Fast Greedy Weighted Fusion

Ken Kennedy

CRPC-TR99789
December 1999

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted December 1999

Fast Greedy Weighted Fusion

Ken Kennedy

ken@rice.edu

Center for High Performance Software

Rice University

Abstract

Loop fusion is important to optimizing compilers because it is an important tool in managing the memory hierarchy. By fusing loops that use the same data elements, we can of the ensure that uses after the first can take data from cache or registers rather than memory, avoiding costly cache misses. Unfortunately the problem of optimal loop fusion for reuse has been shown to be NP-hard, so compilers must resort to heuristics to avoid unreasonably long compile times. Greedy strategies are often excellent heuristics, and can be used to produce high-quality solutions in reasonable running times. We present an algorithm for *greedy weighted fusion,* in which the heaviest edge (the one with the most reuse) is selected for possible fusion on each step. The algorithm is shown to be fast in the sense that it takes $O(VE + V^2)$ time, which is arguably optimal for this problem.

1.0 Introduction

Because processor speed has been increasing far more rapidly than memory speed, compilers have become increasingly concerned with the management of memory hierarchies, in an attempt to increase the frequency with which data values are reused from cache or registers, thus reducing the number of costly cache misses. Loop fusion is an important tool for increasing reuse because it can bring uses of the same data element closer together. Fusion is particularly important in an

array language like Fortran 90 because the typical compiler strategy is to "scalarize" array statements by converting them to simple one-statement loops. For example, the two statements:

```
A(1:1000) = A(1:1000) + C
B(1:1000) = B(1:1000) + A(1:1000)
```

would be converted to:

```
DO I = 1, 1000
     A(I) = A(I) + C
ENDDO
DO I = 1, 1000
     B(I) = B(I) + A(I)
ENDDO
```

It is easy to see that if these two loops are fused, a load of array A could be saved on each iteration of the second loop. However, in a complete program there may be many loops that can be fused and many statements that cannot be fused with any loop. Often the choices of which loops to fuse are not clear as fusing one pair of loops may make it impossible to fuse others. Rather than carrying out loop fusion by trial-and-error, which could be expensive in compile time, we need a way to select groups of loops that are both legal and profitable to fuse. Ideally this selection process should maximize the savings in accesses to memory. This is the job of a global fusion algorithm—it examines the entire program and determines collections of loops that should be fused. Since the size of programs can be large, such an algorithm should be reasonably efficient if it is to be included in a compiler.

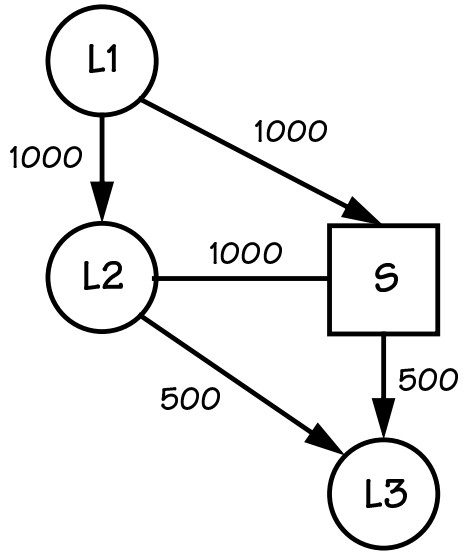
To develop such an algorithm, we will model the problem as a graph in which vertices represent loops and edges represent dependences between the loops, where there exists a dependence between two loops if there is a dependence between the statements in the bodies of those two loops. In addition, each edge will have a different weight reflecting the amount of reuse to be gained through fusing the loops at its endpoints. For example, two different fusion choices might have different iteration ranges:

```
L1:     DO I = 1, 1000
           A(I) = B(I) + X(I)
        ENDDO

L2:     DO I = 1, 1000
           C(I) = A(I) + Y(I)
        ENDDO

S:      Z = FOO(A(1:1000))
```

```
L3      DO I = 1, 500
           A(I) = C(I) + Z
        ENDDO
```

In this example, statement S, which cannot be fused with any loop, must come after loop L1 but before L3 because there is a true dependence from loop L1 to statement S and a true dependence from S to L3. Thus, L1 and L3 cannot be fused. Similarly, loop L2 must come between loops L1 and L3 because of similar dependence patterns. However, since there is only an input dependence between L2 and S, they may occur in any order. Therefore, loop L2 can fuse with either L1 (if it stays before S) or L3 if it is moved after S. The dependence pattern for this loop is shown in Figure 1 below. Note that an undirected edge represents an input dependence.

FIGURE 1 Weighted dependences for the fusion example.



The diagram in Figure 1 makes it clear that L2 may not fuse with both L1 and L3 because that would introduce a cycle in the dependence graph through statement S, so a choice must be made about which fusion to select. Here the choice favors fusing the loops with the largest iteration ranges: L1 and L2.

A slight variation on this example illustrates another way that different fusion opportunities may differ in value.

```
L1:     DO I = 1, 1000
           A(I) = B(I) + X(I)
        ENDDO

L2:     DO I = 1, 1000
           C(I) = A(I) + Y(I)
        ENDDO

S:      Z = FOO(A(1:1000))

L3      DO I = 1, 1000
           A(I) = C(I+500) + Z
        ENDDO
```

In this case the loop iteration ranges are the same, but when L2 is aligned with L3, there are only 500 iterations on which there is reuse.

