Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution[*]

Ken Kennedy[1] and Kathryn S. McKinley[2]

[1] Rice University, Houston TX 77251-1892
[2] University of Massachusetts, Amherst MA 01003-4610

Abstract. Loop fusion is a program transformation that merges multiple loops into one. It is effective for reducing the synchronization overhead of parallel loops and for improving data locality. This paper presents three results for fusion: (1) a new algorithm for fusing a collection of parallel and sequential loops, minimizing parallel loop synchronization while maximizing parallelism; (2) a proof that performing fusion to maximize data locality is NP-hard; and (3) two polynomial-time algorithms for improving data locality. These techniques also apply to loop distribution, which is shown to be essentially equivalent to loop fusion. Our approach is general enough to support other fusion heuristics. Preliminary experimental results validate our approach for improving performance by exploiting data locality and increasing the granularity of parallelism.

1 Introduction

Loop fusion transforms multiple distinct loops into a single loop. It increases the granule size of parallel loops and exposes opportunities to reuse variables from local storage. Its dual, loop distribution, separates independent statements in a loop nest into multiple loops with the same headers.

```
PARALLEL DO I = 1, N
     A(I) = 0.0                              PARALLEL DO I = 1, N
END                     ⟹                         A(I) = 0.0
                      fusion                      B(I) = A(I)
PARALLEL DO I = 1, N                         END
     B(I) = A(I)        ⟸
END                  distribution
```

In the example above, the fused version on the right experiences half the loop overhead and synchronization cost as the original version on the left. If all A(1:N) references do not fit in cache at once, the fused version at least provides reuse in cache. Because the accesses to A(I) now occur on the same loop iteration rather than N iterations apart, they could also be reused in a register. For sequential execution of this example, performance improvements of up to 34% were measured on an RS/6000 Model 540.

[*] This research was supported by the Center for Research on Parallel Computation, a NSF Science and Technology Center. Use of the Sequent Symmetry S81 was provided under NSF Cooperative Agreement No. CDA-8619393.

This paper examines two fusion problems. The first is to fuse a collection of parallel and sequential loops, minimizing synchronization between parallel loops without reducing the amount of parallelism. We refer to this criterion as *maximizing parallelism.* The fusion problem arises when generating efficient code for parallel architectures. We present an optimal algorithm for this problem based on a greedy algorithm.

The second problem is to maximize data locality by minimizing the number of data dependences between fused nests. We show this problem is NP-hard and present two heuristics for solving it. The first is also based on the greedy algorithm and is linear in time and space. It produces solutions of unknown precision when compared to an optimal solution. The second solution is based on the *maximum-flow/minimum-cut* algorithm and is therefore not as quick, but allows us to prove a tight worse-case bound on the precision of its solution.

All of the algorithms are flexible in that they will completely reorder loop nests to improve fusion. The two algorithms for improving reuse may also be used independently or integrated into the parallelization algorithm. Preliminary experimental results demonstrate the efficacy of our approach.

We begin with a brief technical background. We then review the technical criteria for *safe* fusion, for fusions which do not reduce parallelism, and for modeling fusion as a graph problem. The same review of loop distribution follows as well as the map of a distribution problem to one of fusion. Section 4 formally describes the graph framework for fusion we use in the remaining sections. Section 5 presents an optimal greedy algorithm which fuses to maximize parallelism. The next three sections explore fusion to improve reuse of array and scalar references, adding an explicit representation of reuse, proving it NP-hard, and presenting the heuristics. Our experimental results, a careful comparison to related work, and conclusions complete the paper.

2 Technical Background

2.1 Dependence

We assume the reader is familiar with *data dependence* [5] and the terms *true, anti, output* and *input* dependence, as well as the distinction between *loop-independent* and *loop-carried* dependences [4]. *Parallel loops* have no loop-carried dependences and *sequential loops* have at least one.

Intuitively, a *control dependence*, $S_1 \delta_c S_2$, indicates that the execution of S_1 directly determines whether S_2 will be executed [10, 12]. In addition to using control dependence to describe program dependence, we will use the control dependence and postdominance relations for an arbitrary graph G. These definitions are taken from the literature [12].

Definition 1 *y postdominates x in G if every path from x to the exit node contains y.*

Definition 2 *Given two statements x, $y \in G$, y is control dependent on x iff (1) $\exists$ a non-null path p, $x \rightarrow y$, such that y postdominates every node between x and y on p, and (2) y does not postdominate x.*

3 Transformations

Loop fusion and distribution are *loop-reordering transformations*; they change the order in which loop iterations are executed. A *safe* reordering transformation preserves any true, anti and output dependences in the original program. (Input dependences need not be preserved for correctness.)

3.1 Loop Fusion

Safety. Consider safe loop fusion between two loops or loop nests with *compatible* loop headers. Compatible loop headers have exactly the same number of iterations, but not necessarily the same loop index expressions. It is possible to make loop headers with differing numbers of iterations compatible by using conditionals, but the additional complexity may make reuse harder to exploit in registers.

Between two candidate nests the following dependences may occur: (1) no dependence, (2) a loop-independent dependence, and (3) a dependence carried by an outer loop which encloses the candidates. Clearly, fusion is always safe for case (1). Fusion is safe in case (3) as well; any loop-carried dependence between two loops *must* be on an outer loop which encloses them and fusing them does not change the carrier. The dependence will therefore always be preserved.

In the case of a loop-independent dependence, fusion is safe if the sense of the dependence is preserved, *i.e.*, if the dependence direction is not reversed. A simple test for this case performs dependence testing on the loop bodies as if they were in a single loop. After fusion, a loop-independent dependence between the original nests can (a) remain loop-independent, (b) become forward loop-carried or (c) become backward loop-carried. Since the direction of the dependence is preserved in the first two cases, fusion is legal. Fusion is illegal when a loop-independent dependence becomes a backward carried dependence after fusion. These dependences are called *fusion-preventing* dependences [1, 25].

Since a loop is parallel if it contains no loop-carried dependences and is sequential otherwise, fusion in case (b) is safe but prevents parallelization of the resultant loop. If either one or both of the loops were parallel, fusion would reduce loop parallelism. Therefore, *maximizing loop parallelism* requires these dependences to be classified as fusion-preventing.

