of clusters required to satisfy the problem constraints. $\forall v \in V$, create a partition-cluster set $Part(v)$ with a single element.

2. Then traverse the vertices in reverse topological order to find all higher-numbered clusters to which each vertex can belong. Based on the number of elements in their partition–cluster set, the vertices of the graph can be divided into two disjoint subsets:

   (a) if $|Part(v)| = 1$, then $v \in V^o$, i.e., the subset V^o has vertices that can belong to only one specific cluster, and

   (b) if $|Part(v)| > 1$, then $v \in V^s$, i.e., the subset V^s has vertices that can belong to more than one cluster.

   Note that since G has at least one nonfusible edge, the set V^o cannot be empty. Figure 2b shows the result of applying these steps of the algorithm to the example graph of Figure 2a. Observe that three clusters were created, numbered 1, 2, and 3.

3. The key notions behind the construction of the reduced graph are *1)* collapsing all vertices that *must* belong to the same cluster, and *2)* eliminating all nonessential edges between vertices. This simplified graph can then be further partitioned using the maxflow algorithm.

   The new vertex set V' is given by $V' = V^s \cup V^c$. The set V^s, or the set of *simple vertices*, is known from the previous step. The vertices in the set V^c, or the *cluster vertices*, are constructed from V^o by collapsing multiple vertices from G. These multiple vertices, which form a single cluster vertex $v' \in V^c$, are said to belong to $\texttt{member}(v')$. The v' in this case simply represents the number of the appropriate cluster. (See Figure 3 for the details of this construction.)

   The set E' is constructed from E by

   (a) eliminating all edges between vertices that can *never* belong to the same cluster, i.e., if the intersection of the $Part()$ sets for these edges are empty, (This eliminates all nonfusible edges and some other edges that are not nonfusible. Such edges are already predetermined to be inter-cluster edges and therefore play no part in future partitioning decisions. Hence, they can be eliminated from the graph.)

   (b) eliminating all edges having $\texttt{cost} = 0$. Such edges do not determine the total cost of a fusion partition and, hence, must be removed from the decision-making phase of the maxflow algorithm.

   For the example in Figure 2a, three cluster vertices, 1, 2, and 3 are created (shown by the double-circled nodes in Figure 2c). The `member` sets can be found from the values in Figure 2b as follows: $\texttt{member}(1) = \{v_1\}$, $\texttt{member}(2) = \{v_3, v_4\}$, and $\texttt{member}(3) = \{v_6\}$. The simple vertices are v_2 and v_5, shown as single-circled nodes. Figure 2c shows the reduced graph for the example in Figure 2a.

4.2 Application of the `maxflow` Algorithm

In this section we describe how the reduced graph from Section 4.1 is further partitioned. For all simple vertices, i.e., $\forall v \in V^s$, we are faced with the problem of finding an element from its partition-cluster set $Part(v)$ which minimizes the number of inter-cluster edges. We will 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

- the flow value never exceeds the capacity of the edge, and
- 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.

Input: an acylic graph $G = [V, E]$

Output: an acyclic reduced graph $G' = [V', E']$ of G

Procedure:

1. Traverse the vertices of G in topological order; for each vertex v, do the following:
   (a) If v has no incoming edges, assign $Part(v) = 1$.
   (b) If v has incoming edges,

$$Part(v) = \mathtt{MAX}\left(\forall(u,v) \in E, \begin{cases} Part(u)+1, & \text{if } (u,v) \text{ is nonfusible} \\ Part(u), & \text{otherwise} \end{cases}\right)$$

2. Again traverse the nodes of the graph in reverse topological order, and for each vertex v, do the following:
   (a) If v has no outgoing edge, go to the next vertex.
   (b) For each outgoing edge $(v, w) \in E$,
      i. if $Part(w) = Part(v)$, go to the next outgoing edge, (Note that $Part(w) < Part(v)$ is not possible due to the inherent nature of the cluster-number assignment.) and
      ii. if $Part(w) > Part(v)$, then for each $x \in Part(w)$ such that $x >$ every $y \in Part(v)$, add x to $Part(v)$, **unless** either of the following conditions is true:
         A. there is a path from v to some vertex y and $x \in Part(y)$ such that this path goes through a vertex t, where $Part(u) \leq Part(t) < Part(y)$ (i.e., the path violates the acyclic constraint), or
         B. there is a path from v to some vertex y and $x \in Part(y)$, and this path has a nonfusible edge.