In order to construct an algorithm to address the weighted fusion problem, we must assign to each loop-crossing dependence *d* in the program a *weight W(d)* that measures the *benefit* to be gained if its source and sink were located in the same iteration of the same loop. In other words, if the loops containing the source and sink were to be fused, *W(d)* memory operations (loads and stores) would be saved over the version of the program where the endpoints remain in different loops. The numeric labels on edges in Figure 1 represent the weights.

2.0 The Weighted Fusion Problem

Now assume that we have a collection of weighted dependence edges and a collection of fusible loops, along with a collection of statements that cannot be fused with any loop. In the discussion to follow, we will refer to these non-fusible statements as *bad vertices*. In addition, assume that we may have certain edges, called *bad edges*, along which fusion is not possible. For example, in previous discussions we have seen the existence of fusion-preventing dependences. Although such dependences are less important in the cases discussed in this chapter, we will nevertheless include them in the problem model.

Now we are ready to map loop fusion onto an abstract problem. First we begin with the definition of a special kind of graph that contains both directed and undirected edges. In our problem, the undirected edges will be used to model input dependences.

Definition 1. A *mixed-directed graph* is defined to be 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, $E_d \cap E_u = \varnothing$.

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 collection of *bad vertices* $B \subset V$ that cannot be fused with any other vertex in V, and a collection of *bad edges* $E_b \subset E_d$ that prevent fusion of their endpoints, the *weighted loop fusion problem* is the problem of finding a collection of vertex sets $\{V_1, V_2, ...,V_n\}$ such that

a. $\bigcup_{i=1}^{n} V_i = V$, i.e., the collection covers all vertices in the graph,

b. $\forall i, j, 1 \le i,j \le n, (V_i \cap V_j) \ne \varnothing \rightarrow i = j$,, i.e. the vertex sets form a partition of V.

c. $\forall i, 1 \le i \le n, ((V_i \cap B = V_i) \vee (V_i \cap B = \varnothing))$, i.e. each set contains either no bad vertices or exclusively bad vertices.

d. For any i, there is no directed path (all edges in E_d) between two vertices in V_i that passes through a vertex not in V_i.

e. For any i, there is no bad edge between two vertices in V_i.

f. The total weight of edges between vertices in the same vertex set, summed over all vertex sets is maximized.

Note that condition (c) implies that we never fuse bad vertices.

3.0 Greedy Weighted Fusion

Finding the optimal solution to this problem has been shown to be NP-hard [14], so we will need resort to heuristic approaches. The heuristic we present here is designed to be relatively simple while displaying modest computational complexity. It employs a greedy strategy that iteratively selects the edge of highest weight and fuses the endpoints of that edge, along with all edges on a path between them into the same vertex set.

Because the implementation of this algorithm will require sophisticated data structures, we will begin with an outline 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 $O(V+E)$ 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. This can be done in $O(V^2)$ time.
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 $O(E \lg E)$ time. Note that since $\lg E \le \lg V^2 = 2\lg V$, the complexity of this stage can be rewritten as $O(E \lg V)$
6. While *edgeHeap* is non-empty, select and remove the heaviest edge (v,w) from it.
   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 collapse, recompute *successor*, *predecessor* and *neighbor* sets for the composite vertex and recompute weights between the composite vertex and other vertices as appropriate.

A more detailed version of the algorithm is given in Figure 2. Note that phases 1 through 5 can be implemented in $O(EV + V^2 + E\lg V) = O(EV + V^2)$ time. It would be good if the total time for phase 6 could be limited to this asymptotic bound as well. To understand whether this is possible, we need to examine the steps in these phases more carefully.

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 reached by a path from v;
    // badPathFrom[v] contains the set of 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 := ∅;
P2: InitializePathInfo(V, edgeHeap);

L1: while edgeHeap ≠ ∅ do begin
        select and remove the heaviest edge e = (v,w) from edgeHeap;
        if v ∈ pathFrom[w] then swap v and w;
        if w ∈ badPathFrom[v] then
            continue L2; // cannot or need not be fused

        // Otherwise fuse v, w, and all vertices on a path between them
P3:     worklist := ∅;
L2:     for each x ∈ successors[v] do
            if w ∈ pathFrom[x] then worklist := worklist ∪ {x};
        if worklist = ∅ then add w to worklist; // (v,w) undirected

L3:     while worklist ≠ ∅ do begin
            select and remove the first vertex x from worklist;
            if x ≠ w then
                for each y ∈ successors[x] do begin
                    if w ∈ pathFrom[y] and y ∉ worklist.ever then begin
                        worklist := worklist ∪ {y};
                    end
                end

            // fuse x into v;
            rep[x] = v;

            // update pathFrom and pathTo sets
            UpdatePathInfo(v,x);
```

```
            // update the graph representation
            UpdateSuccessors(v,x);
            UpdatePredecessors(v,x);
            UpdateNeighbors(v,x);