Model. We represent the fusion problem with a graph in which each candidate loop nest is represented by a node and each dependence from a statement in one loop to a statement in another is represented as a directed edge between the nodes. If there are fusion-preventing dependences or incompatible loop headers between two nodes, the edge between them is marked fusion-preventing. If other program fragments are interspersed between the loops and these statements are not connected by dependences to any loop, these statements can be ignored during fusion. If there are dependences, the formulation captures the program fragments by placing them in a node(s) and using fusion-preventing edges. For example, a group of statements between two loops is represented as a node in the fusion graph, and the dependence edges between the statements and a loop are marked fusion-preventing. For loops on which the statements do not depend, fusion-preventing edges are added and if possible are oriented such that the node containing the statements may move either before or after pairs of fusable loops.

Given the definition of dependence on the original loop ordering, the fusion graph is a DAG. A safe fusion partitions nodes in the graph such that:

1. a partition contains a set of loops that can be legally fused, and
2. the ordering of nodes in the partitioned graph respects all the dependences.

Achieving *maximum granularity* of loop parallelism is thus equivalent to partitioning this graph into the fewest number of partitions.

3.2 Loop Distribution

Loop distribution is the dual of loop fusion. Rather than merge loops together, it separates independent statements inside a single loop (or loop nest) into multiple loops (or loop nests) with identical headers. Loop distribution exposes partial parallelism by separating statements which may be parallelized from those that must be executed sequentially and is a cornerstone of vectorization and parallelization. Loop distribution preserves dependences if all statements involved in a recurrence (*i.e.*, dependence cycle) in the original loop are placed in the same loop [18]. Loops containing recurrences must be executed sequentially.

Loop distribution first places each set of statements involved in a recurrence in a separate sequential node. Statements without recurrences are parallel and are placed in nodes by themselves. Dependences are edges between nodes and some may be fusion-preventing. Fusion-preventing edges connect pairs of nodes that must be sequential if fused. Since the statements execute correctly, but sequentially in the same loop, the edges are not required for correctness. This graph is a DAG and therefore can always be ordered using topological sort. The construction partitions the graph to its finest possible granularity. Maximizing the granularity of loop parallelism however requires a partition of the graph with the fewest number of parallel loops, an application of fusion to the finest partition graph. Distribution may therefore be viewed as fusion with a preceding step that divides the original nest into the finest partitions. Consequently, we only discuss fusion since our results apply to distribution as well.

4 The Partition Problem

The following description formally maps fusion to a graph partitioning problem.

Partitioning problem: Given a DAG where
nodes = loops and are marked parallel or sequential
edges = dependence edges, some of which are fusion-preventing

Rules:
1. *Separation constraint:* cannot fuse sequential and parallel nodes.
2. *Fusion-preventing constraint:* two nodes connected by a fusion-preventing edge cannot fuse.
3. *Ordering constraint:* the relative ordering of two nodes connected by an edge cannot change.

Goal: group parallel loops together such that parallelism is maximized while minimizing the number of partitions.

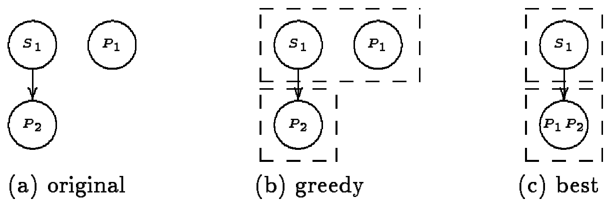


Fig. 1. Partition Example, s = sequential, p = parallel

The separation constraint contributes to *maximizing parallelism* by requiring that a particular partition contains only sequential or parallel loops. The ordering constraint is required for correctness. As discussed in Section 3, some fusion-preventing edges are necessary for correctness and others are needed to achieve an optimal solution.

Allen, Callahan and Kennedy refer to *maximal parallelism with minimum barrier synchronization* [3, 6]. By omitting the separation constraint, they arrive at an optimal greedy algorithm. Their work also tries to partition the graph into minimal sets, but uses a model of parallelism with both loop-level and fork-join task parallelism. These criteria are not equivalent to minimizing synchronization overhead when considering loop-level parallelism. Consider Fig. 1 where the edge (S_1, P_2) is an ordering constraint, S_1 is a sequential node, and P_1 and P_2 are parallel nodes.

The greedy algorithm places as many nodes as possible into a single partition without sequentializing a parallel node, but not all nodes in a partition are fused. Only nodes of the same type in the same partition are fused. For the program in Fig. 1(a), the greedy algorithm forms two partitions, $\{P_1, S_1\}$ and $\{P_2\}$, as illustrated in Fig. 1(b). S_1 and P_1 are not fused, but by using a parallel task construct, they may execute concurrently with each other. The iterations of the parallel loop P_1 also execute concurrently. Once they both complete, the parallel loop P_2 executes concurrently.

Because it ignores the node type, the greedy algorithm is provably optimal, minimizing the total number of partitions and maximizing task parallelism [3, 6]. The greedy algorithm however fails for the partition problem defined above, *i.e.* restricting it to loop-level parallelism, no task parallelism. For example, the parallel loop overhead and synchronization in the partitioning $\{S_1\}$ $\{P_1, P_2\}$ in Fig. 1(c), is half that of the partitioning in 1(b), although the amount of parallelism is equivalent. The greedy algorithm cannot achieve maximum loop-level parallelism because it unable to determine which node to select first. For example, if P_1 is selected first in Fig. 1(a), then three partitions result.

Our model considers the overhead of parallel loop startup and synchronization. The parallelization algorithm presented below maximizes the granularity and amount of loop parallelism by minimizing the number of parallel loops without sacrificing any parallelism. If task and loop parallelism are both advantageous, first using our loop parallelism algorithm and then Allen, Callahan and Kennedy may glean the benefits of both.

5 An Unweighted Fusion Algorithm

The following algorithm for maximizing the granularity of parallel loops is based on the observation that the problem may be divided based on node type because parallel and sequential nodes cannot be placed into the same partition. It consists of the following steps.