3. Construct the reduced graph G'. Define the vertex set V' and the edge set E' as follows:
   (a) For each vertex $v \in V$,
      i. if $v \in V^s$, then $v \in V'$,
      ii. if $v \in V^o$ and $Part(v) = \{x\}$, add a vertex x to V' and add v to $\mathtt{member}(x)$. If x has already been created, we only need to add v to $\mathtt{member}(x)$.
   (b) For each edge $(u, v) \in E$, if $Part(u) \cap Part(v) \neq \phi$ and (u, v) is contractable, then $(u, v) \in E'$.

Figure 3: Step 1 of the Partitioning Algorithm

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. 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*. (See Tarjan for details [Tar83].)

The reduced graph from Section 4.1 comprises of several disconnected components. A component that has only one cluster vertex can automatically assign that cluster number to all of its simple vertices, and therefore the component needs no further partitioning. All components that have more than one cluster vertex contain simple vertices that need to be disconnected from all but one cluster vertex. We do this by assigning a capacity of 1 to all edges and then finding a maximum flow through this graph. The minimum cut obtained from this 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 [Tar83]. We repeatedly apply maxflow until the components of the graph cannot be further partitioned. This happens when each component has exactly one cluster vertex in its vertex set.

Each application of the maxflow algorithm partitions the graph into two components that are each smaller than the original. This is true by definition of the minimum cut of a maxflow algorithm, but sometimes we can do better and partition the graph into more than two components. This is done by attempting to cut the graph at every edge where flow equals capacity and not just at the minimum cut edges. The two cases in which the graph cannot be cut at an edge where the flow equals the capacity are:

1. when a simple vertex is disconnected from all cluster vertices in the component, and
2. when this edge does not disconnect the graph.

The graph should be cut at all other edges where flow equals capacity.

Figures 2d , 2e, and 2f show the application of the maxflow algorithm to the reduced graph shown in Figure 2c. Figure 2d shows the capacity and the flow on each edge of the graph. The capacity of the edge is followed by the "/" sign and the flow on that edge, is obtained from the maxflow algorithm. Note how the two edges between the simple vertex v_2 and the cluster vertex 2 has been replaced by a single edge with capacity 2. A similar replacement was made between vertices 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 partition-cluster set (see Figure 2b). The edge between v_2 and v_5 now creates another edge between the cluster vertex 3 and vertex v_5, and this is replaced by a single edge of capacity 2. Figure 2e shows the application of maxflow to this new graph. This places v_5 into cluster 3, fully partitioning the graph.

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 vertex, add any simple vertices in this component to this cluster.
2. If the component has multiple cluster vertices, further partition the component by applying the maxflow algorithm:
   (a) Assign an edge capacity value of 1 to all edges in this component. If there are multiple edges between the same vertices, replace the 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. Also, do not disconnect a simple vertex from all cluster vertices.
   (c) Repeat the above steps for each connected component that is created.

Figure 4: Step 2 of the Partitioning Algorithm

4.3 Special Cases

The algorithm described in the previous sections does not handle certain special situations which might arise. The reason was to avoid cumbersome details, focusing instead on the essential working of the algorithm. A few special cases, however, are mentioned below, along with a description of the necessary modifications to the algorithm:

- The graph might contain nonfusible nodes, as described in Section 2. These nonfusible nodes contribute to the acyclic partitioning of the reduced graph and must always be assigned to a distinct cluster. Hence, they are used in the initial step of assigning cluster numbers but must not participate in the actual partitioning, during application of the maxflow algorithm. To implement this, we first delete the cluster number of nonfusible nodes from all simple vertices of the reduced graph, and we also delete all edges to and from these nodes in the reduced graph. This isolates the nonfusible nodes from the rest of the graph, and the maxflow algorithm can now be applied.

- The input graph G might be disconnected. In such situations the algorithm is applied once to each component of G. Once all connected components are partitioned, all equally-numbered clusters from each component are merged to obtain the complete partition of the graph. This must be done carefully to avoid merging a cluster formed from a nonfusible node.

- The graph G', or one of its components created by an application of the maxflow algorithm, might contain more than one source node or more than one sink node. Such a graph cannot be used directly on the maxflow algorithm. (A source node is one that has no incoming edges, and a sink node is one that has no outgoing edges). In such situations 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 value, far greater than the maximum flow possible through the graph. For example, one might use a capacity that is greater than the sum of the capacities of all of the edges. This ensures that a pseudo edge will never be a minimum cut edge, and hence, that no partitioning problems will occur, that is, if the set S trivially contains only the pseudo source or if the set T trivially contains only the pseudo sink. The resulting graph is then used as input to the maxflow algorithm.

