

# Resource-Constrained Loop Fusion

Ken Kennedy

Center for High Performance Software  
Rice University  
ken@rice.edu

Chen Ding

Computer Science Department  
University of Rochester  
cding@cs.rochester.edu

## Abstract

Embedded processors have limited on-chip memory. Fusing loops that use the same data can reduce the distance between accesses to the same memory location, avoiding costly off-chip memory transfer. Most existing greedy fusion algorithms solve the *unconstrained* problem—they do not guard against negative effects of excessive fusion. When a large program contains a great number of loops, unconstrained fusion may generate huge loops that overflow on-chip memory, leading to lower performance.

This paper studies the problem for *constrained weighted fusion*, in which the graph edges carry *weights* indicating the profitability of fusing the inputs and vertices are annotated with *resource requirements*. The optimal solution of a constrained weighted fusion problem is a collection of vertex sets such that the total weight associated with pairs of vertices within clusters is maximized and the aggregate resource requirement of every cluster is less than a fixed upper bound  $R$ . Finding the optimal solution to a weighted fusion problem (constrained or unconstrained) is NP-complete, so we use heuristics. We present two methods. The first picks a group of loops at each fusion step. To ease the resource calculation and fusibility test, the second method picks only a pair of candidate loops at each step. The paper presents the two algorithms, their complexity, and an experimental evaluation.

## 1.0 Introduction

Embedded systems often have limited memory and communication capacity for reasons of both cost and energy efficiency. Consequently, transforming code to increase data reuse in programs is an extremely important compiler optimization—it can shorten average memory latency by orders of magnitude by keeping data in local memory between uses.

Ideally, all reuses of the same datum would happen at the same time, so that it would be evicted from cache only when

it is no longer useful. In typical applications, however, data reuses are widely distributed across the program. For example, a program may iterate over an array at the beginning of a calculation and again at the end. There is almost no hope that any of the data accessed in the second loop will be found in cache.

One way to bring data reuses closer together is to merge loops that access the same data. This transformation, known as *loop fusion*, interleaves iterations of multiple loops to bring data reuses into the same or adjacent iterations. The fundamental problem of *global loop fusion* is to select maximal subgroups of loops that may be compatibly fused—that is, for which fusion is not precluded by correctness considerations.

Loop fusion can be formulated as a graph clustering problem in which each edge is weighted by the amount of savings to be gained by fusing its endpoints. Since the problem of weighted fusion has been shown to be NP-complete [17], a heuristic strategy is needed to solve this problem. One promising heuristic is *greedy weighted fusion*, which seeks to fuse the heaviest edge (i.e., the edge with the most savings) on each step. Recently, Kennedy [15] produced an algorithm for greedy weighted fusion that runs in  $O((E+V)V)$  time, where  $E$  is the number of edges in the fusion graph and  $V$  is the number of vertices. This time bound is important because in array languages like Fortran 90, where each array statement in the program becomes a small loop, large programs can produce thousands or even tens of thousands of candidates for loop fusion.

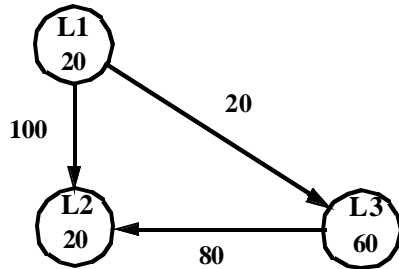
Although fast, Kennedy’s fusion algorithm does not take resource constraints into account. For example, if the fused loop accesses too much data, it may not keep all its data in local memory and consequently suffer from the cost of off-chip communication. Furthermore, on machines with multiple levels of memory, the data utilization of the fused loop might also exceed the storage capacity of level-1 memory while attempting to enhance reuse from level-2 memory. The advent of aggressive new outer-loop fusion schemes [9] make these scenarios even more likely. What is needed to address this problem is some scheme for ensuring that the total cost of a group of fused loop does not exceed a given maximum.

In this paper we present two constrained fusion algorithms that overcome these limitations. First we show a simple adaptation of Kennedy’s algorithm to take constraints into account. Second, we present a modified strategy that simplifies the consideration of resource constraints.

We illustrate the difference between the two methods with an example. Figure 1 shows three loops as nodes with the resource requirement marked inside and three dependences as edges with the amount of data reuse marked by each edge. Our first method, like Kennedy’s algorithm, would pick the heaviest edge, which connects from node 1 to node 2. However, due to data dependence, fusing the first two nodes requires the fusion of the third node. In general, the first method may involve any number of loops at each step. If a group of loops cannot be fused because of either legality or resource constraints, the fusion algorithm will abort the current step and move to the next heaviest edge.

Our second method comes from the observation that testing a pair of loops for constrained fusion is easier than testing a group of loops. It uses a slightly different heuristic, which is to pick the heaviest edge that do not require fusion of any vertices other than the endpoints of the edge. The edge between the first two nodes in Figure 1 will not be considered since it involves more than two nodes in loop fusion.

FIGURE 1 Constrained fusion example.



In most studies of fusion, addition is used as the reweighting operation. However, in fusion for the purpose of increasing reuse, the reweighting operation can be much more complicated. If the two original edges were associated with different array variables, addition is indeed the right reweighting operator. However, if the edges are associated with the same array, then the reweighting operation must determine the extent to which the range of access in the fused loop overlaps with the usage of the same array in the loop at the other endpoint. If all the edges in Figure 1 were due to a single array, the correct weight of the edge from the composite loop to L2 would be closer to 100 than to 200. Similar reweighting is needed for calculating the combined resource requirement after loop fusion. Both of our methods use an accurate reweighting scheme, as describe in Section 5.0.

The rest of the paper is organized as follows. Section 2.0 defines the fusion graph and the problem of constrained loop fusion. Section 3.0 and Section 4.0 present two algorithms for constrained fusion. Section 5.0 describes the dynamic updates during constrained fusion. Section 6.0 evaluates constrained fusion. Section 7.0 discusses related work and Section 8.0 concludes.

## 2.0 The Constrained Fusion Problem

To map loop fusion onto an abstract problem, we define a special kind of graph that contains both directed and undirected edges. In our problem, directed edges represent data dependences and undirected edges model input dependences, i.e. data sharing among memory reads.

**Definition 1.** A *mixed-directed graph* is a triple

$$M = (V, E_d, E_u)$$

where  $(V, E_d)$  forms a directed graph and  $E_u$  is a collection of undirected edges between vertices in  $V$ . By convention, no pair of vertices can be joined by both a directed and an undirected edge—that is,  $E_d \cap E_u = \emptyset$ .