            // delete vertex x
            delete x, predecessors[x], successors[x], neighbors[x],
                pathFrom[x], badPathFrom[x], pathTo[x],
                and badPathTo[x];
            delete x from successors[v];
        end L3
    end L1
end WeightedFusion;
```

Code for the initialization routines is given in Figure 3 and in Figure 4. Note that we are computing not only *pathFrom* and *badPathFrom* sets but also the inverse sets *pathTo* and *badPathTo*, where $x \in pathTo[y]$ if and only if $y \in pathFrom[x]$. These sets are needed for the update operations that will be performed after collapses.

FIGURE 3 Initialize *predecessors, successors* and *neighbors.*

```
procedure InitializeGraph(V, E_d, E_u);

    for each v ∈ V do begin
        successors[v] := ∅;
        predecessors[v] := ∅;
        neighbors[v] := ∅;
    end

    for each (x,y) ∈ E_d do
        successors[x] := successors[x] ∪ {y};
        predecessors[y] := predecessors[y] ∪ {x}

    for each (x,y) ∈ E_u do begin
        neighbors[x] := neighbors[x] ∪ {y};
        neighbors[y] := neighbors[y] ∪ {x};
    end
end InitializeGraph;
```

FIGURE 4 Initialize *path* sets.

```
procedure InitializePathInfo(V, edgeHeap);

    // v is the vertex into which merging is taking place
    // x is the vertex currently being merged.

L1: for each v∈V in reverse topological order do begin
        rep[v] := v;
        pathFrom[v] := {v};
        if v∈B then badPathFrom[v] = {v} else badPathFrom[v] := ∅;
        for each w ∈ successors[v] do begin
            pathFrom[v] := pathFrom[v] ∪ pathFrom[w];
            badPathFrom[v] := badPathFrom[v] ∪ badPathFrom[w];
            if (v,w) is a bad edge or w ∈ B then
                badPathFrom[v] := badPathFrom[v] ∪ pathFrom[w];
            add (v,w) to edgeHeap;
        end
        for each w ∈ neighbors[v] do
            if w ∈ pathFrom[v] then begin
                delete w from neighbors[v];
                delete v from neighbors[w];
                successors[v] := successors[v] ∪ {w};
            end
            add (v,w) to edgeHeap;
    end
    invert pathFrom to compute pathTo;
    invert badPathFrom to computer badPathTo;
end InitializePathInfo;
```

Once it has selected an edge (v,w) along which to collapse, the greedy weighted fusion algorithm will perform the following steps:

1. Create a queue named *worklist* and initialize it to contain all the successors of v that reach w. If none of the successors of v reach w, then there is no path from v to w, so put w on *worklist*.
2. While *worklist* is not empty
   a. Select an remove the oldest element x of *worklist*;
   b. Add to *worklist* all elements $y \in successors[x]$ such that $w \in pathFrom[y]$ and y has never been an element of *worklist*;
   c. Merge x into v;
   d. Reconstruct *pathFrom* sets for the entire graph. Note that if some vertex u reaches x and v reaches another vertex z, then, after merging x into v, u reaches z..

e. Create a new successor, predecessor, and neighbor lists for the composite v by merging $successors[x]$, $predecessors[x]$, and $neighbors[x]$ into $successors[v]$, $predecessors[v]$. and $neighbors[v]$ as appropriate;

3.1 A Fast Set Implementation

A naive implementation of the steps above might use a list and associated bit vector for all sets. Then, step 1 above would require O(V) operations because the bit vector would need to be initialized to empty.

The cost of step 1 can be reduced to a constant amount of time per edge by using a well-known but rarely-discussed strategy for avoiding the initialization costs for a set represented by an indexed array. The trick is to use two integer arrays of length V for the set representation. The array Q will contain all the vertices in the queue in order from element $Q[next]$ to element $Q[last]$. The array In will serve as the bit array for element testing—$In[v]$ will contain the location in Q of vertex v.

To enter a new vertex y at the end of the queue requires the following steps:

$$last = last + 1;\ Q[last] = y;\ In[y] = last;$$

Note that this leaves the element $In[y]$ pointing to a location in Q that is within the range $[0{:}last]$ and contains y; Thus the test whether a vertex z is a member of the Queue is:

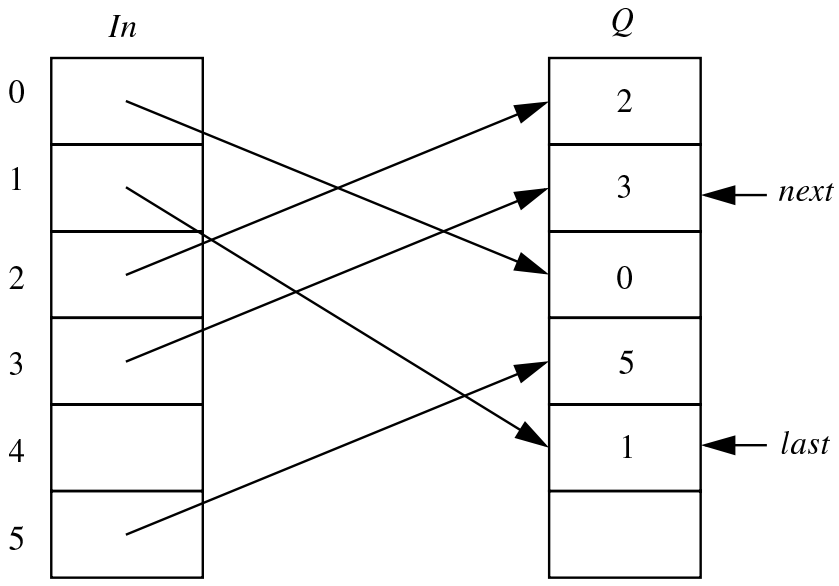
$$next \le In[z] \le last \textbf{ and } Q[In[z]] = z;$$

and the test whether z has ever been a member of Q is:

$$0 \le In[z] \le last \textbf{ and } Q[In[z]] = z;$$

Figure Chapter 5 gives a graphical description of the queue representation. Note that vertex 4 cannot be mistaken for a member because no element of Q in the right range points back to 4.

FIGURE 5 Fast queue representation



Using this representation we can perform all queue operations in constant time including initialization. This data structure will be used for many of the set representations in the algorithm.

3.2 Elaboration of Algorithm Steps

Returning to our analysis of the overall algorithm, we see that Step 2a could be done in constant time by simple list operations. Steps 2b through 2e are involved with a merger and require some analysis.

Step 2b. Suppose we charge the cost of traversing the edge (x,y) to the vertex y if $y \in pathFrom[v]$ Since each vertex can be merged into another at most once, the total cost of such edge traversals is $O(V)$ over the entire program. Next consider those vertices y that are successors of some node in the collapsed region, but are not themselves in that region. Such vertices become successors of composite region represented by v. If we charge the cost of traversing edges out of x to such an outside vertex y to the edge (x,y), an edge can accumulate $O(V)$ charges over the course of the algorithm. Thus the total cost of Step 2b over the course of the algorithm is $O(EV)$.

Step 2c is performed at most O(V) times so the tricky parts of the analysis arise from the update operations in steps 2d and 2e. We will treat these steps separately. For the purposes of this discussion, we will consider the these steps in reverse order.

3.2.1 Updating Successors and Neighbors

Step 2e. First, let us consider the cost of updating successors. The code for this operation is given in Figure 6

FIGURE 6 Update *successors*.