- Create a parallel component graph G_p from the original fusion graph G_o by placing all parallel nodes and edges between parallel nodes in G_o into G_p.
- Add fusion-preventing edges to G_p that preserve constraints in G_o not present in G_p.
- Partition G_p using the greedy algorithm into $G_{p'}$ and perform the partition specified by $G_{p'}$ on G_o, collapsing the original edges and forming $G_{o'}$.
- Create a sequential component graph G_s similarly from $G_{o'}$, but using sequential nodes and edges and then adding fusion-preventing edges to G_s that preserve constraints in $G_{o'}$ not present in G_s.
- Partition G_s into $G_{s'}$ with the greedy algorithm. Perform the partition specified by $G_{s'}$ on $G_{o'}$, forming the solution DAG.

This algorithm takes advantage of the constraint that sequential and parallel nodes cannot be fused by first separating the problem into a a parallel graph G_p, partitioning it, tanslating it onto G_o, and then creating the sequential graph G_s, partitioning it, and translating it. As we show in Section 5.2, partitioning the parallel graph first is required to minimize the number of parallel loops. However, because the process is the same for both of the *component graphs* G_p and G_s, we discuss them below simultaneously.

A component graph G_c represents an independent fusion problem with nodes of type c, in this case parallel or sequential. It consists of nodes of type c and the edges connecting them from G_o. These edges are not sufficient however because G_o may represent relationships that prevent nodes of the same type from being in the same partition, but that do not have an edge between them.

Example. Consider P_4 and P_7 in G_o in Fig. 2(a) where fusion-preventing dependences are marked with a slash. Although the edge (P_4, P_7) is not fusion-preventing, P_4 and P_7 may not legally fuse together without including S_6. Placing a parallel node and sequential node in the same partition however forces the partition to be executed sequentially, violating the maximal parallelism constraint. We prevent it by adding fusion-preventing dependences when we create G_p.

Transitive Fusion-Preventing Edges. The simplest way to find and preserve all the fusion-preventing relationships in G_o for G_c is to compute a modified transitive closure G_{tr} on G_o before pulling out G_c. In general, a fusion-preventing edge is needed between any two nodes of type c when a path between them contains a node not of type c. For example, if there is a path between any two parallel nodes that contains at least one sequential node, then they cannot be placed in the same partition and a fusion-preventing edge is added between them in G_p. Adding all of these edges unfortunately introduces redundant constraints.

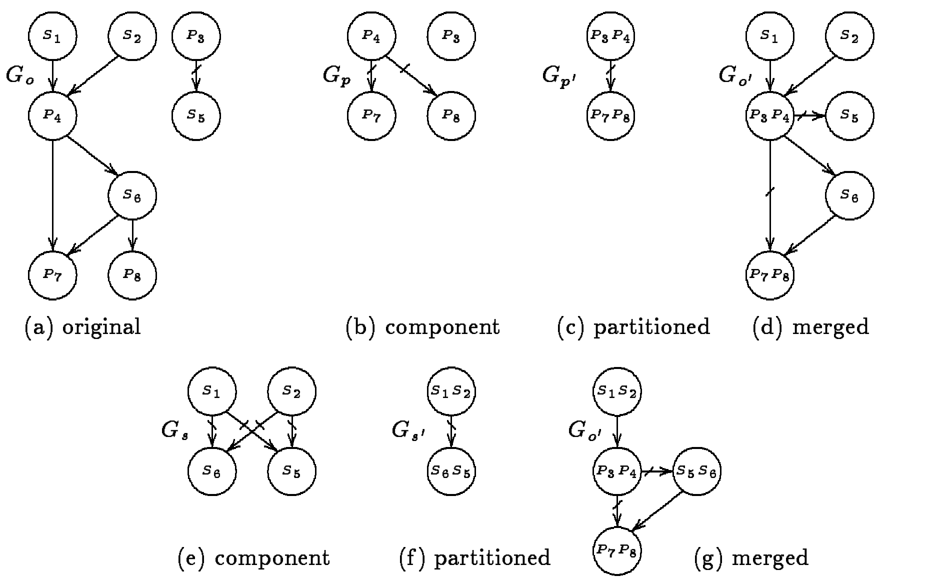


Fig. 2. Partitioning for Parallelism

Example. Assume for the moment that S_1 in Fig. 2(a) is parallel and call it P_1. Applying a full transitive closure algorithm for parallel nodes to this modified version of G_o would results in the following fusion-preventing edges: (P_1, P_7), (P_1, P_8), (P_4, P_7), (P_4, P_8). However, the edges (P_1, P_7) and (P_1, P_8) are redundant because of the original ordering edge (P_1, P_4) and the two other fusion-preventing edges (P_4, P_7) and (P_4, P_8) which together prevent P_1 from being fused with P_7 or P_8.

To simplify the discussion here, we present the algorithm *NecessaryEdges* in Section 5.1. It introduces the necessary and sufficient fusion-preventing edges to complete a component graph.

Partitioning a Component Graph. Given G_c is at most the transitive closure of a DAG, it must also be a DAG and can be partitioned optimally using the greedy algorithm into sets of loops that can be fused (see Callahan's dissertation for the proof [6]). The greedy algorithm places the roots of all the connected components into a partition and then recursively tries to add the successors in breadth-first order. A node can be added to the partition of a predecessor if it is not connected to any node in the partition by a fusion preventing edge. If a node cannot be added to an existing partition, a new partition is created for it. We call the partitioned component graph $G_{c'}$. Each node $N_{c'} \in G_{c'}$ contains one or more of N_c.

Translating a partition onto $\mathbf{G}_o$. To translate the partition $N_{c'}$ onto G_o and form $G_{o'}$, we simply combine the nodes in G_o corresponding to $G_{c'}$ in the same way. The edges in G_o are inherited in the obvious way; if $(n, m) \in G_o$ then a corresponding edge between the possibly singleton partition nodes containing n

```
NecessaryEdges ($G_o$, $G_c$)
   INPUT:      $G_o$ the original graph
               $G_c$ the component graph
   OUTPUT:     $G_c$ component graph with necessary transitive fusion-preventing edges
   ALGORITHM:
       forall $n$ in $G_o$ in preorder
(1)        if $type(n) \neq c$
               $Paths(n) = \bigcup_{(m,n) \in G_o} Paths(m)$
           else
(2)            $Paths(n) = n$
(3)            forall $(m, n) \in G_o$ s.t. $type(m) \neq c$
                   forall $r_i \in Paths(m)$
                       addFusionPreventingEdge ($G_c, r_i, n$)
                   endforall
               endforall
           endif
       endforall
```

Alg. 1. Adding Sufficient Transitive Fusion-Preventing Edges to G_c

and m is in $G_{o'}$. Edges that become redundant may be removed. Because G_o is a DAG and $G_{c'}$ contains only safe fusions, $G_{o'}$ is also a DAG.