**Definition 2.** A mixed-directed graph is said to be *acyclic* if  $G_d = (V, E_d)$  is acyclic. A vertex  $w$  is said to be a *successor* of vertex  $v$  if there is a directed edge from  $v$  to  $w$ , i.e.,  $(v, w) \in E_d$ . In this case, vertex  $v$  is a *predecessor* of vertex  $w$ . A vertex  $v$  is said to be a *neighbor* of vertex  $w$  if there is an undirected edge between them. From above, note that a neighbor of vertex  $v$  cannot also be a successor or a predecessor of  $v$ .

**Definition 3.** Given an acyclic mixed-directed graph  $M = (V, E_d, E_u)$ , a *weight function*  $W$  defined over the edges in  $E_d \cup E_u$ , a *cost function*  $C$  that maps any subset of vertices in  $V$  to the total resource requirement of the subset, a *legality predicate*  $Q(e)$  that is true if fusing the endpoints of edge  $e$  is legal, and a *maximum resource constraint*  $R$  the *constrained weighted loop fusion problem* is the problem of finding a collection of vertex sets  $\{V_1, V_2, \dots, V_n\}$  such that

- a. The collection covers all vertices in the graph, i.e.:
 
$$\bigcup V_i = V,$$
- b. The vertex sets form a partition of  $V$ , i.e.,
 
$$\forall i, j, 1 \leq i, j \leq n, (V_i \cap V_j) \neq \emptyset \rightarrow i = j,$$
- c. If each of the vertex sets  $V_i$  is reduced to a single vertex with the corresponding natural edge reduction, the resulting graph is acyclic.
- d. For any  $i$ , if  $e$  is an edge between two vertices in  $V_i$ , then  $Q(e)$ , i.e., fusion along edge  $e$  is legal.
- e. The total weight of edges between vertices in the same vertex set, summed over all vertex sets is maximized.
- f. For each  $V_i$ ,  $C(V_i)$  is no more than  $R$ , i.e., no fusion group requires more than the maximum available resource.

## 3.0 Greedy Constrained Fusion

We begin with a discussion of Kennedy’s algorithm for greedy weighted fusion and show how it can be extended to handle resource constraints. We begin with an overview of the implementation that will illuminate some of the algorithmic issues. The algorithm can be thought of as proceeding in six stages:

1. Initialize all the quantities and compute initial successor, predecessor, and neighbor sets. This can be implemented in  $O(E+V)$  time.
2. Topologically sort the vertices of the directed acyclic graph. This takes  $O(E_d + V)$  time.

3. Process the vertices in  $V$  to compute for each vertex the set  $pathFrom[v]$ , which contains all vertices that can be reached by a path from vertex  $v$  and the set  $badPathFrom[v]$ , a subset of  $pathFrom[v]$  that includes the set of vertices that can be reached from  $v$  by a path that contains a bad vertex. This phase can be done in time  $O(E_d + V)$  set operations, each of which takes  $O(V)$  time.
4. Invert the sets  $pathFrom$  and  $badPathFrom$  respectively to produce the sets  $pathTo[v]$  and  $badPathTo[v]$  for each vertex  $v$  in the graph. The set  $pathTo[v]$  contains the vertices from which there is a path to  $v$ ; the set  $badPathTo[v]$  contains the vertices from which  $v$  can be reached via a bad path. Inversion can be done in  $O(V^2)$  total time by simply iterating over  $pathFrom[v]$  for each  $v$ —adding  $v$  to  $pathTo[w]$  for every  $w$  in  $pathFrom[v]$ .
5. Insert each of the edges in  $E = E_d \cup E_u$  into a priority queue  $edgeHeap$  by weight. If the priority queue is implemented as a heap, this takes  $O(E \lg E) = O(E \lg V)$  time.
6. While  $edgeHeap$  is non-empty, select and remove the heaviest edge  $(v,w)$  from it. If  $w \in badPathFrom[v]$  then do not fuse—repeat step 6. Otherwise, do the following:
  - a. Collapse  $v$ ,  $w$ , and every edge on a directed path between them into a single node.
  - b. After each collapse of a vertex into  $v$ , adjust the sets  $pathFrom$ ,  $badPathFrom$ ,  $pathTo$ , and  $badPathTo$  to reflect the new graph. That is, the composite node will now be reached from every vertex that reaches a vertex in the composite and it will reach any vertex that is reached by a vertex in the composite.
  - c. After each vertex collapse, recompute *successor*, *predecessor* and *neighbor* sets for the composite vertex and recompute weights between the composite vertex and other vertices as appropriate.
  - d. Kennedy has shown that the entire algorithm can be implemented in  $O(EV + V^2)$  time.

To extend this algorithm to handle resource constraints, we must be able to determine whether the collapse operation in step 6a will create a vertex that exceeds the maximum resource constraint  $R$ . Given that one computes resource requirements for a composite vertex by summing the resource requirements of the component vertices, we can make the required determination by simply computing the region of collapse by the following expression:

$$S := pathFrom[v] \cap pathTo[v] \cup \{v, w\};$$

where the explicit inclusion of the endpoints is there to handle the case where the selected edge is undirected. Since the  $pathFrom$  and  $pathTo$  sets are  $O(V)$  in size, this operation takes  $O(V)$  time. We can then iterate over the set  $S$  in  $O(V)$  time, summing to compute the resource requirement for the entire region of collapse. Since this process will be done only  $O(E)$  times, the total work is  $O(EV)$ , which does not increase the asymptotic running time of the algorithm. Note that this procedure works because Kennedy’s algorithm correctly recomputes the  $pathFrom$  and  $pathTo$  sets after each collapse, without exceeding the  $O(EV + V^2)$  time bound.

A more detailed version of the algorithm driver is given in Figure 2. Except for *ExceedsResourceConstraints*, the sub-procedures that are invoked from the algorithm in Figure 2 can all be found in Kennedy’s papers [15,16] and will not be repeated here. The implementation of *ExceedsResourceConstraints* is straightforward from the discussion above.

**FIGURE 2** Greedy weighted fusion.

**procedure** *WeightedFusion*( $M, B, W$ )

```

//  $M = (V, E_d, E_u)$  is an acyclic mixed-directed graph
//  $B$  is the set of bad vertices
//  $W$  is the weight function

//  $pathFrom[v]$  contains all vertices reachable from  $v$ ;
//  $badPathFrom[v]$  contains vertices reachable from  $v$ 
// by a path containing a bad vertex
//  $edgeHeap$  is a priority queue of edges

```

```

P1: InitializeGraph( $V, E_d, E_u$ );
topologically sort the vertices using directed edges;

```

```

edgeHeap :=  $\emptyset$ ;

```

```

P2: InitializePathInfo( $V, edgeHeap$ );

```

```

L1: while edgeHeap  $\neq \emptyset$  do begin

```

```

    select and remove the heaviest edge  $e = (v,w)$ 
    from edgeHeap;

```

```

    if  $v \in pathFrom[w]$  then swap the names of  $v$  and  $w$ 
    (so  $v$  must precede  $w$ );

```

```

    if  $w \in badPathFrom[v]$  then
        continue L1; // cannot or need not be fused

```

```

     $S := pathFrom[v] \cap pathTo[v] \cup \{v, w\}$ ;

```

```

    if ExceedsResourceConstraints( $S$ ) then
        continue L1; // do not fuse

```

```

    // Otherwise fuse  $v, w$ , and vertices between them

```

```

    Collapse( $v, S$ );

```

```

end L1

```

```

end WeightedFusion

```

## 4.0 Constrained Fusion On Prime Edges

The second heuristic we use in this paper is a variant on the greedy strategy. Recall that a standard implementation of the greedy strategy would iteratively select the edge of highest weight, determine whether the endpoints can be fused, and, if possible, fuse the endpoints of that edge, along with all edges on a path between them into the same vertex set. In our version, we fuse only those edges that do not require fusion of any vertices other than the endpoints of the edge, thus making it easier to determine whether the fusion is both legal and within the given resource constraints.