```
procedure UpdateSuccessors(v,x);

    // v is the vertex into which merging is taking place
    // x is the vertex currently being merged.

    // Make successors of x be successors of v and reweight
    for each y ∈ successors[x] do begin
        if y ∈ successors[v] then begin
            W(v,y) = W(v,y) + W(x,y) // charge to deleted edge (x,y)
            reheap edgeHeap;
            delete (x,y) from edgeHeap and reheap;
        end
        else if y ∈ neighbors[v] then begin
            successors[v] = successors[v] ∪ {y};
            W(v,y) = W(v,y) + W(x,y); // charge to deleted edge (x,y)
            reheap edgeHeap;
            delete (x,y) from edgeHeap and reheap;
            delete y from neighbors[v];
            delete x from neighbors[y];
        end
        else begin // y has no relationship to v
            successors[v] = successors[v] ∪ {y};
            replace (x,y) with (v,y) in edgeHeap; // no charge
        end
        delete x from predecessors[y];
    end
end UpdateSuccessors;
```

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. For each such vertex,

the procedure visits each successor. Since no vertex is ever collapses more than once, the total number of successor visits is bounded by E. All of the operations in the visit take constant time, except for the reheap operation on *edgeHeap*, which takes $\lg E = \lg V$ time.

One wrinkle that must be dealt with is the case where the edge from x to y is not deleted but rather moved so that it now connects v to y. This is the third case in the if statement within *UpdateSuccessors*. If we are not careful, it is possible that the same edge will be visited many times as collapses are performed. However, note that the third case does no reweighting and no reheaping is required to deal with it. Therefore the total cost contributed by the third case in the if statement is at most. $O(EV)$. The cost of operations in the other two cases can be charged to the edge being deleted, so the total cost is bounded by $O(E \lg V)$.

Thus the cost associated with all invocations of *UpdateSuccessors* is $O(EV + E \lg V + V) = O(EV)$. Similar analyses for *UpdatePredecessors* and *UpdateNeighbors*, given in figures Figure 7 and Figure 8, yield the same time bound.

FIGURE 7 Update *predecessors*.

```
procedure UpdatePredecessors(v,x);

    // v is the vertex into which merging is taking place
    // x is the vertex currently being merged.

    // Make predecessors of x be predecessors of v and reweight
    for each y ∈ predecessors[x] do begin
        if y ∈ predecessors[v] then begin
            W(y,v) = W(y,v) + W(y,x)
            reheap edgeHeap // charge to deleted edge (y,x)
            delete (y,x) from edgeHeap and reheap;
        end
        else if y ∈ neighbors[v] then begin
            predecessors[v] = predecessors[v] ∪ {y};
            W(v,y) = W(v,y) + W(x,y);
            reheap edgeHeap; // charge to deleted edge
            delete (y,x) from edgeHeap and reheap;
            delete y from neighbors[v];
        end
        else begin
            predecessors[v] = predecessors[v] ∪ {y};
```

```
            replace (y,x) with (y,v) in edgeHeap; // no charge
        end
        delete x from successors[y];
    end
end UpdatePredecessors;
```

FIGURE 8 Update *neighbors*.