Example. For G_o in Fig. 2(a), the algorithm begins by extracting G_p and adds two fusion-preventing edges (P_4, P_7) and (P_4, P_8) as depicted in Fig. 2(b). The greedy partitioning fuses P_3 and P_4, and P_6 and P_7 in Fig. 2(c). This partition is translated onto G_o in Fig. 2(d). Fig. 2(e) through (g) show the formation, partitioning and translation of G_s onto $G_{o'}$.

5.1 Finding the Necessary Transitive Fusion-Preventing Edges

This section describes Alg. 1 *NecessaryEdges* which adds fusion-preventing edges to a component graph G_c. Without loss of generality, consider G_p. Intuitively, a fusion-preventing edge needs only be added to G_p when there exists a path with length greater than one between two parallel nodes and all the nodes on the path are sequential. These edges are sufficient because a path containing all parallel nodes just inherits its relationships from G_o and therefore need not be augmented. This characterization encompasses all cases since a path between two parallel nodes that contains both sequential and parallel nodes is just a sequence of paths between parallel nodes that either contain all sequential nodes or all parallel nodes. Definition 3 captures this notion formally.

Definition 3 *Two nodes $node_i$ and $node_j \in G_c$ of type c require a transitive fusion-preventing edge between them in G_c iff:*

$$\forall\ path \mid node_i \rightarrow node_h^+ \rightarrow node_j \ \in\ G_o \text{ where } \forall\, h,\ type(node_h) \neq c.$$

Based on this definition, Alg. 1 *NecessaryEdges* computes transitive fusion-preventing edges to be inserted into a G_c. *NecessaryEdges* formulates this problem similarly to a data-flow problem, except that solutions along an edge differ depending on node types.

The data structure *Paths* is used to recognize paths between nodes of the same type as G_c which only contain nodes h with $type(h) \neq c$. It stores sets of initial nodes for these paths and is initialized to the empty set. The traversal is in breadth-first order and begins by creating singleton sets, $Paths(n) = n$, where n is a root of G_o and is of type c. In step (1), *NecessaryEdges* visits node n with $type(n) \neq c$ and unions all the paths of n's predecessors. Thus, $Paths(n)$ where $type(n) \neq n$ contains the set of initial nodes of type c for paths that otherwise consist of nodes h where $type(h) \neq c$. In step (3), the visit of n when $type(n) = c$ inserts a fusion-preventing edge from each of the initial nodes in $Paths(m)$ where $(m, n) \in G_o$, and $type(m) \neq c$. At step (2), it sets $Paths(n) = n$ to begin any set of paths originating at n.

5.2 Discussion

Correctness and Optimality. The correctness and optimality of this algorithm are shown as follows. With the above construction, parallel nodes and sequential nodes are never placed in the same partition, satisfying the separation constraint. Because G_p is a DAG, minimizing the number of parallel partitions is achieved using the greedy algorithm (the proof of minimality from Callahan's dissertation is directly applicable [6].)

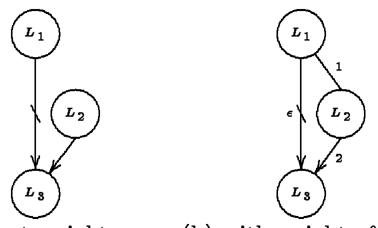
We now show that the construction of G_p does not introduce any constraints which would cause the component solution $G_{p'}$ to be non-optimal.

Proof. All the edges which are not fusion-preventing in G_p are in G_o by construction. We therefore need only prove there are no unnecessary fusion-preventing dependences between nodes that can be fused. The proof is by contradiction. Assume there is a fusion-preventing edge in G_p which prevents two parallel loops from being fused that can be fused. If the fusion-preventing edge was in G_o, we have a contradiction. If the fusion-preventing edge was not in G_o and the nodes can be fused then either the nodes were not connected by a path in G_o or there is no path between them in G_o which contains a sequential node. In either case, *NecessaryEdges* would not insert a fusion-preventing dependence. □

Although the same proof holds for G_s, the minimality of $G_{s'}$ is for the new problem posed in $G_{o'}$. The *total* solution therefore may not be minimal because partitioning G_p constrains fusions in G_s and may increase the number of sequential partitions. As an example, consider exchanging parallel and sequential nodes in Fig. 2(a) and then applying the algorithm. Similarly, partitioning G_s and G_p simultaneously may result in inconsistent fusions.

Complexity. This algorithm takes $O(N * E)$ time and space, making it practical for use in a compiler. In the worst case, *NecessaryEdges* computes the equivalent of a transitive closure to insert fusion-preventing edges. The greedy algorithm is linear in time and space.

```
L1  DO I = 1, N
        A(I) = D(I)
    ENDDO
L2  DO I = 1, N
        B(I) = C(I) + D(I)
    ENDDO
L3  DO I = 1, N
        ... = A(I-1) + C(I) + B(I)
    ENDDO
```



(a) without weights or input dependences

(b) with weights & undirected input dependences

Fig. 3. Modeling Memory Reuse

This approach may be applied to other graph partitioning problems as well. The separation of concerns lends itself to problems that need to sort or partition items of different types and priority while maintaining transitive relationships. This feature has been instrumental in designing a multilevel fusion algorithm [20]. The overall structure of the algorithm also enables different partitioning and sorting algorithms to be used on the component graphs. Sections 7 and 8 present partitioning algorithms based on reuse that can be used independently or in place of the greedy algorithm.

6 Loop Fusion for Reuse

Fusion can improve reuse by moving references closer together in time, making them more likely to still be in cache or registers. For example, reuse provided by fusion can be made explicit by using *scalar replacement* to place array references in a register [7]. However, fusion may also increase working set size, negatively impacting cache and register performance.