**Definition 4.** In a graph with both directed and undirected edges, an edge  $e$ , directed or undirected, is said to be *prime* if there is no directed path in the graph from its source to its sink other than  $e$ .

In Figure1, the edges from L1 to L2 and L2 to L3 are prime but the edge from L1 to L3 is not. If we fuse only prime edges, then we can easily compute the cost of each fusion because, unlike Kennedy’s algorithm, we do not need to concern ourselves with implied secondary fusions. In Figure1, if we attempted to fuse the non-prime edge from L1 to L3, we would have to fuse L2 as well to preserve the correctness of the program.

By using only prime edges, we strive for two important goals. The first is profitability. For prime edges, fusion involves only the two end vertices, so we can always pick the prime edge that is the most profitable. For non-prime edges, however, the benefit and cost of fusion depend on all vertices involved in secondary fusions. When the fusion is too costly due to secondary fusions, the effort has to be restarted on sub-groups of already considered loops. This leads to our second point: using prime edges is efficient. For a prime edge, we can calculate the benefit and cost of fusion in one step. For a non-prime edge, the calculation needs to examine an arbitrary number of vertices, which include all vertices in the worst case. Since loop fusion is intended for optimizing large programs, we believe that bounding the cost of each fusion step is important.

To implement this strategy, at each fusion step we need to recompute any quantities that are affected by the fusion itself. In particular, we must determine if the fusion has caused a prime edge to become non-prime. For example, consider the example in Figure3. All the edges in this graph are prime. However if we fuse L1 and L2, the edge from the fused group to L4 is no longer prime. These recomputations will be carried out in a post-fusion update phase.

The constrained fusion algorithm incorporates the following steps:

FIGURE 3 Prime edges after fusion.

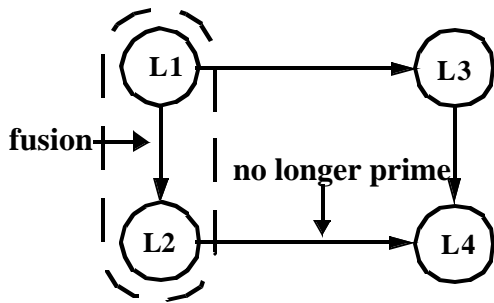


FIGURE 4 Constrained fusion.

**procedure** *ConstrainedFusion*( $M, B, W$ )

```
//  $M = (V, E_d, E_w)$  is an acyclic mixed-directed graph
//  $W$  is the weight function

//  $pathTo[v]$  contains all vertices reachable from  $v$ ;
//  $edgeHeap$  is a priority queue of edges
//  $rep[x]$  is the node in which  $x$  is fused into
```

```
S1: InitializeGraph( $V, E_d, E_w$ );
    $edgeHeap := \emptyset$ ;
S2: InitializePrimeEdges( $V, edgeHeap$ );

L1: while  $edgeHeap$  is not  $\emptyset$  do begin
S3:   select and remove the heaviest edge  $e = (v, x)$ 
      from  $edgeHeap$ ;
      if  $x \in pathTo[v]$  then swap  $v$  and  $x$ ;
S4:   if not FusionLegal( $e$ ) or  $e.Cost > R$  then
      continue L1; // cannot or need not be fused

      // Otherwise fuse  $v$  and  $x$ 
S5:    $rep[x] = v$ ;
      ComputeCostBenefit( $v, x$ );

      // update  $pathTo$  sets
S6:   UpdatePrimeEdges;

      // update the graph representation
S7:   UpdateSuccessors( $v, x$ );
      UpdatePredecessors( $v, x$ );
      UpdateNeighbors( $v, x$ );

S8:   remake  $edgeHeap$ ;

      // delete vertex  $x$ 
      delete  $x, predecessors[x], successors[x],$ 
          $neighbors[x], pathTo[x]$ ;
      delete  $x$  from  $successors[v]$  or  $neighbors[v]$ ;
end L1
```

**end** *ConstrainedFusion*

1. Construct the graph along with all edge weights, successors and predecessors.
2. Identify and mark all the prime edges in the graph and insert the edges into a priority queue by weight.
3. Select and delete the heaviest edge from the priority queue. Test the fusion for legality as follows:
  - a. Check to ensure that the fusion is legal.
  - b. Check to ensure that the cost of fusing the endpoints does not exceed the resource limit  $R$  for the problem.
4. If the fusion is not legal or overly costly, repeat step 3; otherwise go on to step 5.
5. Fuse the endpoints into a single vertex.
6. Determine whether any edges have become non prime as a result of the fusion and delete them from the priority queue.
7. Update the successor and predecessor data structures to reflect the fusion; in the process update the weight and cost of any edge incident on the collapsed node.
8. Remake the priority queue.
9. Return to step 3.

A more detailed version of this algorithm is given in Figure 4, in which the labels correspond to individual steps described above.

FIGURE 5 Initialize *prime* edges.

**procedure** *InitializePrimeEdges*( $V, edgeHeap$ )

```

// V is the set of vertices in the graph
// edgeHeap is the priority queue of edges by weight.
topologically sort the vertices using directed edges;
visited := ∅;
for each v ∈ V in topological order do begin
  pathTo[v] := {v}; visited := visited ∪ {v};
  for each w ∈ predecessors[v] do
    pathTo[v] := pathTo[v] ∪ pathTo[w];
  for each w ∈ neighbors[v] do
    if w ∈ pathTo[v] then begin
      delete w from neighbors[v];
      delete v from neighbors[w];
      predecessors[v] := predecessors[v] ∪ {w};
      successors[w] := successors[w] ∪ {v};
    end
  for each w ∈ predecessors[v] ∪
    (neighbors[v] ∩ visited) do begin
    edgePrime := true;
    for each x ∈ predecessors[v] - {w} do
      if w ∈ pathTo[x] then begin
        edgePrime := false; exit loop;
      end
    if edgePrime then add (w, v) to edgeHeap;
  end
end
make edgeHeap into a heap;
end InitializePrimeEdges

```

#### 4.1 Algorithm Analysis

It is fairly easy to analyze the complexity of this algorithm. The initialization step S1, which is not shown, simply builds the *successor*, *predecessor* and *neighbor* lists from the input edge lists in  $O(E+V)$  time. Construction of the initial priority queue (step 2) requires determination of prime edges. This process is shown in Figure 5. It requires a pass through the graph that is similar to a reachability calculation, which takes  $O((E+V)V)$  time. It replaces undirected edges with directed ones if the two end-nodes are connected by a directed path. If we make the heap on the last step rather than incrementally, construction takes  $O(E_p)$  time where  $E_p$  is the number of prime edges in the problem graph.