```
procedure UpdateNeighbors(v,x);

    // v is the vertex into which merging is taking place
    // x is the vertex currently being merged.

    // Make neighbors of x be succs, preds, or nbrs of v, as appropriate;
    for each y ∈ neighbors[x] do begin
        if y ∈ pathFrom[v] then begin // make y a successor of v
            if y ∈ successors[v] then begin
                W(v,y) := W(v,y) + W(x,y);
                reheap edgeHeap; // charge to deleted edge;
                delete (x,y) from edgeHeap and reheap;
            else begin
                W(v,y) := W(x,y);
                replace (x,y) with (v,y) in edgeHeap; // no charge
                successors[v] = successors[v] ∪ {y};
            end
        else if v ∈ pathFrom[y] then begin // make y a pred of v
            if y ∈ predecessors[v] then begin
                W(y,v) := W(y,v) + W(y,x);
                reheap edgeHeap // charge to deleted edge;
                delete (x,y) from edgeHeap and reheap;
            end
            else begin
                W(y,v) := W(y,x);
                replace (x,y) with (y,v) in edgeHeap;
                predecessors[v] = predecessors[v] ∪ {y};
            end
        end
        else if y ∈ neighbors[v] then begin // make y a nbr of v
            W(v,y) := W(v,y) + W(x,y);
            reheap edgeHeap // charge to deleted edge
            delete (x,y) from edgeHeap and reheap;
        end
```

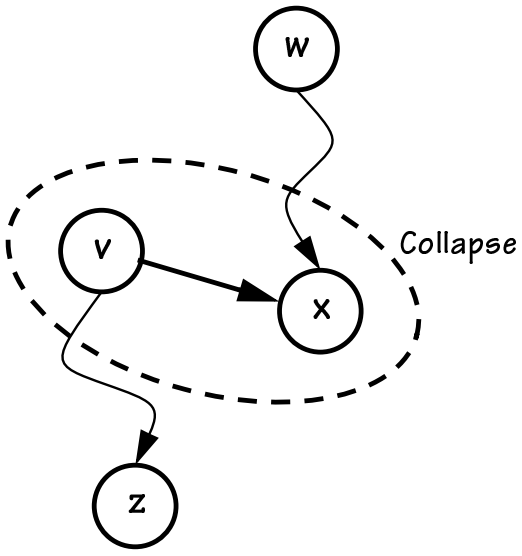
```
            else begin // no relationship—make them neighbors
                neighbors[v] = neighbors[v] ∪ {y};
                replace (x,y) with (v,y) in edgeHeap; // no charge
            end
            delete x from neighbors[y];
        end
    end UpdateNeighbors;
```

3.2.2 Updating Path Information

Step 2d. We now turn to the problem of updating the *pathFrom, badPathFrom, pathTo,* and *badPathTo* sets after a collapse. This step must be performed before the *successor*, *predecessor* and *neighbor* sets are updated because it uses the old relationships to perform the update.

The key difficulty with this step is that any vertex that reaches the vertex x that is being collapsed into v, now transitively reaches any vertex reached by v. This problem is illustrated by the diagram in Figure 9.

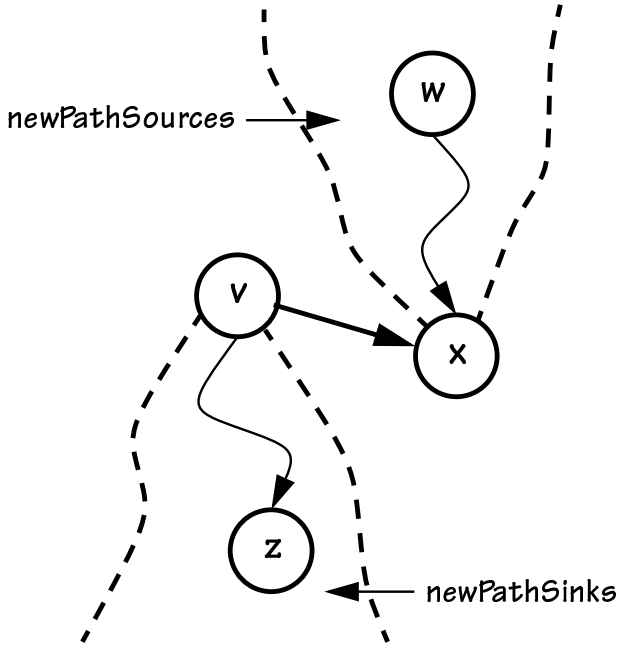
FIGURE 9 Illustration of the path update problem.



If we were to recompute the *pathFrom* sets from scratch after each collapse, it would take a total of $O(EV^2 + V^3)$ time, since a single application of the *pathFrom* computation takes $O(EV + V^2)$ time.

To reduce this time bound we must ensure that we do no more work than is required. The basic strategy is to compute *newPathSinks*, the set of vertices reached from *v* but not from *x*, and *newPathSources*, the set of vertices that reach *x* but not *v*.

FIGURE 10 Solving the path update problem.



Once we have these sets, we will update the *pathFrom* sets for every vertex in *newPathSources* and we will update the *pathTo* set of every vertex in *newPathSinks*.

One way to do this would be the following:

```
for each b ∈ newPathSources do
    pathFrom[b] := pathFrom[b] ∪ newPathSinks;
for each b ∈ newPathSinks do
    pathTo[b] := pathTo[b] ∪ newPathSources;
```

The problem with this approach is that it might take $O(V^2)$ time because there could be $O(V)$ vertices in *newPathSinks* $\cup$ *newPathSources* and the set operations each take $O(V)$ time. Thus the total time would be bounded by $O(V^3)$, making it more expensive than any other part of the algorithm.

One reason the cost is high is the possibility that we are doing unnecessary work. Since we only increase the size of the *pathTo* and *pathFrom* sets, if we use bit matrices of size V^2 to represent these sets and we only turn on bits once, we will limit the total amount of work to $O(V^2)$. However, in the strategy above, we may attempt to turn on the same bit many times. This is because vertices in *newPathSources* might already reach vertices in *newPathSinks* if there is an edge between the two sets that bypasses *v* and *x*.

To avoid such redundant work we will back up through the vertices in *newPathSources* taking care not to turn on any *pathTo* bit more than once. Similarly, we will move down through *newPathSinks* to turn on *pathFrom* bits at most once. In the procedure *UpdatePathInfo*, given in Figure 11, these processing passes are encapsulated in calls to *UpdateBothWays*, which calls *UpdateSlice*, given in Figure 12, once for each direction—upward from its second parameter *x* and downward from its first parameter *v*. Note that there are at most eight calls to *UpdateSlice* for each of the $O(V)$ collapses. To achieve the desired time bound, we must bound the total amount of work in each these calls by $O(EV + V^2)$.

FIGURE 11 Update *pathFrom* and *pathTo* sets.

```
procedure UpdatePathInfo(v,x);

    // First, check for the case where (v,x) is an undirected edge
    //   and neither vertex reaches the other.
    //   In this case, update pathFrom for variables that reach v but not x
    //   and pathTo for vertices that are reached from x but not v.

    if x ∉ pathFrom[v] then begin // special case: undirected (v,x)

        UpdateBothWays(x,v,pathFrom, pathTo);
        UpdateBothWays(x,v,badPathFrom, badPathTo);
    end

    //   Now update pathFrom for vertices that reach x but not v,
    //        and pathTo for vertices reached from v but not from x
```

```
    UpdateBothWays(v,x,pathFrom, pathTo);
    UpdateBothWays(v,x,badPathFrom, badPathTo);
end UpdatePathInfo

procedure UpdateBothWays(v,x,pathFrom, pathTo);
    // Traverse upward from x and downward from v
    newPathSinks := pathFrom[v] – pathFrom[x];
    newPathSources := pathTo[x] – pathTo[v];
    pathTo[v] := pathTo[v] ∪ newPathSources;
    UpdateSlice(x, pathFrom, newPathSources, newPathSinks,
            successors, predecessors);
    UpdateSlice(v, pathTo, newPathSinks, newPathSoures,
            predecessors, succcessors);
end UpdateBothWays
```

We now examine the work done in a single call to *UpdateSlice.* In each call, the parameter *pathFrom* is meant to represent the *pathFrom* or *pathTo* sets, depending on the direction of the traversal. (Note that a second set of calls passes *badPathFrom* and *badPathTo* to this parameter.)

Since the effect of the calls in different directions is symmetric, we will analyze the cost of a single call to update all the *pathFrom* sets in *newPathSources.* For each vertex w in the slice, represented by *newPathSources*, we will compute a set *newPathFrom*[w] that represents the set of new vertices reached from w after the collapse. As each vertex is processed, we will visit all its predecessors. At each predecessor y, we will examine each vertex z in *newPathFrom*[w] to see if it is in *pathFrom*[y]. If z is not in *pathFrom*[y], we will add it to *pathFrom*[y] and to *newPathFrom*[y]. The algorithm is given in Figure 12.

FIGURE 12 Update *pathFrom* sets for a slice of vertices reaching or reached from x.

```
procedure UpdateSlice(x, pathFrom, newPathSources, newPathSinks,
        cesors, pcesors);
    // x is the vertex that is being collapsed
    // pathFrom is the set that is being updated
    // newPathSources is the set of vertices that can reach x
    //        but not the vertex it is being collapsed into
    // newPathSinks is the set of vertices reachable from the vertex being
    //        collapsed into but not from x
```

```
    // cesors is the successor set (if traversing backward from x)
    // pcesors is the predecessor set (if traversing backward from x)

    // Update pathFrom sets backing up from x in newPathSources
    //   adding vertices in newPathSinks