The effects of fusion on memory access are not captured in the representation used in the loop parallelization algorithm because only true and input dependences indicate opportunities for reuse.[3] The example in Fig. 3 and its representation as an unweighted graph in Fig. 3(a) illustrate the problem. In the original formulation for maximizing parallelism, there is no representation for the reuse of D(I) between loops L_1 and L_2, nor is the reuse of both C(I) and B(I), between loops L_2 and L_3 given any importance. By adding input dependences, edge weights, and factoring in dependence type, we include these considerations.

Input dependence, like true data dependence, captures reuse. It is not required to preserve correctness and therefore should not restrict statement order. These edges are undirected and result in a graph that is not a DAG (see Fig. 3(b)).

Edge weights represent the *amount* of reuse between two loops. For simplicity, true and input dependence edges have weight of one, output and anti dependences have weight zero, and fusion-preventing dependences have weight $\epsilon \ll 1$. If there exists

[3] Depending on the cache hardware and coherence mechanism, the time to write to cache may not change appreciably whether the corresponding line is in cache or not, the case for output or anti dependences.

more than one edge between two nodes, the edges are collapsed and the weights are added. The cumulative edge is directed if any of the collapsed edges are, otherwise if all edges are undirected, it is undirected. Any measure of reuse which results in a single numeric value could replace this measure.

We now show that the problem of finding a fusion that maximizes reuse is NP-hard. Consider the Multiway Cut problem proven NP-hard for $k \geq 3$ by Dahlhaus *et al.*[11]. Given a graph G = (V, E), a set S = $\{s_1, s_2, \ldots, s_k\}$ of k specified vertices called *terminals*[4] and undirected edges with weight $w(e) = 1$, find a minimum weight set of edges $E' \subset E$ for each edge $e \in E$ such that the removal of E' from E disconnects each terminal from all other terminals. To prove fusion for reuse NP-hard, we transform Multiway Cut to the Fusion problem. In so doing we establish that the subproblem with no input dependences is NP-hard.

Theorem 1: *Fusion for reuse is NP-hard.*

Proof. To establish the result we present a polynomial-time algorithm for reducing Multiway Cut to Fusion.

1. Select an arbitrary *terminal* $s \in V$ as a start node and number the nodes of V as follows. Initially, let the set *ready* contain all the successors of the start node. Number the start node 1. Select a vertex from *ready*, always picking a non-terminal vertex if one exists (*i.e.*, pick a terminal if only terminals are in the set). Give the selected vertex the next number and add to *ready* all its successors that are neither numbered nor in *ready*.
2. Construct a Fusion DAG G' in G by making each vertex in V be a loop node and orienting each edge from E such that the source of the edge has lower number than the sink in E'. None of these edges are fusion-preventing.
3. Let $\{s_1, s_2, \ldots, s_k\}$ be the terminals in the Multiway Cut problem, listed in order of their numbering. Add the following fusion-preventing edges oriented from lower to higher number to the DAG:

$$(s_1, s_2), \ldots (s_1, s_k), (s_2, s_3) \ldots, (s_2, s_k), \ldots (s_j, s_{j+1}), \ldots, (s_j, s_k), \ldots, (s_{k-1}, s_k)$$

The constructed Fusion problem has a minimal solution with weight m *iff* the corresponding Multiway Cut problem has the same solution.

Only If. Consider a weight m solution to the Fusion problem G' (m non-fusion-preventing dependence edges are cut). This solution corresponds to the removal of m undirected edges from the Multiway Cut problem G and is a weight m solution that disconnects terminals. Suppose the solution is not correct, *i.e.*, let $v_1, v_2, \ldots, v_p$ be a path in G from terminal $s_1 = v_1$ and terminal $s_2 = v_p$ such that no edge in the path is mapped to an edge broken in the solution to the Fusion problem. But an uncut edge in the Fusion solution, regardless of its orientation, means that the source and sink are in the same fusion group in G'. Thus all of $v_1, v_2, \ldots,$ and v_p may be fused, but this is precluded by the existence of a fusion-preventing edge between v_1 and v_p. This contradiction establishes that every undirected path between terminals in G must be cut as a part of the Fusion solution in G'.

[4] All other vertices are called *non-terminals.*

If. We show that the weight m solution to the Multiway Cut problem G corresponds to a correct weight m solution to the Fusion problem G'. Suppose there is a path between a terminal and a non-terminal $s_1, v_1 \in G$ after the cut but s_1 and v_2 cannot be fused in G' because there exists path from s_1 to v_2 in G' that passes through terminal s_2. In other words, the Multiway Cut solution places s_1 and s_2 in different partitions after the cut and s_1 and v_1 in the same partition, but since there is directed a path in G' from s_1 to s_2 to v_1, v_1 cannot fuse with s_1. However, this situation can never arise because of the numbering scheme.

Under the numbering scheme, if there is a path of non-terminals v_j connecting two terminals s_1 and s_2, where s_1 has the lower number, then all v_j have a smaller number than s_2 (because the numbering algorithm always numbers an available non-terminal vertex first). There can thus be no directed path from s_2 to any v_j. In particular, there can be no directed path from s_1 to v_j that passes through s_2, because each v_j has a lower number than s_2. Thus any solution of the Multiway Cut corresponds to a solution to the Fusion problem.

Minimality. Because solutions have the same weight and are correct in both, a minimal solution in one corresponds to a minimal solution in the other. □

The Fusion problem with input dependences must be NP-hard as well, because the subproblem without undirected input dependences is NP-hard; any algorithm that optimally solves the problem with input dependences must be able to solve the case without input dependences.

7 Improving Reuse with Loop Fusion: The Simple Algorithm

The simple algorithm is a straightforward modification of the greedy algorithm that moves loops to different partitions to improve reuse when legal. Like the greedy algorithm it is linear. Given a greedy solution that specifies loops in partitions, we increase the amount of reuse based on the following observation.

> In the greedy partition graph, if there exists two partitions g_1 and g_2 with a directed edge (g_1, g_2), then no node in g_2 may be legally placed in g_1 or the greedy algorithm would have put it there. However, it may be safe and profitable to move nodes in g_1 down into g_2.

We determine if it is safe and profitable to move $n \in g_l$ down into another partition g_h as follows.

Safety. A node $n \in g_l$ may move to g_h *iff* it has no successors in g_l and there is no fusion-preventing edge (n, r) such that $r \in g_h$ or $r \in g_k$ where g_k must precede g_h.