The edge selection step (S3) is executed at most  $E_p$  or  $E$  times. At each execution, the heaviest edge is selected and removed from the queue. This requires  $O(E \lg E) = O(E \lg V)$  time in the aggregate. The test for correctness is carried out in the routine *FusionLegal*, which is invoked  $O(E)$  times. In many cases the test for legality will be constant-time. However, we use a more expensive test discussed in Section 5.0.

The remainder of the body of loop L1, representing steps 5 through 8, is executed at most  $O(V)$  times because each execution devours one vertex. Thus the cost of step 5 is determined by the complexity of computing the cost and benefit for the fused vertex in the call to *ComputeCostBenefit*, which is invoked  $O(V)$  times. In Section 5.0, we propose a specific algorithm for this function and analyze its cost.

Step 8, the remake of the edge heap, can be done in time  $O(E)$ , so the entire cost of remaking the heaps after each collapse is  $O(EV)$ . All that remains is the cost of updating the data structures, which is discussed in the next section.

#### 4.2 Updating Data Structures After Fusion

Once the edge to be collapsed has been identified, there are two remaining tasks. First, edges that are no longer prime must be identified and removed from the graph. Second, the successor, predecessor, and neighbor data structures must be updated to reflect the new graph structure. At the same time, the costs associated with each edge incident on the collapsed edge must be recomputed. The following subsections treat these issues.

##### 4.2.1 Discovering Non-prime Edges

The graph in Figure 6 illustrates the problem of discovering when edges are no longer prime as the result of a fusion. When the edge from vertex  $v$  to vertex  $x$  is fused into a single vertex, the edge from  $a$  to  $v$  is no longer prime because of the path through  $b$  and the edge from  $x$  to  $d$  is no longer prime because of the path through  $c$ . The general case is illustrated by the edge from  $e$  to  $f$ . Because there is a path from  $e$  to  $x$  and from  $v$  to  $f$ , after fusion the path from  $e$  through the fused node and back to  $f$ , makes the edge  $(e, f)$  non-prime.

We address this problem by computing two sets *PathToXnoV* and *PathFromVnoX*. Here the names are intended to be synonymous with the function of the set. *PathToXnoV* contains all the vertices from which there is a path to vertex  $x$  that does not contain vertex  $v$ . Similarly, *PathFromVnoX* contains the vertices that are reachable from vertex  $v$  by a path that does not pass through vertex  $x$ .

An edge becomes non-prime if it has a source in *PathToXnoV* and a sink in *PathFromVnoX*. These two sets can be easily computed by two sweeps through the graph. The entire process is shown in Figure 7 and Figure 8.

The key to this algorithm is the routine *ComputePaths* that sweeps forward from  $v$  in the first call to compute the set *PathFromVnoX* and backward from  $x$  in the second call to compute the set *PathToXnoV*. The code for this routine is shown in Figure 8.

The key observation is that we can avoid all paths through  $x$  by simply not putting  $x$  on the worklist in the first step. This works because the edge  $(v, x)$  is prime, therefore the only path from  $v$  to  $x$  is via the direct edge.

*ComputePaths* is clearly  $O(E+V)$  in complexity and so the entire *UpdatePrimeEdges* takes  $O(E+V)$  because the iteration over edges is easy to implement in  $O(E)$  time. There is one special case that must be handled carefully. If the edge  $(v, x)$  selected for fusion is an undirected edge, it will be necessary to call the routine *UpdatePrimeEdges* once for  $(v, x)$  and once for  $(x, v)$  to ensure that all non-prime edges are properly eliminated.

FIGURE 6 New non-prime edges.

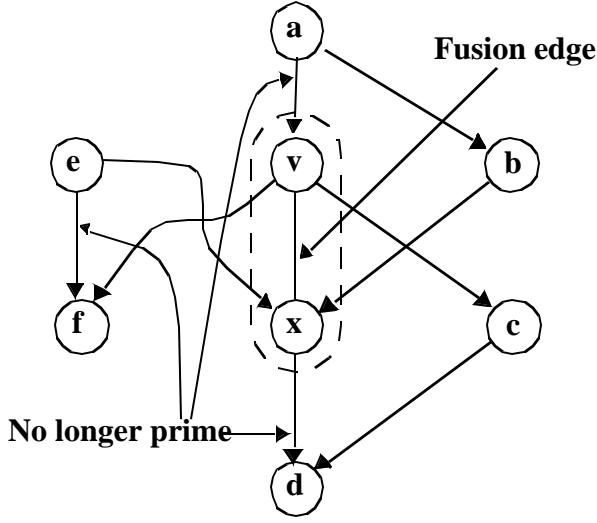


FIGURE 7 Finding edges that are no longer prime.

```

procedure UpdatePrimeEdges( $v, x, edgeHeap$ )
    //  $x$  is the vertex being fused into vertex  $v$ 
    //  $edgeHeap$  is the priority queue of prime edges

    ComputePaths( $v, x, successors, PathFromVnoX$ );
    ComputePaths( $x, v, predecessors, PathToXnoV$ );
    for each prime edge  $e = (a, b) \in edgeHeap$  do
        if ( $a \in PathToXnoV$  and  $b \in PathFromVnoX$ ) or
            ( $a \in PathFromVnoX$  and  $b \in PathToXnoV$ )
        then delete  $e$  from  $edgeHeap$ ;
    end UpdatePrimeEdges
  
```

FIGURE 8 A sweep through the graph.

```

procedure ComputePaths( $v, x, cesors, pathSet$ )
    //  $v$  is the vertex from which paths are being traced
    //  $x$  is the vertex to be avoided
    //  $cesors(b)$  is the successor set for the vertex  $b$ ;
    //  $pathSet$  is the output set of vertices reachable from  $v$ 
    // by a path not including  $x$ 

     $pathSet := \{v\}$ ;  $worklist := \emptyset$ ;
    L1: for each  $z \in cesors(v) - \{x\}$  do
         $worklist := worklist \cup \{z\}$ ;
    L2: while  $worklist$  is not  $\emptyset$  do begin
        pick and remove an element  $y$  from  $worklist$ ;
         $pathSet := pathSet \cup \{y\}$ ;
    L3: for each  $z \in cesors(y) - \{x\}$  do
         $worklist := worklist \cup \{z\}$ ;
    end
end ComputePaths
  
```

### Updating Incident Edges

We will illustrate the process of updating the edges incident on the collapsed region by showing the code for updating

successors (Figure9). This procedure is invoked once for each vertex  $x$  that is collapsed into another vertex  $v$ . Thus, it can be invoked at most  $O(V)$  times during the entire run of the algorithm.