S0: newPathFrom[x] := newPathSinks;
L1: for each b ∈ newPathSources– {x} do newPathFrom[b] := ∅;
    worklist := {x};
L2: while worklist ≠ ∅ do begin
        pick an element w from the front of worklist and remove it;
L3:     for each y ∈ pcesors[w] such that y ∈ newPathSources do begin
            if y ∉ worklist.ever then add y to worklist;
L4:         for each z ∈ newPathFrom[w] do
S1:             if z ∉ pathFrom[y] then begin
                    pathFrom[y] := pathFrom[y] ∪ {z};
                    newPathFrom[y] := newPathFrom[y] ∪ {z};
                end
        end
    end
end UpdateSlice;
```

We establish the correctness of procedure *UpdateSlice* by induction on the order in which elements are removed from worklist. Since the first element off the worklist is x, we are certain that its *newPathFrom* set is correctly computed because it must be the original *newPathSinks* by the way we have computed it. Now assume that *pathFrom*[b] and *newPathFrom*[b] are computed correctly for every vertex that comes off the worklist before vertex w. Furthermore, assume that vertex w has an incorrect *pathFrom* set when it comes off the worklist. This can only be because *pathFrom*[w] has some bit off that should be on. This bit must correspond to some vertex in *newPathSinks*. But this means that none of the successors of w have that bit turned on in either their *pathFrom* or *newPathFrom* sets, but since there must be a path from w to the vertex represented by that bit, then some predecessor, which necessarily came off the worklist before w, must have the bit for that vertex set incorrectly as well, a contradiction.

To show that the algorithm stays within the desired time limits, we must consider several aspects of the implementation. If we use the indexed set representation described earlier, we can initialize each *newPathFrom* set and the *worklist* to the empty set in constant time. Since the procedure is entered at most O(V) times and there are at most O(V) vertices in a slice each of which is put on

the worklist at most once, the body of loop L2 is executed at most O(V^2) times. Loop L3 visits each predecessor of a vertex, so its body should be executed at most O(EV) times.

Unfortunately the work done in the body of L3 includes the loop L4. We must have some way of bounding the work done by the body of loop L4 which consists of a single if statement S1. The true branch of the if statement is taken only if the collapse has made *y* reach *w* and this is the first time that fact is noticed. We charge the constant time of this branch to the entry in the *pathFrom* matrix that is being set to *true*. Thus the true branch of the if statement can be executed only O(V^2) times over the entire algorithm.

This leaves the false branch, which is empty. however, the test itself represents a constant time cost for the false branch. How many times can the test be executed with the false outcome? Since *newPathFrom*[*w*] is true, it means that the immediate predecessor *w* of *y* within the *newPathFrom* set, has just had its *pathFrom* bit for *z* set to true. Thus we will charge this test to the edge between *y* and *w*. Since a given *pathFrom* bit can be set at most once in the algorithm, each edge can be charged at most *V* times, once for each vertex that can be reached from the sink of that edge. Thus, the total cost of the charges for the false test over the entire algorithm is O(*EV).*

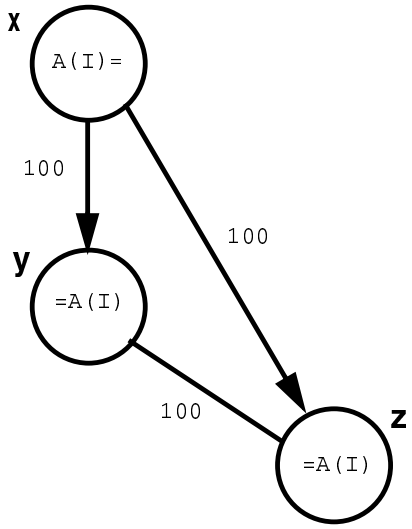
These considerations establish that *pathFrom* and *badPathFrom* can be updated in O($EV + V^2$) time over the entire algorithm. An identical analysis establishes the same time bound for the updates to *pathTo* and *badPathTo*.

3.2.3 Final Observations

To sum up, the total cost of the algorithm is O($EV + V^2$) for the first five phases, O(EV) for phase 6a and O($EV + V^2$) for phase 6b. Thus the entire algorithm takes time O($EV + V^2$).

A critical quality of the algorithm is the recomputation of weights. between a collapsed vertex and some successor or predecessor. In the algorithm as presented the cost of a combined edge is computed by adding the costs of the edges incident on the two original vertices. In many cases this is not the best method. For example, in the fusion for reuse problem. We might have a use of the same array variable in three different loops. Suppose that the loops each execute 100 iterations. The graph for the three loops might look like the one in Figure 13.

FIGURE 13 Weight computation example.

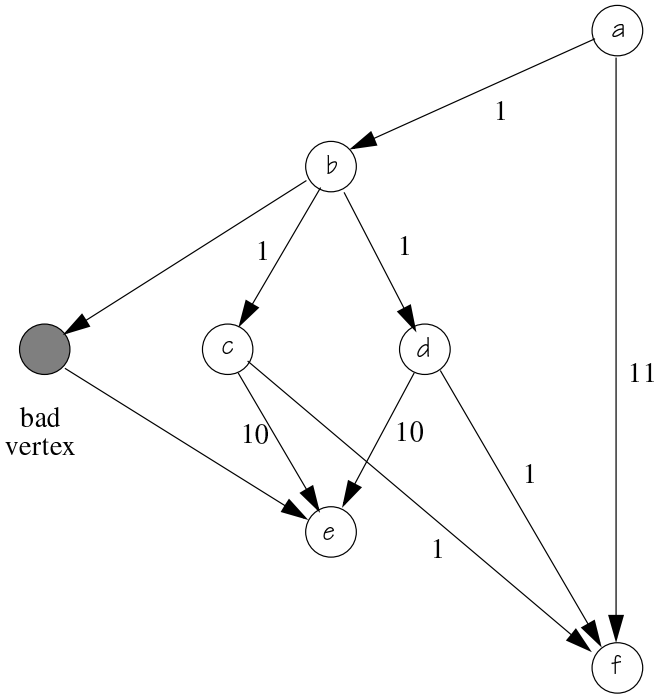


In this example, If vertices *x* and *y* are merged, reweighting by addition would compute an aggregate weight of 200 between *x* and *z*, when the total additional reuse available after merging is 100. In this case, using *maximum* as the reweighting operation would be more appropriate.