4.4 Algorithm Correctness and Complexity

In this section we discuss the correctness and complexity of the above algorithm.

Algorithm Correctness: We must prove that our algorithm produces a legal fusion partition, where a fusion partition is *legal* iff

1. for each non-fusible edge, the source and destination nodes belong to distinct fusion clusters, and

2. the reduced graph defined by the fusion partition is acyclic.

Proof (sketch): We will prove that the cluster numbering done by Step 1 of our algorithm ensures a legal fusion partitioning.

To prove condition (1) we will show that for any nonfusible edge (u, v), the partition-cluster sets assigned to nodes u and v by Step 1 of the algorithm will be disjoint, i.e., that $Part(u) \cap Part(v) = \phi$. This is sufficient to prove condition (1) since u and v can subsequently be assigned a cluster number only from their respective partition-cluster sets.

For each vertex $v \in V$, a single cluster number is added to $Part(v)$ during the forward pass, and multiple cluster numbers can be added during the reverse pass. We claim that after the forward pass, $Part(u) \cup Part(v) = \phi$, for any nonfusible edge (u, v). This is true by the cluster number assignment made in the forward pass, where v will always have a cluster number that exceeds u's cluster number, by at least one (see Step 1, Figure 3). We must prove that during the reverse pass, u can never be assigned a cluster number from $Part(v)$. Note that the reverse path is done in reverse topological order so that $Part(v)$ is complete before $Part(u)$ is modified. Assume that $x \in Part(v)$. We claim that $x \notin Part(u)$, because otherwise, it would violate constraint B, specified by Step 2 of the algorithm in Figure 3. This condition states that for a vertex u to be assigned a cluster number x, there must not exist any node y with this cluster number x, such that a path from u to y (if it exists) has a nonfusible edge. In our case the direct edge between u and v is nonfusible; therefore, u can **never** be assigned any cluster number assigned to v. This concludes our proof that condition (1) is satisfied by our algorithm.

To prove condition (2), we show that the clustering of nodes induced by our algorithm does not violate the partial ordering induced by the directed edges of the graph, and since the original graph was acyclic, the reduced graph must also be acyclic.

Let the original graph have two fusible edges, (v_1, v_2) and (v_2, v_3), and no nonfusible edges between these three vertices. If vertices v_1 and v_3 belong to a cluster x, then v_2 must also belong to cluster x. This is true because of constraint A in Step 2 of the algorithm given in Figure 3. This constraint ensures that every pair of vertices v_i and v_j can belong to some cluster x iff every vertex v_k on every path between v_i and v_j also belongs to the same cluster x. This proves condition (2); thus our algorithm produces a legal fusion partitioning of the graph.

Algorithm Complexity: We assume that the graph has n vertices and m edges. Step 1 of the algorithm is itself composed of three steps: The first step is the forward pass of the vertices that examines all incoming edges. In doing so, each node is visited once and all of the edges are visited once, so the total time taken for this step is $O(n + m)$. The second step is the reverse pass that visits each node and traverses paths leading out of this node. In the worst case, it could traverse all paths leading out of this node, and hence, all the edges of the graph could be traversed once for each node; therefore, the total time taken is $O(nm)$. In the third step the reduced graph is built. This requires a visit to all the vertices to build the new vertex set and a visit to all the edges to build the new edge set; therefore, the time taken is $O(n + m)$. The total time taken for Step1 is $O(n + m) + O(nm) + O(n + m) = O(nm)$.

To find the complexity of Step 2, assume that, in the worst case, each application of the maxflow algorithm isolates only one vertex, therefore requiring $n - 1$ applications of maxflow. If we assume that the maxflow algorithm on a graph takes time $O(nm \log n)$ [Tar83], the total time taken for Step2 is $O(n^2 m \log n)$. Thus the total time for the entire algorithm is dominated by the second part of algorithm, $O(nm) + O(n^2 m \log n) = O(n^2 m \log n)$.