An important function of this procedure is to reweight edges that are incident on the collapsed region. If there are edges from each of the fused endpoints to the successor or neighbor, (cases C2 and C3), both the benefit and the cost for the combined edge must be recomputed. On the other hand, if the original graph contained only one edge into the region of fusion from a given vertex, then only the cost must be updated for that edge after fusion (cases C1 and C4).

The procedure in Figure9, begins by visiting each successor of the collapsed vertex. Since no vertex is ever collapsed more than once, the total number of such successor visits is bounded by  $E$  over the algorithm. All operations in the visit take constant time, except for the calls to *FullWeightUpdate* and *FastCostUpdate*.

In both the cases labelled C2 and C3, the cost of the call to *FullWeightUpdate* can be charged to the edge from  $v$  to  $y$  that is deleted in these cases. Thus there are at most  $O(E)$  calls to this routine over the entire algorithm, which will be discussed in Section5.0.

On the other hand, the case labeled C4 is more complicated because the edge from  $x$  to  $y$  is not deleted but rather moved so that it now connects  $v$  to  $y$ . Thus, it is possible that the same edge will be visited many times as collapses are performed. Thus we must assume that this code is executed as many as  $O(EV)$  times over the entire algorithm. A similar analysis holds for the case labeled C1—each edge out of  $v$  is visited whenever another node is collapsed into  $v$ . so we must assume that this code is executed  $O(EV)$  times. All is not lost, however, if the *FastCostUpdate* executes in constant time.

FIGURE 9 Update successors.

```

procedure UpdateSuccessors( $v, x$ )
    // ( $v, x$ ) is the fusion edge
    for each  $y \in successors[v]$  do begin  $e := (v, y)$ ;
    C1: if  $y \notin successors[x]$  then FastCostUpdate ( $v, x, e$ );
    end
    // Make successors of  $x$  be successors of  $v$  and reweight
    for each  $y \in successors[x]$  do begin  $e := (x, y)$ ;
    C2: FullWeightUpdate ( $v, x, y, e$ ); // charge to deleted  $e$ 
        delete ( $x, y$ ) from  $edgeHeap$ ;
    end
    else if  $y \in neighbors[v]$  then begin
    C3:  $successors[v] = successors[v] \cup \{y\}$ ;
        FullWeightUpdate ( $v, x, y, e$ ); // charge to deleted  $e$ 
        delete ( $x, y$ ) from  $edgeHeap$ ;
        delete  $y$  from  $neighbors[v]$ ;
        delete  $v$  from  $neighbors[y]$ ;
    end
    else begin //  $y$  has no relationship to  $v$ 
    C4:  $successors[v] = successors[v] \cup \{y\}$ ;
  
```

```

    FastCostUpdate(v,x,e);
  end
  delete x from predecessors[y];
end
end UpdateSuccessors

```

To summarize, we have shown that the complexity of this procedure is  $O(EV + V^2)$ , not counting the time spent in the routines *FusionLegal*, called  $O(E)$  times, *ComputeCostBenefit*, called  $O(V)$  times, *FullWeightUpdate*, called  $O(E)$  times, and *FastCostUpdate*, called  $O(EV)$  times.

## 5.0 Dynamic Updates

Fusion cost and benefit are determined by the amount of data accessed within, and shared between, candidate loops. This section describes how the cost and benefit are measured and used in constrained fusion, as well as why dynamic updates are necessary for accurate consideration of the resource constraint.

### 5.1 Cost and Benefit of Fusion

For a single loop, we measure two sets of data accesses. The first set includes loop accesses, *i.e.*, it is the set of data accessed by the whole loop. The second measures the volume of data accessed by each iteration. In the following example, the *loop access set* is the  $i$ th column of array  $A$  plus one element of array  $B$ , and the *iteration access set* includes three elements,  $A(I,T)$ ,  $A(I+1,T)$ , and  $B(T)$ , parameterized by the index variable  $I$  and a loop invariant  $T$ .

```

DO I=1, N-1
  A(I,T)=A(I,T)*B(T)-A(I+1,T)
ENDDO

```

For a pair of loops, we measure the *benefit* of fusion by the size of the common data elements accessed by both loops, *i.e.*, the intersection of their loop access sets. This accurately represents the saving of loop fusion because the overlapped data do not need to be re-loaded once the loops are fused.

We measure the *cost* of fusing a pair of loops by the size of their aggregate data access set in each iteration, *i.e.*, the union of their iteration access sets. The union represents the minimal cache or register resources needed to fully buffer data from one iteration to another in the fused loop. The fused loop would overflow available cache space if the iteration access set of the fused loop includes data items that cannot all fit in cache.

We use a version of array-section analysis [13] to measure loop and iteration access sets. We omit the details of the analysis because constrained fusion does not depend on using any particular method. Other analysis can be used in this framework.

There is one final wrinkle to the algorithm we use for selecting edges to fuse. Rather than selecting the prime edge with the largest benefit at each step, our constrained fusion algorithm selects the edge with the greatest ratio of benefit to cost. The intent of this heuristic is to maximize the benefit gained for the resources used in any fusion group. It has

been shown to work well for bin packing [11] and in earlier work on memory hierarchy by Carr and Kennedy [6].

FIGURE 10 Dynamic updates after each loop fusion

```

procedure ComputeCostBenefit(v,x)
  // Node v and x are fused into v with alignment R
  newIA := v.LoopAccess ∪ x.LoopAccess;
  newIA := v.IterationAccess ∪R x.IterationAccess;
  dCv := || newIA || - || v.IterationAccess ||;
  dCx := || newIA || - || x.IterationAccess ||;
  v.IterationAccess := newIA;
end ComputeCostBenefit

procedure FullWeightUpdate(v, x, y, e)
  find the new alignment, R, between {v,x} and y;
  e.Benefit := || y.LoopAccessSet ∩ v.LoopAccessSet ||;
  e.Cost :=
    || y.IterationAccessSet ∪R v.IterationAccessSet ||;
  e.Weight := e.Benefit / e.Cost;
end FullWeightUpdate

procedure FastCostUpdate(v, x, e)
  if head(e) = v
    then e.Cost := e.Cost + dCv;
    else e.Cost := e.Cost + dCx;
  e.Weight := e.Benefit / e.Cost;
end FastCostUpdate

```

### 5.2 Dynamic Updates of Cost and Benefit

To carry out the fusion strategy described in the previous section, three quantities must be associated with each edge: the fusion *benefit*, the fusion *cost*, and the edge *weight*. As we saw in Section 4.0, whenever two loops are fused, updates are made in three routines (shown in Figure 10). First, after each fusion *ComputeCostBenefit* determines the total benefit and cost of the new fused loop. Then, during the process that updates the successors predecessors and neighbors, calls are issued to *FullWeightUpdate* if both the benefit and cost must be recomputed or to *FastCostUpdate* if only the cost needs to be updated.