Profitability. Compute a sum of edges (m, n), $m \in g_l$, and a sum for each g_h of edges (n, p), such that $p \in g_h$ and n may be safely moved into g_h. Pick the partition g with the biggest sum. If $g \neq g_l$ move n down into g.

```
Simple ($G_g$)
  INPUT:     $G_g$ a greedy partition graph
  OUTPUT:    $G_{g'}$ a partitioning with improved reuse
  ALGORITHM:
    forall partitions $g_l \in G_g$ in reverse topological order
      forall nodes $n \in g_l$ in reverse topological order
        consider $g_h$ where $(n, p) \in G_g$ and $p \in g_h$
          if it is safe and profitable to move $n$ to $g_h$
            move $n$ to $g_h$
    return $G_{g'}$
```

Alg. 2. Simple Algorithm for Improving Reuse

Alg. 2 *Simple* applies these criteria to improve reuse. It begins by performing a bottom-up pass on the partition graph G_g formed by the greedy algorithm. For all nodes n within a partition, it performs a bottom-up pass testing if it is safe and profitable to move n down into another partition g_h. For Fig. 3, this algorithm produces a partition of $\{L_1\}$, $\{L_2, L_3\}$ correcting the initial greedy partition $\{L_1, L_2\}$, $\{L_3\}$.

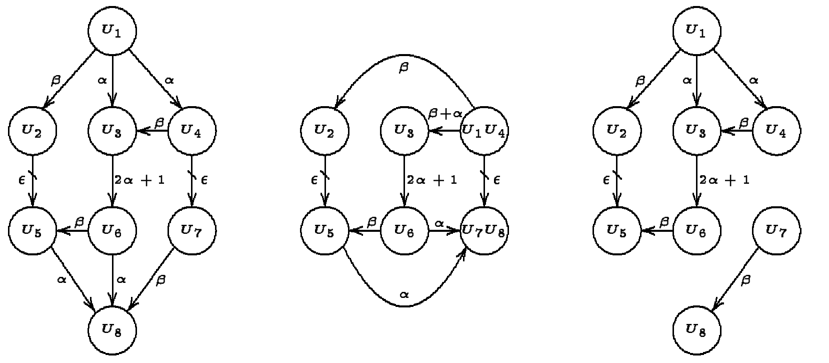
Further adjusting the results of the greedy algorithm for reuse on sequential loops, we only fuse loops in a given partition if they offer reuse. Given no reuse and negligible sequential loop overhead, it is more important to keep register pressure at a minimum than to decrease loop overhead. Since all the loops in a partition may be safely fused, we fuse only those connected by true or input dependences to improve reuse. Assuming fork and join of parallel tasks is relatively expensive, we still fuse all nodes in parallel partitions.[5] This decision may differ depending on the parallel architecture.

8 Improving Reuse with Maximum-Flow/Minimum-Cut: The Weighted Algorithm

In this section, we describe a more general and powerful algorithm which is a modification of the maximum-flow/minimum-cut algorithm that seeks to maximize reuse via loop fusion.

If there is only one fusion preventing dependence ($k = 1$), the graph can be divided in two using the maximum-flow/minimum-cut algorithm [13, 16]. Maximum-flow/minimum-cut is polynomial time and makes a single minimum cut that maximizes flow in the two resultant graphs. The maximum-flow algorithm works by introducing flow at the source such that no capacity is exceeded and the capacity of the flow network to the sink is maximized. To divide the graph in two parts, a cut is taken. The minimum cut consists of the edges that are filled to capacity and can be determined with a breadth-first search originating at the source of flow. If $k = 2$, the problem is polynomial time solvable by using two applications of the 2-way cut algorithm [26].

[5] Simple replaces the greedy algorithm on the component graphs in loop parallelization.



(a) original (b) collapsed (c) cut & un-collapsed

Fig. 4. Using Minimum-Cut for Fusion ($\beta > \alpha$)

Dahlhaus *et al.* develop an *isolation heuristic* [11] and combine it with Goldberg and Tarjan's maximum flow algorithm [16] to design a polynomial algorithm that produces a multicut that is at most (2k - 1)/k times the optimal. Goldberg and Tarjan's maximum flow algorithm is to date the most efficient with time complexity $O(nmlog(n^2/m))$ [16]. The running time of Dahlhaus *et al.*'s algorithm is $O(2^{2k}knmlog(n^2/m))$, which is polynomial for fixed k [11]. They leave open whether a more efficient algorithm with a similar optimality guarantee exists.

In the Fortran programs in our studies, k, n, and m have always been very small. The following section is devoted to the design of an $O(knmlog(n^2/m))$ algorithm for reuse problem. The algorithm is based on Goldberg and Tarjan's maximum-flow algorithm; it is guaranteed to be within k times the optimal. In some cases, it is optimal.

8.1 Structuring the Maximum-Flow/Minimum-Cut Problem

First, we perform a topological sort on the fusion-preventing edges and select the first fusion-preventing edge (*src, sink*) to cut. In maximum-flow/minimum-cut, *src* provides infinite flow and *sink* infinite consumption. In the minimum-cut algorithm, these additions force the fusion-preventing edge to be cut and in some cases, are sufficient to characterize the problem. For instance, infinite flow into L_1 and out of L_3 in Fig. 3(b) would cause (L_1, L_2) and (L_1, L_3) to be cut, therefore optimally fusing L_2 and L_3.

Consider breaking the fusion preventing dependence (U_4, U_7) in Fig. 4(a). Providing flow to U_4 and consumption to U_7 only cuts the fusion-preventing edge, but other edges must be cut as well to separate them. The insight is that to use minimum cut, the nodes that must precede *src* are *temporarily* collapsed into *src* and nodes that must follow *sink* are temporarily collapsed into *sink*, so that they truly represent the source and the sink of flow. However, collapsing all predecessors of *src* and all successors of *sink* can impose unnecessary ordering constraints on the resulting graph. We therefore use the control dependence relation to determine the nodes to temporarily collapse. After each application of minimum-cut, the nodes are "un-collapsed".