5 A Case Study and Experimental Results

In this section we illustrate the impact of partitioning strategy upon the performance of a collection of loops. Specifically, we compare the performance of code for a sample collection of loops to the performance of various versions of transformed code. The specific transformations tested were collective loop fusion and array contraction. The loop cluster used during our tests contains six loops, each loop consisting of a comparatively simple loop body. The loop collection itself in shown in Figure 5, along with various transformed codes, and the corresponding loop dependence graph, or LDG, for the original cluster is shown in Figure 6.

Notice that the original collection has two nonfusible data dependences, from loop 1 to loop 3 and from loop 4 to loop 6, which can be detected by the compiler [SG91, Ols92]. At this point the compiler has several alternatives: *a)* forgo collective transformation altogether, *b)* perform array contraction on the original cluster without performing any fusion (the notion here will soon become clear), *c)* naively partition the LDG, possibly cutting more edges than necessary, and perform the corresponding loop fusion, *d)* naively partition the loops and perform array contraction within each partition, or alternatively, *e)* efficiently partition the graph and stop there, or *f)* efficiently partition the graph and perform array contraction within the loop bodies of the respective partitions. All of these alternatives are depicted in the various source codes shown in Figure 5. 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 5b) where scalar references are used for array elements to avoid redundant array accesses.[1]

5.1 Testing Objectives and Methodology

The test codes shown in Figure 5 were designed with several different evaluation objectives in mind: *1)* to evaluate our partitioning algorithm against a more naive approach, *2)* to isolate the effects of loop fusion and array contraction when applied at various stages of transformation, *3)* to determine the effect of standard compiler optimization upon these source-level transformations, and *4)* to compare the effectiveness of loop transformations on computer architectures which support different degrees of instruction-level parallelism. Specifically, we wanted to measure the effect of collective transformation when run on both a scalar and a superscalar computer. The particular machines used during our tests were a Sun SPARC Server 4/490 and an IBM RISC System/6000, Model 550, Type 7013.

The results of partitioning based upon timings taken on the SPARC server are shown in Table 1a. 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