Which routine to call is determined by whether the loop adjacent to the fused region had edges to each of the original vertices or an edge to only one of them. If it had edges to both, then it shares data with both and the impact of that sharing must be computed. However, if it had an edge into only one, the benefit of fusion remains the same as originally computed for the single edge—the other vertex adds nothing because it shares no data with the outside vertex.

In this second case, only the cost needs to be updated. But this can be done with a simple computation. Suppose we are fusing along the edge  $(v,x)$ . The revised cost for edge into the fused region is given by the following two formulas. If the edge  $e$  is incident on  $v$  in the original graph before fusion the change in cost is given by

$$dCv := Cost(\{v,x\}) - Cost(v)$$

If the edge was incident on  $x$ , the change is

$$dCx = Cost(\{v,x\}) - Cost(x)$$

In the code in Figure10, the costs are calculated by the size of loop and iteration access sets in *ComputeCostBenefit* and used in *FastCostUpdate*.

The accuracy of the updates to the benefit for a fusion depends on knowing the alignment that must be used in fusing the two loops. The operator “ $\cup_R$ ” is used to specify that the second loop is aligned with the first after shifting by offset  $R$ . Alignment is illustrated in the following example, where the union set of iteration access in the fused loop is  $\{A(I), A(J-1)\}$ . The size of the set depends on the alignment between  $I$  and  $J$  loops.

```
DO I
  A(I) = ...
ENDDO
DO J
  ... = A(J-1)
ENDDO
```

In multi-level loop fusion, the size of iteration access sets depends on the alignment of all enclosing loops. Furthermore, the size of loop access sets also depends on the alignment of all outer loops. Without alignment information, we may determine that two loops share data while they are in fact accessing different columns of the same array.

Given a program with  $A$  arrays, if the set operations in *ComputeCostBenefit* and *FullWeightUpdate* and the fusion test *FusionLegal* all take at most  $O(A)$ , the overall complexity of the constrained fusion algorithm with dynamic updates is  $O((E + V)(V+A))$ . To prove, we saw in Section4.0 that the algorithm had complexity  $O(EV + V^2)$ , not counting the cost of updates. We also saw that *ComputeCostBenefit*, which has a complexity of  $O(A)$ , is called  $O(V)$  times and *FusionLegal* and *FullWeightUpdate* are each called  $O(E)$  times at a cost of  $O(A)$ . Together, these calls contribute  $O(EA + VA)$  time. *FastCostUpdate* is called  $O(EV)$  times but takes only constant time, so it adds nothing extra to the complexity.

## 6.0 Evaluation

We have implemented constrained fusion as an extension to our previous work of unconstrained loop fusion [9]. It is built on a variant of the D System compiler from Rice University. The D compiler performs whole program compilation and uses a powerful value-numbering package to handle symbolic variables. It incorporates standard versions of loop and dependence analysis, and data flow analysis. Our previous work added sequential loop fusion along with supporting data transformations. For this work, we implemented the constrained fusion algorithm as described in Section4.0, and dynamic updates of cost and benefit of fusion as described in Section5.0. Constrained fusion inherits fusion-enabling transformations including loop embedding, loop alignment and iteration reordering. It can be applied at any loop level as part of the multi-level fusion algorithm (described in [9]). Code generation after fusion is currently implemented by mapping from the old iteration space to the fused iteration space. The final code is produced by the Omega library [22], which has been integrated

into the D System [1]. Our compiler does not perform register allocation, for which we rely on the machine’s back-end compiler.

The purpose of the evaluation is to demonstrate the benefit of constrained fusion over its unconstrained counterpart. In the absence of an embedded system, we use conventional workstations and treat their registers as local memory. We test different fusion schemes on two different machines and compilers. In addition, we use machine hardware counters to accurately measure the amount of register traffic.

We do not yet have a benchmark suit of embedded applications. In this experiment, we use four well-known scientific applications described in Table 1. All are benchmarks from the SPEC and NASA suites except for *ADI*, which we implemented ourselves as a kernel with separate loops processing boundary conditions. These programs present to a

TABLE 1. Applications tested

name	source	input	lines	loops	arrays
<i>Swim</i>	SPEC95	513x513	429	8	15
<i>Tomcatv</i>	SPEC95	513x513	221	18	7
<i>ADI</i>	self-written	2K x 1K	108	8	3
<i>SP</i>	NASA	class A&B	1141	218	42

fusion compiler challenges from real applications: loops have different dimensions and loop counts, and many loops are non-perfectly nested.

We measured the benefit of loop fusion in terms of register savings. We made this choice because registers are the most severely limited resource in the memory hierarchy. We focused on innermost loops, where register usage is determined, but constrained fusion can be applied at outer loop levels to optimize for specific cache size as well.

### 6.1 Effect on Register Traffic

We tested programs on two high-end microprocessors: a 336 MHz Sun Ultra II on an Enterprise 4500 server and a MIPS R12K processor at 300 MHz on an SGI Origin2000. The first is a widely used computing platform, and the second has hardware utility for us to measure exact amount of register traffic. Both processors have 32 integer and 32 floating-point registers. Three of the four tested programs had fewer than 15 loops and our compiler found no fused loop that exceeded the register resource. The fourth application, *SP*, contained nearly 500 loops after loop distribution<sup>1</sup>, about 200 of which were at the innermost level. The rest of this section considers constrained fusion on *SP*.

We tested four versions of *SP*, shown in Table2. *Original* was the original program. *OuterFused* was obtained by applying outer-loop fusion as in [9], which left 195 loops at the innermost level. The remaining two versions were

1. We use loop distribution before loop fusion because programmers may write loops with statements that share no physical data.



derived from *OuterFused: Unconstrained* fused innermost loops into about 18 loops, and *Constrained* generated about 145 loops so that no fused loop required more than 32 floating-point registers. The program reported performance by million flops per second (Mflop/s). The base performance of *Original* was obtained after enabling the full optimization in the machine compiler: *-O5* on Ultra II and *-mips4 -Ofast* on MIPS. All versions used Class B input, where major arrays were of size  $102^3$ .

