Scheduling Tasks to Maximize Usage of Aggregate Variables In Place *

Samah Mahmeed, Ken Kennedy, Cheryl McCosh,
Zoran Budimlić, and Steve Rogers

August 21, 2006

Abstract

We present an algorithm for greedy in-placeness that runs in $O(TlogT + E_wV + V^2)$ time, where T is the number of in-placeness opportunities, E_w is the aggregate number of wire successors and v is the number of virtual instruments in a program graph.

1 Introduction

2 Greedy In-Place Algorithm

In this section we describe a greedy heuristic algorithm for choosing which pairs of input and output wires can be computed in place at a virtual instrument.

We begin with a representation of the input graph to the problem. Following the practice established at National instruments, we will represent a LabView program as a collection of two kinds of vertices. Let V be the set of virtual instruments in the program and W be the collection of wires. Each wire is produced by a single virtual instrument but may be an input to an unbounded number of virtual instruments. Thus we will assume that there are two sets of edges: E_v is the set of wires out of virtual instruments $v \in V$ and E_w is the set of edges out of wires $w \in W$. Note that the number of wires out of virtual instruments is the same as the number of wires in total, since each wire is the output of a single virtual instrument. Thus we write:

$$||W|| = ||E_v|| \tag{1}$$

*This research was supported by the National Instruments Corporation.

As a notational convention, we will use set names of wires and edges to represent both the set itself and the number of elements in the set, whenever the context is clear. Thus $O(E_w V)$ means the same as $O(||E_w||||V||)$.

For the purposes of this algorithm the edges in E_w will be organized as an adjacency list that allows us to quickly iterate over all virtual instruments that consume a particular wire w.

The algorithm will begin by constructing a set of in-placeness *opportunities* T, each of which is a triple $t = (v, w_1, w_2)$ where w_1 is the input to v that could be overwritten in place by the contents of output wire w_2. Since this overwriting is by definition destructive, choosing this triple for in-placing requires that all other consumers of w_1 be scheduled before it. Thus if there is a path in the original graph from v to another virtual instrument x that also consumes wire w_1, the triple t cannot be in-placed.

For each triple t that represents an in-placeness opportunity, there will be a *benefit* representing the value (in saved copies, for example) of in-placing t. We will denote this as $B(t)$.

The greedy algorithm will proceed by selecting the triple with maximum benefit for in-placing, testing whether the in-placing is legal, and marking it as in-placed. Whenever a particular triple is in-placed, it creates new scheduling constraints that can make other in-place choices illegal. As we have said, if $t = (v, w_1, w_2)$ is in-placed then all of the other virtual instruments to which w_1 is an input must be scheduled before v. In effect, this creates new paths in the graph that must be considered when testing for the legality of other in-place decisions.

We will model these effects by creating a *scheduling graph* $G_S = (V, E)$ in which the vertex set V is the set of virtual instruments as before and each $(x, y) in E$ indicates that virtual instrument x must be scheduled *before* virtual instrument y. The initial version of the scheduling graph will be computed from the program graph by a simple direct calculation.

In addition, to make the legality testing fast, we will compute a side data structure *After*, which represents the transitive closure of the scheduling graph at any point in the program. That is, $y \in After(x)$ if and only if y must be scheduled after x in the current scheduling graph.

The steps of the greedy in-placeness are as follows:

- Compute the set T of reasonable opportunities for in-placing and compute a benefit for each. T should be organized as a priority queue (*e.g.*, a heap).
- Compute the initial scheduling graph G_S and the inital *After* relationship from the initial program graph (V, W, E_v, E_w).

- While T is non-empty, iteratively remove the highest benefit triple t and test for in-place legality. If it is legal, mark the triple as in-placed and update both G_S and *After* to reflect the new scheduling constraints introduced by in-placing the triple.

These steps are described in the following subsections.

2.1 Constructing the Opportunities Heap

In theory, the number of in-placeness opportunities is large. A loose upper bound is the number of input wires times the number of output wires. Thus there could be as many as $E_v E_w = W E_w$ triples. However, a number of factors will reduce this bound. First, at a particular virtual instrument, some choices may not be appropriate. For example, it is not likely that computing an array in place with a much smaller array or a scalar wire will be useful.

Once a pruned set of opportunities T is constructed, it can be reorganized into a heap in $O(TlogT)$ time.

Overall, the total number of triples that are *chosen* for inplacing should be far smaller than the number that are considered. Thus it is very important that we be able to rapidly test for legality of in-placing a particular triple. As a result, much of the machinery in this algorithm is designed to facilitate such a fast test.

2.2 Constructing the Initial Graph

The goal of this activity is to construct the initial Scheduling Graph G_S along with the side data structure *After*, which can be thought of as the transitive closure of the initial scheduling graph. The algorithm for accomplishing this is given in n Figure **??**.