[1] The compilers we used failed to recognize such references as scalar identities and therefore unnecessarily reload the previously written array element value again from memory, rather than using the value which is still in register, see Callahan et al. [CCK90].

```
      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**

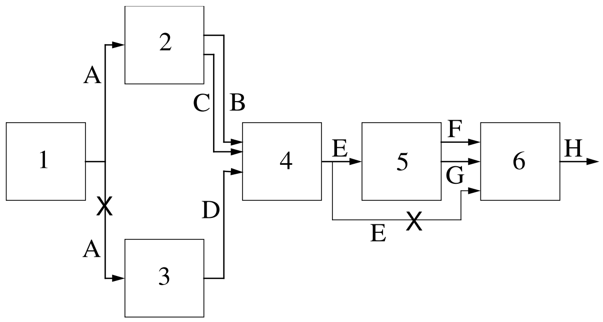

Figure 6: Loop Dependence Graph for a Sample Loop Collection

timed on an IBM RISC System/6000 (RS/6000) and the results are shown in Table 1b. Because the test code segments were short, they were each executed fifty times within a repetition loop to obtain improved timing-function resolution. For the f77 compiler on the SPARC we used the `etime()` function (with resolution of one-tenth of a second), and for the xlf compiler on the RS/6000, we used `mtime()` (with resolution of one-hundreth of a second). The statistics reported in Table 1 use the metric *eps*, or output array elements generated per second. This number represents the rate at which the collective calculation is able to proceed, or in this specific instance, 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 anomolies which otherwise might affect the result were the measurements based solely upon a single vector size. To assist in interpreting these results, we show in Table 2 the corresponding relative improvement due to the various loop transformations. Because we were looking for the dominant effects caused by the various loop transformations, our evaluation objectives were achieveable using just the single example described above. In this respect, we make no assertion regarding whether our sample code could be part of any particular computation or that it is representative of codes typically found in practice.

5.2 Evaluation of Transformation Effectiveness

In general, little improvement occurred as a result of applying array contraction to the original loop (refer to Figure 5b). 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; therefore the benefit was minimal. In the case of the RS/6000, it actually resulted in lower performance

a. Loop Transformation Performance on a Sun SPARC Server 4/490†

Phase of Transformation	integer f77	integer f77 -O	single precision f77	single precision f77 -O	double precision f77	double precision f77 -O
original loop collection	111	171	127	215	87	141
array-contracted original	117	180	123	222	86	143
fused naive partition	125	186	145	229	95	141
array-contracted naive part.	138	204	140	261	94	164
fused efficient partition	118	178	143	210	103	145
array-contracted efficient part.	147	228	154	328	114	239

b. Loop Transformation Performance on an IBM RISC System/6000†

Phase of Transformation	integer xlf	integer xlf -O	single precision xlf	single precision xlf -O	double precision xlf	double precision xlf -O
original loop collection	222	936	231	1099	220	963
array-contracted original	214	900	224	1088	217	943
fused naive partition	276	1071	280	985	265	1024
array-contracted naive part.	279	1103	271	1056	280	1012
fused efficient partition	274	1135	283	1152	279	997
array-contracted efficient part.	342	1390	354	1230	349	1859

† 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**

(cf. top row of Table 2). The transformation does, however, provide a basis for comparing the other loop-transformation alternatives since it allows us to isolate the effects due to partitioning.

When the loop cluster was naively partitioned to allow loop fusion, as shown in Figure 5c, the benefit was again small, if at all, because all that was eliminated in this case was some loop-control overhead and this too was small when compared to the overall computation (cf. second row of Table 2). Likewise, the efficiently partitioned loops (Figure 5e) also showed little improvement for the same reason (cf. forth row of Table 2). In both cases, the effect of transformation was merely to reduce the number of loops from six to three. Another potential benefit from fusion would be to increase the size of the corresponding loop bodies, commensurately increasing the pool of instructions available for instruction scheduling. Unfortunately, compared with even the small loop bodies of the original loops, the larger fused loops made little difference. Regardless, fusion serves a more important purpose in the collective-transformation process since it facilitates further transformation by array contraction.

The improvements obtained by performing both loop fusion and array contraction were

Table 2: **Relative Improvement due to Loop Transformation**

Phase of Transformation	Sun 4/490			IBM RS/6000		
	integer	sgl. prec.	dbl. prec.	integer	sgl. prec.	dbl. prec.
array-contracted original	1.05	1.03	1.01	0.96	0.99	0.98
fused naive partition	1.08	1.03	1.00	1.14	0.90	1.06
array-contracted naive part.	1.19	1.21	1.16	1.18	0.96	1.05
fused efficient partition	1.04	0.97	1.03	1.21	1.05	1.04
array-contracted efficient part.	1.33	1.53	1.70	1.49	1.12	1.93

† Entries represent speedup over that of the original loop collection run on the respective machine.

far greater than that of the other cases discussed so far (see Figures 5d and 5f, and Table 2). Part of the reason was, of course, the incremental improvement provided by the prior loop fusion; but the main reason for improvement was the elimination of an array. Notice that in the second loop nest of Figure 5d, the `D` array has been eliminated entirely and replaced by the scalar variable `d`. In the last case we tested, an array-contracted efficient partitioning (Figure 5f), the test statistics showed much greater improvement—up to almost a two-fold performance improvement on the RS/6000 for double-precision floating point arrays (see the last row, Table 2). The reason for the improvement in this case 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 minimum nonfusible edges in the LDG (cf. Figure 6). This demonstrates the superiority of our efficient partitioning algorithm to the naive approach.

It is worth noting that the preceding observations applied uniformly regardless of whether a scalar or superscalar architecture was employed. Specifically, where differences in performance were observed across the two architectures, the particular architectural difference applied across the entire range of transformations, with none of the transformations appearing to be any more or less susceptible to any architectural effect than any other transformation. Even so, major architectural differences were observed. The first relates to the extent of improvement derivable from compiler optimization and the second relates to the relative performance of single- and double-precision floating-point calculation. For the SPARC, compiler optimization resulted in up to a seventy percent improvement in performance over nonoptimized code, with the most improvement coming as a result of array contraction; whereas on the RS/6000, a four-fold improvement was observed, with the largest improvement coming with double-precision codes. None of these effects, however, are new.