On Ultra II, the outer-loop fusion improved whole-program performance by 20%. Unconstrained fusion among innermost loops, however, was not beneficial. In fact, *Unconstrained* version ran 13% slower than *Original* and 33% slower than *OuterFused*. In contrast, constrained fusion among innermost loops achieved the fastest speed on Ultra II--23% faster than *Original*, 3% than *OuterFused*, and 41% than *Unconstrained*. The resource estimation of 32 registers seemed accurate. We tested fusion with a resource constraint of 16 registers and the performance was half percent slower than *Constrained*, which assumed 32 registers. It should be mentioned that Sun's compiler ran out of memory and failed to optimize *Unconstrained* at *-O5*, so we used the *-O4*. For a fair comparison, we compiled *OuterFused* and *Constrained* at this lower optimization level. They still outperformed unconstrained fusion by 8% and 16% respectively. Therefore, on UltraII, constrained fusion was crucial in realizing the full benefit of locality optimization, and unconstrained fusion was in fact counter productive.

On MIPS R12K, we were able to measure register traffic (by the number of graduated register loads and stores) in

addition to performance. To isolate the effect of different types of loop fusion, we disabled SGI compiler's own prefetching (which generates non-binding register loads) and SGI's loop fusion and distribution except in the base version, where we enabled all optimization in the SGI compiler.

As shown in Table2, outer-loop fusion reduced register traffic by 30%. This was due to not loop fusion but its supporting transformation, *data regrouping* [9], which merged multiple arrays into a single array and reduced the demand for array base registers. When data regrouping was disabled, the total register traffic returned to within 3% of the original amount. The reduction from data regrouping was entirely on integer loads and stores. To improve register reuse among floating-point computation, we need loop fusion at the innermost level.

Unconstrained fusion caused considerable register spillage, increasing total register traffic by 15% over outer-loop fusion. Of all versions, constrained fusion achieved the minimal register traffic, which was 5.5% less than outer-loop fusion, 22% than unconstrained fusion, and 37% than no fusion. The reduction on register loads was even larger--6.7% less than outer-loop fusion, 25% than unconstrained fusion, and almost half (48%) than no fusion. The reduction should be much larger if we could observe floating-point registers separately. Overall, *Constrained* saved one of every two register loads and stores in this application benchmark.

TABLE 2. Register traffic and performance of SP with Class B input

versions of SP	336MHz Ultra II	300MHz MIPS R12K			
	Performance (Mflop/s)	Performance (Mflop/s)	loads (billion)	stores (billion)	total loads/stores
<i>Original</i>	39.7	64.7	3.31	1.40	4.71
<i>OuterFused</i>	47.8	99.2	2.39	1.24	3.63
<i>Unconstrained</i>	34.4	93.5	2.79	1.40	4.19
<i>Constrained</i>	48.7	98.8	2.24	1.20	3.44

Because of significant saving in register traffic, constrained fusion ran 6% faster than unconstrained fusion and 53% faster than no fusion. Interestingly, outer-loop fusion was a little faster than constrained fusion in spite of causing more register traffic. This was due to the limitation of our current compiler implementation. First, our compiler generated code with Omega library, which inserted 24 branch statements in innermost loops after fusion. These branches could all be removed by loop peeling if code generation was directly implemented in our compiler. In addition, we did not consider cache reuse, which would require strip-mining of innermost loops. We simulated the effect of strip-mining by using a smaller data input, Class A, where arrays were of size  $64^3$ . As expected, constrained fusion achieved the fastest speed of 102 Mflop/s, which was 5% faster than uncon-

strained fusion and 2% faster than outer-loop fusion. Although not reported in the table, we also applied constrained fusion on *OuterFused* assuming 16 registers or effectively unlimited registers. Neither performed as good as constrained fusion assuming the exact number of registers, indicating that the resource control of our fusion compiler was fairly accurate.

## 6.2 Effect on Fusion Graph Size

Table3 lists the statistics of all fusion graphs. *Swim* and *SP* have more than one loop nest after outer-level loop fusion. Therefore they generate multiple fusion graphs. Two pairs of loops in *SP* are listed together because they have the same statistics. The first five columns of Table3 show the program, the loop, the number of loop nodes, the number of

data-sharing edges, and the number of prime edges. A loop node is either an innermost loop or a non-loop statement. Column 4 shows the number of data sharing edges including dependences, read-read sharing edges, and their total. If two loops have a dependence and read-read sharing, the sharing is counted as a dependence. Column 4 and 5 shows that while the amount of data sharing is considerable in large programs, the number of prime edges is quite small. *ADI* has 12 nodes, 50 data-sharing edges, but only 12 prime edges; *SP* has a total of 195 nodes, 7707 data-sharing edges, but only 207 prime edges. In both programs, a loop shares data with a significant portion of the rest of the program. On average, each loop shares data with 4.5 loops or 45% of the program in *ADI* and with 40 loops or 21% of the program in *SP*. However, on average, each node is connected to no more than 1.1 prime edges. Column 4 also shows that all data-sharing pairs are data dependent except in the first loop

of *SP*, where read-read sharing accounts for 2% of all data sharing.

The frequency of dynamic updates is shown by the last two columns, which lists the number of loops fused and the number additions and deletions of prime edges during loop fusion. In the implementation, when two nodes are fused, all their adjacent edges are deleted, and only prime edges are added back. So the numbers of added and deleted edges are an over-estimate. The numbers show that the cost of dynamic updates is linear in the number of loop fusions. We can compute the average edge change for each loop fusion by dividing Column 6 with Column 5. On average, each loop fusion causes up to 3.0 edge additions and 3.6 edge deletions. Therefore, the average number of dynamic updates is no more than 3.0 for each fusion.

Although not shown in the table, the number of prime edges does not increase as loops are fused, neither does the maximal number of adjacent prime edges for each node.

**TABLE 3. Size of fusion graphs of loops after outer-level fusion**

program	loop	initial nodes	data sharing dep/read/total	prime edges dir/undir/total	fusion performed	edge change add/del/total
<i>Swim</i>	1	5	4/0/4	4/0/4	2	5/7/12
	2	3	3/0/3	3/0/3	2	6/5/11
<i>Tomcatv</i>	1	5	7/0/7	4/0/4	0	0/0/0
<i>ADI</i>	1	12	50/0/50	12/0/12	11	16/28/44
<i>SP</i>	1	155	7409/185/7594	156/20/176	31	84/112/196
	2, 4	9	21/0/21	6/0/6	0	0/0/0
	3, 5	8	28/0/28	7/0/7	7	10/17/27
	6	6	15/0/15	5/0/5	5	4/9/13

In summary, constrained fusion has shown to be efficient. On average, while each loop shares data with up to 45% of the other loops in test programs, it has no more than 1.1 prime edges in the fusion graph; moreover, each loop fusion incurs no more than 3.0 dynamic updates. For *NAS/SP* benchmark, constrained fusion has shown to improve performance over unconstrained fusion by 41% on an Ultra SparcII processor and by 6% on a MIPS R12K processor. It reduced register traffic by 22% on MIPS compared to unconstrained loop fusion. These results have demonstrated the significant advantage of constrained fusion over its unconstrained counterpart.