It is fairly straightforward to determine an upper bound on the time spent in this initialization. L1 entered $O(V)$ times and the body of L2 is executed $O(E_w)$ times. (Here we count the header of L2 and the enclosing loop as a single loop.) So the entire time spent in loop L1 is $O(E_w + V)$

The loop at L3 implements a straight transitive closure. The loop at L3 is entered V times, while the loop at L4 is executed once for each edge in E. Since the loop body contains a set operation taking at most $O(V)$ time, the time taken by the entire loop is $O(EV)$.

Figure 1: Algorithm for Constructing G_S, *After*, and I

```
procedure InitGraph(V, E_v, E_w. E, successors, predecessors, After);
    for each v ∈ V do begin
        After[v] := ∅;
        count[v] := 0; predecessors[y] := ∅; successors[x] := ∅;
    end
    for each wire w ∈ W do
        for each v ∈ successors[w] do count[v] := count[v] + 1;
    worklist := ∅; E := ∅;
    for each v ∈ V do
        if count[v] = 0 then worklist := worklist ∪ {v};
L1: while worklist ≠ ∅ do begin
        pick an element x from the front of worklist and remove it;
        for each output wire w from x do begin
L2:         for each y ∈ successors[w] do begin
                if (x, y) ∉ E then begin
                    E := E ∪ {(x, y)};
                    predecessors[y] := predecessors[y] ∪{x};
                    successors[x] := successors[x] ∪ {y};
                end
                count[y] := count[y] − 1;
                if count[y] = 0 then worklist := worklist ∪ {y};
            end
        end
    end
    // Next compute the initial After relationship, backing up through G_S
    for each v ∈ V do s_count[v] := 0;
    for each (x, y) ∈ E do s_count[x] := s_count[x] + 1;
    for each v ∈ V do
        if s_count [v] = 0 then worklist := worklist ∪ {v};
L3: while worklist ≠ ∅ do begin
        pick an element y from the front of worklist and remove it;
L4:     for each x ∈ predecessors[y] do begin
            After[x] := After[x] ∪ After[y];
            s_count[x] := s_count[y] − 1;
            if s_count[x] = 0 then worklist := worklist ∪ {x};
        end
    end
end InitGraph
```

2.3 Picking Opportunities and Testing for Legality

Once we have the triples priority queue T, organized by benefit B, we can iteratively select an in-placeness opportunity, test it for legality.

A triple $t = (v, w_1, w_2)$ is legal to in-place if *neither* of the following conditions hold:

- There exists some $x \in \mathit{successors}[w_1] - \{v\}$ such that $x \in \mathit{After}[v]$. This would violate the requirement that v be scheduled after all other sinks of w_1.
- $w_1 \in I[v]$ or $w_2 \in I[v]$, where a wire is in $I[v]$ if it is either the input or output of a triple that has *already* been in-placed. This would violate the restriction that neither w_1 nor w_2 can be previously in-placed as a part of some other triple.

If in-placing the triple is legal, new scheduling constraints must be introduced and the side data structures must be updated to incorporate these new scheduling constraints. The code for this part, including the initialization of $I[v]$ is given in Figure **??**.

Note that the total time for this code, not counting the time spent in *InitGraph* and *UpdateGraph*, is $O(TlogT + E_wV)$. It is clear that the heap operations take $O(TlogT)$. The trick here is to avoid traversing the successors of an input wire w_1 every time a triple with that as an input is processed. The side data structure *wire_used* avoids this because once a wire has been in-placed at some vertex, it cannot be in-placed at some other vertex, because that would create a scheduling cycle. The set *wire_used* contains all input wires that have already been in-placed. Because it is interrogated first, we traverse the successors of a wire at most once for every vertex to which it is an input where it might be in-placed. If one of those traversals results in a legal in-placeness decision, the traversals are significantly fewer. Overall the total time spent traversing the successors of an input wire is $O(E_wV)$.

Observe that if there is no pruning of the set of triples T, then $T = O(E_wV)$ so the entire process, aside from the graph updating, takes $O(TlogT)$ time. However, we will assume that there is significant pruning so it is useful to separate the two time estimates to yield $O(TlogT + E_wV)$.

2.4 Updating the Graph

We now turn to the process for updating the graph after in-placing $t = (v, w_1, w_2)$, perhaps the trickiest part of the algorithm. The goal is to produce a time bound

Figure 2: Main Algorithm for Greedy In-Placing

```
InitGraph(V, E_v, E_w. E, successors, predecessors, After);
for each v ∈ V do I(v) := ⊘;
wire_used := ⊘;
while T ≠ ⊘ do begin
    pick the highest-benefit element t = (v, w_1, w_2) from the top of the heap, remove it, and reheap
    // Test for legality
    legal := true;
    if w_1 ∈ I[v] or w_2 ∈ I[v] then legal := false;
    if w_1 ∈ wire_used then legal := false;
    if legal then begin
        other_inputs := successors[w_1] − {v};
        while legal and other_inputs ≠ ⊘ do begin
            pick an element x from other_inputs and remove it;
            if x ∈ After[v] then legal := false;
        end
    end
    if legal then begin
        mark t = (v, w_1, w_2) as in-placed;
        I[v] := I[v] ∪ {w_1} ∪ {w_2};
        UpdateGraph(v, w_1, V, E, successors, predecessors, After);
    end
end
```

of $O(EV + V^2)$ time, where E is the number of edges in the scheduling graph G_S. Since $E \leq E_w$, this will give us the desired time bound for the running time of the algorithm.