In the reuse graph G_r, we compute all control dependence ancestors of *src* and collapse them into *src* and *src* inherits their edges (self edges are ignored). For *sink*, we compute control dependence on the *reverse* graph of G_r, RG_r. The reverse graph simply reverses the direction of each edge in G_r. All control dependence ancestors of *sink* in RG_r are collapsed into *sink* in G_r and their edges inherited. The cut of this graph is applied to G_r the original, un-collapsed graph. The cut always includes the fusion-preventing edge and breaks the original graph into two subgraphs.

The algorithm collapses each subgraph with an uncut fusion-preventing edge, applies minimum-cut and un-collapsing as above. It repeats this process until all fusion-preventing edges are broken. For instance, one application of this process to Fig. 4(a) where $\beta > \alpha$ results first in the collapsed graph 4(b) and then in the cut graph 4(c). The next application cuts edges (U_2, U_5), (U_1, U_3) and (U_1, U_4).

Optimality. Given a graph with a single fusion-preventing edge, this algorithm performs optimally. For the general problem, given there will be at most k cuts and each cut is minimal, the total of the multicut will be at most k times greater than the optimal. This bound is a tight worst case, as illustrated by Fig. 4 where the total cost of the cuts made by our algorithm is 4α when $\beta > \alpha$. However, the optimal cut is (U_2, U_5), (U_3, U_6) and (U_4, U_7) with cost $2\alpha + 1$.

One solution to this inaccuracy would be to collapse all the sources of fusion-preventing dependences that have control dependence ancestors in common and cut all the corresponding sinks at once. Instead of multicuts, this construction would always make a single cut of the middle edge in Fig. 4(a). However, if the edge (U_3, U_6) has a weight greater than 4α then the multicut is better and this construction cannot find it. Differentiating between multicuts of degree 1, l and $l + 1$ makes these tradeoffs very difficult.

Complexity. This algorithm applies the $O(nmlog(n^2/m))$ maximum-flow algorithm k times splitting the graph at each application. For these types of graphs, computing control dependence is linear [10]. The complexity of our algorithm is therefore $O(knmlog(n^2/m))$ time.

9 Results

As our experimental vehicle we used the ParaScope Editor [21], an interactive parallelization tool which provides source-to-source program transformations using dependence analysis. We implemented the tests for correctness and the update of the source program and dependence information for fusion between two adjacent loops. We also implemented the correctness tests and updates for distribution to the finest granularity. We selected only programs which contained candidates for fusion or distribution to explore the impact on performance. In each program, performance improves.

9.1 Maximizing Parallelism

We performed the fusion algorithm for maximizing parallelism using ParaScope on 3 programs, Erlebacher, Seismic and Ocean. Both the fused and original hand-coded versions were compiled with the standard Fortran compiler and then executed in parallel on 19 processors of a Sequent Symmetry S81. The improvements due to fusion appear in Fig. 5.

	parallel execution time in seconds		
program	without fusion	with fusion	% improvement
Erlebacher	6.67	6.20	7%
Seismic	17.05	12.59	26%
Ocean	116.6	79.3	32%

Fig. 5. Improving Parallel Performance with Fusion

Erlebacher is an ADI benchmark program with 835 non-comment lines written by Thomas Eidson of ICASE. It performs tridiagonal solves on all dimensions of a three dimensional array ($50\times50\times50$). In the original version, many of the loops consisted of a single statement. Twenty-six loop nests benefit from fusion, improving parallel performance by 7% as illustrated in Fig. 5. Multiple fusions were performed which reduced the twenty-six nests to eight with fusion groups of size 2, 3, 4 and 5 loops. In the fusion problems posed, three consisted of seven loop nodes with two fusion-preventing dependences. In six of the eight, fusion improved reuse as well increased granularity.

Seismic is 1312 non-comment lines and performs 1-D seismic inversion for oil exploration. It was written by Michael Lewis at Rice University. In this program, there was one opportunity to fuse four loops into a single nest which improved parallel performance by 26%. The original nests were connected by data dependence (*i.e.*, contained reuse) and accounted for a significant portion of the total execution time. These nests were fused across procedure boundaries using *loop extraction* to place the nests in the same procedure [17, 22]. Loop extraction pulls a loop out of the called routine and into the caller, actually increasing procedure call overhead. The increased reuse and decreased parallel loop synchronization resulting from fusion more than overcame the additional call overhead.

Ocean is 3664 non-comment line program from the Perfect benchmark suite [9]. Fusion improved parallel performance by 32 % on Ocean. Thirty-one nests benefit from fusion across procedure boundaries [17, 22]. Some of the candidates were exposed after constant propagation and dead code elimination. Loop extraction enabled fusion. (Again, extraction's only effect is to increase total execution time because of increased call overhead.) The fused nests consisted of between two and four parallel loops from the original program and increased both reuse and granularity. As Seismic and Ocean indicate, fusion is especially effective across procedure boundaries.

	seconds		
Erlebacher	original	fused	% improvement
IBM RS6000	.813	.672	4.25 %
Intel i860	.548	.518	5.47 %
Sun Sparc2	.400	.383	17.34 %

Fig. 6. Effect of Fusion on Erlebacher for Uniprocessors

9.2 Improving Reuse on Uniprocessors

In this section, we illustrate the benefits of fusion and of loop distribution to improve uniprocessor performance.

Erlebacher. In this experiment, we applied loop fusion only to loop nests with reuse using the simple algorithm. For two fusion graphs, the greedy algorithm was optimal for reuse. One required the simple fusion algorithm to obtain maximum reuse and in three, the algorithms reordered the loops to achieve better reuse. In Fig. 6, we compare the fused and original versions of the same nests on three modern microprocessors. Improvements due to fusion ranged from 4 to 17 percent.

Gaussian Elimination. This kernel illustrates that distrubition and fusion should not be considered only in isolation; the improvements due to fusion and distribution will be enhanced by using them in concert with other transformations. In Gaussian Elimiation, the benefit of loop distribution comes from its ability to enable loop permutation. In the original KIJ form in Fig. 7, the J loop has poor data locality for Fortran 77 where arrays are stored in column-major order. Placing the I loop innermost for statement S_2 would instead provide stride-one access on the columns. Using loop distribution enables the interchange and the combination significantly improves the execution time. The algorithm that evaluates and applies loop permutation with distribution appears elsewhere [8].