## 7.0 Related Work

Loop fusion for optimization of array and stream operations has a long history [10,12,14]. Combining loop distribution and fusion were originally discussed by Allen, Callahan, and Kennedy [2]. Fusion for reuse of vector registers was introduced by Allen and Kennedy [3]. The idea of fusion for register reuse using the dependence graph was further elaborated by Carr, Callahan, and Kennedy [5,6]. Kennedy and

McKinley [17] developed the original proof that weighted fusion is NP-complete with additive reweighting. Kennedy and McKinley [18] and Darte [7] considered unweighted fusion and established NP-completeness for a broad class of such problems. Ding and Kennedy [8] proved NP-completeness for a hypergraph problem with a reuse-based reweighting strategy that is not additive.

Gao et al. [10,23] and Kennedy and McKinley [17] use partitioning-based methods that repeatedly cut nodes of a weighted fusion graph into sub-graphs with minimal-weight inter-partition edges. Singhai and McKinley addressed a subset of the fusion problem, where dependence edges form a tree [24]. Megiddo and Sarkar explored optimal fusion by integer programming [21]. In all these methods, addition is built in as the reweighting operation. Furthermore, none of these methods considers resource constraints, although integer programming could be extended to handle them.

Wang, Tempe, and Pande[26] present an algorithm that uses a variant of greedy weighted fusion similar but not identical to the one used in Kennedy's  $O(V(E+V))$  algorithm described in Section 3.0. Their method iteratively chooses a pair of loops with the most sharing per memory unit to fuse,

then tries to fuse as many other loops with the resulting group as correctness, resources and profitability permit. Their complexity is at least  $O(V^3)$ . We note that the exclusive consideration of prime edges in our method also yields a heuristic that is not the same as greedy weighted fusion.

Although most of the methods described above are global, our experience with actual compiler outputs indicates that global fusion has not found its way into commercial practice. Most compilers examine loops in sequential order and do not fuse two loops without visiting and optimizing all other loops or statements in between. This drawback is shared by the implementations of Allen and Kennedy for vector registers [3], McKinley et al. for cache hierarchy [20], Manjikian and Abdelrahman for reuse and parallelism [19], and Ding and Kennedy for fusing loops of different shapes [9]. Unlike these, we select fusion candidates globally and consider resource constraints.

Carr and Kennedy address the problem of register reuse using scalar replacement and unroll-and-jam [6]. Their goal is to increase reuse in a single loop nest, which is orthogonal to our goal of multi-loop register reuse.

## 8.0 Conclusion

Loop fusion changes the computation and data access order of the whole program and therefore has important application in areas such as locality optimization, DSP compilation, vectorization, and Grid computing. Although fusing loops can lead to better reuse of hardware resources, it can also be counter productive if excessive fusion overflows available resources. This paper presents two algorithms for constrained weighted fusion that fuse along most beneficial edges but maintain a bounded resource consumption in fused loops. The first algorithm extends Kennedy's fast greedy fusion, and the second uses a new concept called prime edges. Both methods dynamically update the resource requirement of loops during the fusion process. Their time complexity is of  $O((E+V)(V+A))$ , where  $E$  is the number of dependence edges,  $V$  is the number of loops, and  $A$  is the number of arrays in the program.

The evaluation of prime-edge based constrained fusion on standard benchmarks has shown significant reduction in both the size of fusion graphs and the amount of register traffic. The size of fusion graph was linear to the size of the program, so was the cost of dynamic updates, despite quadratic data-sharing relationships among loops. On the largest application benchmark tested, constrained fusion reduced total register traffic by 22% and improved performance by up to 41%, compared to unconstrained loop fusion.

## 9.0 References

[1] V. Adve and J. Mellor-Crummey. Using Integer Sets for Data Parallel Program Analysis and Optimization. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, Montreal Canada, June 1998.

[2] R. Allen, D. Callahan and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conf. Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987.

[3] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers* 41(10):1290-1317, October 1992.

[4] R. Allen and K. Kennedy. *Advanced Compilation for High Performance Computers*. Morgan Kaufman, to be published.

[5] S. Carr, D. Callahan and K. Kennedy. Improving register for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, White Plains NY, June 1990.

[6] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems* 16(6):1768-1810, 1994.

[7] A. Darte. On the complexity of loop fusion. In *Proceedings of PACT '99*, Newport Beach, CA, October 1999.

[8] C. Ding and K. Kennedy. Memory bandwidth bottleneck and its amelioration by a compiler. In *Proceedings of IPDPS'00*, Cancun, Mexico, May 2000.

[9] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings of IPDPS'01*, San Francisco, CA, April 2000, to appear.

[10] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of LCPC'92*, New Haven, CT, August 1992.

[11] M. Garey and D. Johnson. *Computers and Intractability, A the Theory of NP-Completeness*. W. H. Freeman, NY, 1979.

[12] A. Goldberg and R. Paige. Stream processing. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 228-234, August 1984.

[13] P. Halvak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *ACM Transactions on Programming Languages and Systems* 2(3):350-360, July 1991.

[14] D.-C. Ju, C.-L. Wu, and P. Carini. The classification, parallelization of array language primitives. *IEEE Transactions on Parallel and Distributed Systems* 5(10):1113-1120, October 1994.

[15] K. Kennedy. Fast greedy weighted fusion. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[16] K. Kennedy. Fast greedy weighted fusion. To appear in *Journal of Parallel and Distributed Computing*.

[17] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of LCPC'93*, (U. Banerjee, et. al., editors), Lecture Notes in Computer Science 768, 301-320, Springer-Verlag, Berlin, 1993.

[18] K. Kennedy and K. McKinley. Typed fusion with applications to parallel and sequential code genera-

tion. CRPC-TR94646, Center for Research on Parallel Computation, Rice University, 1994.

- [19] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems* 8, 1997.
- [20] K. McKinley, S. Carr and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems* 18(4):424-453, July, 1996.
- [21] N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, 1997.
- [22] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM* 35(8):102-114, August 1992.
- [23] V. Sarkar and G. Gao. Optimization of array accesses by loop transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.
- [24] S. Singhai and K. McKinley. Loop fusion for data locality and parallelism. In *Proceedings of MAS-PLAS'96*, New Paltz, NY, April 1996.
- [25] W. Tembe and S. Pande. Optimizing on-chip memory usage through loop restructuring for embedded processors. To appear in *IEEE Transactions on Computers*.
- [26] L. Wang, W. Tembe and S. Pande. A Framework for loop distribution on limited memory processors. In *Proceedings of CC'2000*, Berlin, Germany, May 2000.