The procedure begins by inserting new edges between all the other virtual instruments to which the input wire w_1 is also an input and updating the predecessor and successor lists. Then, the algorithm must update the *After* data structure. To avoid spending too much time at this task, we will try to actually add a new vertex to *After*[v] only once for each v. This will require backing up through the predecessors of all the vertices with new edges (other inputs of w_1) while maintaining a new data structure called *newAfter*, which gets reduced whenever there already exists a path to some element in *After* for the

predecessor. The algorithm is given in Figure **??**.

There are a few comments on the complexity of this algorithm. First, since each wire w_1 is input to an in-placed triple only once, the body of loop L1 is executed only E_w times. Furthermore, since the conditional at statemetnt S1 eliminates duplicate edges, the body of the conditional is executed at most E times over the entire algorithm.

At this point it is useful to note that the number of edges E grows during the algorithm. As we observed earlier, E is smaller than E_w initially. However, can we still establish a bound on the size of E in terms of E_w after the algorithm is done? To answer, consider that at each in-placing step, the input wire w_1 can no longer be in-placed at any of its other inputs. So the total number of in-placings is bounded by the number of wires W. At each such in-placing we put edges into E for each vertex to which the wire w_1 is an input except the in-placed vertex. The total number of such edges is therefore at most $E_w - W$. Hence the total number of edges in E after all in-placing steps is no more than $2E_w - W = O(E_w)$.

Returning to loop L1, the most expensive operations are the set unions and differences, each of which take $O(V)$ time, so the entire cost of loop L1 is $O(E_w + EV) = O(E_wV)$. Note that we limit, in statements S2 and S4, the number of times a vertex goes on the worklist to those times when it will actually add a vertex to its *After* set. Hence no vertex goes on the worklist more than V times.

Now we consider loop L2. Since a vertex y can only be added to the worklist at most V times, the body of L2 is executed an aggregate of V^2 times. However, if we count the number of times the body of loop L3 is executed, we want to charge the cost, including the cost of the loop iterator, to the edge (x, y). Given that each y can be on the worklist only V times, this means that the total number of times that we can process each edge is V times, so the total number of executions of the body of L3 is EV. Even though the header of loop L4 is executed EV times, in actuality, we can charge each execution of the body to a new element of the *After* set for y. Again, since an element is added to the *After* set only once per vertex, the total number of executions of the body of L4 is $O(V^2)$ What about the body of the if-statement at S3? All these operations are constant time, so the aggregate time over the entire algorithm for L4 is $O(V^2)$.

Thus we have shown that the aggregate time for all operations in loop L2 is $O(EV + V^2)$. Since $E = O(E_w)$, we have established that the running time for the entire algorithm is bounded by $O(TlogT + E_wV + V^2)$.

Figure 3: Algorithm for Updating Slices

```
procedure UpdateGraph(v, w_1, V, E, successors, predecessors, After);
    // v is the vertex where the in-placing is happening and w_1 is the input
    // The graph being updated is G_S = (V, E)
    // Actual updates occur to E, successors, predecessors
    // The side data structure After is also updated
    // newAfter[x] = the set of vertices added to the After set for x by this in-placing
    // The set processed is used to ensure that a vertex goes on worklist at most once
    worklist := ∅; processed := ∅
L1: for each y ∈ successors[w_1] − {v} do begin
S1:     if (y, v) ∉ E then begin
            E := E ∪ (y, v);
            successors[y] := successors[y] ∪ {v};
            prececessors[v] := predecesors[v] ∪ {x};
            newAfter[y] := After[v] − After[y];
            After[y] := After[y] ∪ After[v];
S2:         if newAfter[y] ≠ ∅ then begin
                worklist := worklist ∪ {y}; processed := processed ∪ {y};
            end
        end
    end
    // Now we update the After sets by backing up through the graph
L2: while worklist ≠ ∅ do begin
        pick an element y from the front of worklist and remove it;
L3:     for each x ∈ predecessors[y] − processed do begin
L4:         for each z ∈ newAfter[y] do begin
S3:             if z ∉ After[x] then begin
                    After[x] := After[x] ∪ {z};
                    newAfter[x] := newAfter[x] ∪ {z};
                end
S4:             if newAfter[x] ≠ ∅ then begin
                    worklist := worklist ∪ {x}; processed := processed ∪ {x};
                end
            end
        end
    end
end UpdateGraph;
```