Loop Distribution & Interchange

```
       { KIJ form }                      ⇒           { KJI form }
    DO K = 1,N                                    DO K = 1,N
      { select pivot, exchange rows }               { select pivot, exchange rows }
      DO I = K+1,N                                  DO I = K+1,N
S1      A(I,K) = A(I,K) / A(K,K) - 1.1                A(I,K) = A(I,K) / A(K,K) - 1.1
        DO J = K+1,N                                DO J = K+1,N
S2        A(I,J) = A(I,J) - A(I,K)*A(K,J)             DO I = K+1,N
                                                        A(I,J) = A(I,J) - A(I,K)*A(K,J)
```

	256 × 256		
		distribution	
Processor	original	& interchange	% improvement
Sun Sparc2	12.23	5.22	57 %
Intel i860	8.35	2.63	69 %
IBM RS6000	27.29	4.84	82 %

Performance Results in Seconds

Fig. 7. Gaussian Elimination

9.3 Discussion

The fusion problems we encountered were fairly simple indicating that the complexity of the weighted algorithm is perhaps unnecessary. This preliminary evidence also indicates that the flexibility of the simple algorithm is required, *i.e.* the greedy algorithm is not sufficient. For example, reordering the loops in *Erlebacher* and improving on the greedy solution were both necessary to fully exploit reuse. Our results for these programs show a performance improvement every time fusion was applied to improve reuse or increase the granularity of parallelism.

Fusion is especially important for Fortran 90 programs because of the array language constructs. To compile Fortran 90 programs, the array notation is expanded into loop nests containing a single statement, providing many opportunities for fusion. Fusion and distribution also enable loop interchange, making them an important component for many optimization strategies.

10 Related Work

In the literature, fusion has been recommended for decreasing loop overhead, improving data locality and increasing the granularity of parallelism [2, 23]. Abu-Sufah and Warren have addressed in depth the safety, benefits and simple application of loop fusion [1, 25]. Neither presents a general algorithm for loop fusion that supports loop reordering or any differentiation between fusion choices. Goldberg and Paige [15] address a related fusion problem for stream processing, but their problem has constraints on fusions and ordering that are not present in the general fusion problem we address. Allen, Callahan, and Kennedy consider a broad fusion problem that introduces both task and loop parallelism, but does not address improving data locality or granularity of loop parallelism [3, 6]. Our algorithm for maximizing loop parallelism and its granularity is a new result.

Sarkar and Gao present an algorithm to perform loop fusion and array contraction to improve data locality on uniprocessors for single assignment languages [24]. This work is limited because it does not account for constraints imposed by data dependence [14, 24]. Their more recent work on loop fusion for uniprocessors and pipelining [14] takes into consideration data dependence constraints and also is based on the maximum-flow/minimum-cut algorithm. (Our work was developed independently.) Our algorithm is distinguished because of its tight worst-case bound. Both of our reuse algorithms reorder loop nests which is not possible in their formulation. The results indicate this flexibility is necessary. In practice, our preliminary experimental results also indicate that the additional complexity of the maximum-flow/minimum-cut algorithms is probably not necessary. Our overall approach is more flexible because it optimizes for both multiprocessors and uniprocessors.

For parallel code generation for shared-memory multiprocessors, this work extends our previous work by providing comprehensive fusion and distribution algorithms [19, 22]. In Carr, Kennedy, McKinley and Tseng, they combine loop permutation with fusion and distribution to improve data locality on uniprocessors [8]. The algorithms presented here are complementary to this work.

11 Summary

This paper presents an optimal algorithm for performing loop fusion and its dual, loop distribution, to maximize the granularity of loop parallelism, therefore minimizing sychronization. We prove that finding a fusion that results in maximum reuse is NP-hard and describe two heuristics to perform fusion and distribution based on reuse. The reuse algorithms work independently or as elements in the parallelization algorithm. The first algorithm for improving data locality uses a variant of the greedy algorithm and is linear in time. In practice, it may be that this algorithm is sufficient. However, the more ambitious algorithm may be beneficial for languages such as Fortran 90 that can pose difficult fusion problems. These algorithms are flexible and allow loop reordering to achieve a desired and safe partition. This paper also provides a general framework for solving loop fusion and loop distribution problems.

Acknowledgments

While pursuing the research on the weighted partition problem, we thank Alejandro Schaffer and Mark Krentel for their advice and expertise during several fruitful discussions. We also acknowledge Chau-Wen Tseng, Preston Briggs, Amer Diwan, Mary Hall and Nathaniel McIntosh for their valuable comments on numerous drafts of this paper.

References

1. W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.
2. F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, 1972.
3. J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, Jan. 1987.
4. J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.
5. A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, Oct. 1966.
6. D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Dept. of Computer Science, Rice University, Mar. 1987.
7. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.
8. S. Carr, K. Kennedy, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. Technical Report TR92-195, Dept. of Computer Science, Rice University, Nov. 1992.

9. G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.
10. R. Cytron, J. Ferrante, and V. Sarkar. Experiences using control dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* The MIT Press, 1990.
11. E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, May 1992.
12. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
13. Ford, Jr., L. R. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, NJ, 1962.
14. G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992.
15. A. Goldberg and R. Paige. Stream processing. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 228–234, Aug. 1984.
16. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the Association for Computing Machinery*, 35(4):921–940, Oct. 1988.
17. M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.
18. K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, New York, NY, Nov. 1990.
19. K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.
20. K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993.
21. K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice & Experience*, to appear 1993.
22. K. S. McKinley. *Automatic and Interactive Parallelization.* PhD thesis, Dept. of Computer Science, Rice University, Apr. 1992.
23. A. Porterfield. *Software Methods for Improvement of Cache Performance.* PhD thesis, Dept. of Computer Science, Rice University, May 1989.
24. V. Sarkar and G. Gao. Optimization of array accesses by collective loop transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.
25. J. Warren. A hierachical basis for reordering transformations. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, Salt Lake City, UT, Jan. 1984.
26. M. Yannakakis, P. C. Kanellakis, S. C. Cosmadakis, and C. H. Papadimitriou. Cutting and partitioning a graph after a fixed pattern. *Automata, Languages, and Programming - Lecture Notes in Computer Science*, 154:712–722, 1983.

This article was processed using the LaTeX macro package with LLNCS style