A nice feature of the algorithm presented is that edge reweighting is done at most O(*E*) times. This means we can use a relatively expensive reweighting algorithm without substantively affecting the asymptotic time bound. For example, suppose there are *A* different arrays in the program. If we associate a length A array with each vertex that indicates the volume of usage of each array in that loop, then reweighting could be done by taking the maximum of the usage counts of the arrays in the two vertices being collapsed and then summing the minimums of the weights between corresponding arrays in each successor of the composite vertex. This array-by-array computation could be done in O(*A*) time. So the entire fusion algorithm would take time O((*E* + *V*)(*V*+*A*)). If *A* = O(*V*) then the algorithm has the same asymptotic time bound as the fusion algorithm that uses addition for reweighting.

Although the algorithm should produce good answers in practice, the diagram in Figure 14 shows that it will not always produce the optimal result.

FIGURE 14 Non-optimality of greedy weighted fusion.



In this case the highest weight edge is (*a*,*f*), with a weight of 11. If we collapse this edge, the final merge set will be {*a, b, c, d, f*} and the total weight of all edges will be 16. However, by merging *c* and *d* with *b*, we have precluded merging *c* and *d* with *e* because of the bad vertex on a path from *b* to *e*. If we merged *c* and *d* with *e* and *f* instead, the total weight saved would be 22, a better result.

4.0 Implementation Status

This algorithm is being implemented in the D System infrastructure which supports compilation of Fortran 90 and HPF. The D System uses a standard scalarization procedure for implementing array assignments, so this should make it possible to accurately assess the value of the greedy heuristic on scalarized Fortran 90 programs.

5.0 Related Work

To my knowledge, the fusion algorithm presented here is the first published global fusion algorithm that uses the greedy strategy. Kennedy and McKinley [14] developed the original proof that weighted fusion is NP-complete and presented a heuristic strategy based on repetitive application of Goldberg and Tarjan's maximum flow algorithm for networks [14]. The resulting algorithm takes time $O(kEV\lg(V^2/E))$, where $k \leq V$ is the number of required applications of the max-flow algorithm. It is not clear whether the solutions it produces are better or worse than those produced by greedy weighted fusion. Gao, Olsen, Sarkar, and Thekkath [13] produced a solution to the 0-1 fusion problem in which the only weights for edges are 0 or 1. This approach, designed to support the locality optimization of *array contraction* [16], used an $O(EV)$ prepass to reduce the size of the graph then successively applied max-flow on the reduced graph $G_R = (E_R, V_R)$. The overall algorithm takes time $O(EV+V_R^2 E_R \lg V_R)$. Thus the Gao algorithm produces a heuristic solution to a subproblem of weighted fusion in a time which is asymptotically worse than greedy weighted fusion. However, it is difficult to compare these algorithms because the solutions they produce may differ due to the different heuristics used.

6.0 References

[1] [Alle^83] J.R. Allen, "Dependence analysis for subscripted variables and its application to program transformations," Ph.D dissertation, Department of Mathematical Sciences, Rice University, May, 1983.

[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, October 1992, 1290-1317.

[4] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon, "Coloring heuristics for register allocation," *Proceedings of the SIGPLAN `89 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 24, 7, July 1989, 275–284.

[5] D. Callahan, J. Cocke, and K. Kennedy, "Estimating Interlock and improving balance for pipelined machines," *Journal of Parallel and Distributed Computing* 5, 4, August 1988, 334-358.

[6] S. Carr. Memory hierarchy management. Ph.D. thesis, Department of Computer Science, Rice University, September 1992.

[7] S. Carr, D. Callahan and K. Kennedy, “Improving Register Allocation for Subscripted Variables”, In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, White Plains NY, June 1990.

[8] S. Carr and K. Kennedy, “Compiler Blockability of Numerical Algorithms”, In Proceedings of Supercomputing'92, Minneapolis MN, November 1992.

[9] S. Carr and K. Kennedy, “Scalar Replacement in the Presence of Conditional Control Flow”, Software - Practice & Experience 24(1), January 1994.

[10] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J.Cocke, M.E.Hopkins, P.W. Markstein, “Register allocation via coloring,” *Computer Languages* 6, 1981, 47-57.

[11] G.J. Chaitin, “Register allocation and spilling via graph coloring,” *Proc. SIGPLAN 82 Symposium on Compiler Construction, SIGPLAN Notices* 17, 6, June 1982, 98-105.

[12] F. Chow and J. Hennessy, “Register allocation by priority-based coloring,” *Proc. SIGPLAN 84 Symposium on Compiler Construction, SIGPLAN Notices* 19, 6, June 1984, 222-232.

[13] 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, August 1992.

[14] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, (U.~Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors), Lecture Notes in Computer Science, Number 768, 301-320, Springer--Verlag, Berlin, 1993.

[15] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. Communications of the ACM, 22(2):96–103, February 1979.

[16] 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.