From these tests we are able to conclude that the benefit from collective transformation can be substantial. The benefit, in this case, derives not from loop fusion, per se, but from a combination of array contraction and loop fusion, suggesting that the number of across-partition arcs remaining after partitioning is more important than the number of partitions created.

6 Related Work

Although there has been much progress made in optimizing the performance of loops, most of this work has focused on individual loops rather than on collections of loops [KRP+81, AK87, Wol89]. Only recently has a combination of loop optimization techniques been applied together and only in the case of a single loop nest; specifically, we refer to unimodular loop transformations [Ban90, WL90].

Loop distribution and loop fusion are two well known loop transformations which deal with multiple loop nests [Wol89]. Typically, loop distribution is performed maximally (Π-block partitioning) and loop fusion is used as follow-up to selectively merge the distributed loops back together. In general, loop fusion is useful for reducing loop overhead for serial machines, increasing the size of loop bodies, which affects instruction-scheduling effectiveness, and reducing memory requirements, in turn, reducing memory-register traffic. This last benefit is especially important for vector machines for which vector register loads and stores are costly. In spite of these benefits, the ability to effectively perform loop fusion remained a difficult problem in past work [All83].

Earlier we noted that the term "array contraction" was used by Wolfe for a different optimization: reducing the size of a compiler-generated temporary array that was created by scalar expansion [Wol89]. Allen also discussed the problem of minimizing temporary array storage in a single loop nest, by a technique called *sectioning* [All83]. However, optimization beyond a single loop nest was not considered in these cases. Register allocation of subscripted variables (arrays) has been recognized as a challenging optimization problem and studied by Callahan et al., but as with the other related research, their work too focused on single loop nests [CCK90].

The opportunity for optimizing array operations across a collection of loops for *dataflow software pipelining* was recognized in Gao [Gao86, Gao90], and loop parallelization and loop chunking across collections of loops has been described by Sarkar [Sar89, SC90, Sar90a]. In our work we consider collective loop optimization both at the individual loop-nest level (e.g., loop reversal and loop interchange) and at a global level, where loops are partitioned into clusters for fusion. Our method can be used both in exploiting instruction-level parallelism on uniprocessors and on multiprocessors.

In 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 [SG91]. This paper extends our previous work by taking into account the data dependence constraints imposed by loop fusion and investigating how the loop nests should be fused together so as to maximize the number of array accesses that are actually optimized by array contraction.

7 Conclusions and Future Work

In this paper we presented a method for performing collective loop fusion and array contraction. We also presented an efficient algorithm for partitioning programs into fusible loop clusters to enable array contraction. We then demonstrated the effectiveness of this method for exploiting instruction-level parallelism. From these results it is clear that the method can be useful in the optimization of scientific programs.

As future work we plan to investigate a combination of our partitioning technique with advanced loop scheduling techniques, such as loop unrolling and software pipelining, techniques to further exploit instruction-level parallelism within fused loops. We also plan to apply the analysis methodology employed in the case study to experiments on large benchmark programs, to further test and improve the method. 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), and Radhika Thekkath was supported in part by NSF PYI Award #MIP-9058-439.

References

[AK87] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9, 1987.

[All83] John R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation.* PhD thesis, Rice University, 1983.

[Ban90] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. To be published by Pitman in *Monographs in Parallel and Distributed Computing.*

[CCK90] 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.

[FOW87] 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.

[FST91] 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.

[Gao86] G. R. Gao. A maximally pipelined tridiagonal linear equation solver. *Journal of Parallel and Distributed Computing*, 3(2):215–235, June 1986.

[Gao90] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Boston, December 1990.

[GJG88] Kyle Gallivan, William Jalby, and Dennis Gannon. On the problem of optimizing data transfers for complex memory systems. Technical report, University of Illinois–Center for Supercomputing Research and Development, July 1988. Also in *Proceedings of ACM 1988 International Conference on Supercomputing*, St. Malo, France, July 4–8, 1988, pp 238–253.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 319–329, 1988.

[KRP+81] 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.

[Ols92] Russell Olsen. Analysis and transformation of loop clusters. Master's thesis, McGill University, Montreal, 1992. In preparation.

[Por89] Allan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989. COMP TR89-93.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London and The MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. This monograph is a revised version of the Author's Ph.D. dissertation published as Technical Report CSL-TR-87-328, Stanford University, April 1987.

[Sar90a] Vivek Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. Technical report, IBM Research, October 1990. Submitted to IBM Journal of Research and Development.

[Sar90b] 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.

[SC90] Vivek Sarkar and David Cann. POSC—a partitioning and optimizing SISAL compiler. *Proceedings of the ACM 1990 International Conference on Supercomputing*, pages 148–163, June 1990. Amsterdam, the Netherlands.

[SG91] 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.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[WL90] Michael E. Wolf and Monica S. Lam. Maximizing parallelism via loop parallelism. *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, August 1990.

[WL91] 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.

[Wol89] 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.