Scheduling Tasks to Maximize Usage of Aggregate Variables In Place

Samah Abu-Mahmeed[1], Cheryl McCosh[1], Zoran Budimlić[1], Ken Kennedy[1], Kaushik Ravindran[2], Kevin Hogan[2], Paul Austin[2], Steve Rogers[2], and Jacob Kornerup[2]

[1] Computer Science Department, Rice University, Houston, TX 77005
{samah, chom, zoran}@cs.rice.edu
[2] National Instruments, 11500 North MoPac Expressway, Austin, TX 78759
{kaushik.ravindran, kevin.hogan, paul.austin, steve.rogers, jacob.kornerup}@ni.com

Abstract. Single-assignment languages with copy semantics have a very simple and approachable programming model. A naïve implementation of the copy semantics that copies the result of every computation to a new location, can result in poor performance. Whereas, an implementation that keeps the results in the same location, when possible, can achieve much higher performance.

In this paper, we present a greedy algorithm for in-place computation of aggregate (array and structure) variables. Our algorithm greedily picks the most profitable opportunities for in-place computation, then updates the scheduling and in-place constraints in the program graph. The algorithm runs in $O(TlogT + E_W V + V^2)$ time, where T is the number of in-placeness opportunities, E_W is the number of edges and V the number of computational nodes in a program graph.

We evaluate the performance of the code generated by the LabVIEW^{TM} compiler using our algorithm against the code that performs no in-place computation at all, resulting in significant application performance improvements. We also compare the performance of the code generated by our algorithm against the commercial LabVIEW compiler that uses an ad-hoc in-placeness strategy. The results show that our algorithm matches the performance of the current LabVIEW strategy in most cases, while in some cases outperforming it significantly.

1 Introduction

At their core, functional, data-flow programming and other single-assignment languages are free from side-effects, making it easier to write, parallelize, verify and optimize programs written in such languages. At the conceptual level, the side-effect free behavior implies that the involved variables are copied at each stage. Translated directly into an implementation this can result in programs that consume large amounts of time and space. Compiler transformations that recognize unnecessary copies and avoid them can significantly improve the

performance of programs in such languages. We call such copy avoidance transformations an in-placeness strategy, since data that is not copied is kept in-place. Unfortunately, the general problem of finding the minimum number of copies in a program involving aggregate data structures is NP-Complete [1].

In this paper (an extended version is given in [2]), we present an $O(TlogT + E_W V + V^2)$ greedy in-placeness algorithm that significantly reduces the amount of copying of aggregate data structures for single assignment languages. Here, T is the number of in-placeness opportunities, E_W is the aggregate number of edges and V is the number of computational nodes in a program graph. This algorithm is general and can be applied to wide range of functional, data-flow and other single assignment languages. We have chosen to implement a prototype of the algorithm as an optimization phase in a compiler for LabVIEW$^{\mathrm{TM}}$, a graphical data flow programming language from National Instruments Corporation (NI) [3], but it can be just as easily implemented in any other modern compiler infrastructure for other single assignment and functional languages. LabVIEW is a compiled, statically typed programming language widely used by scientists and engineers around the world. An integral part of its compilation process is its in-placeness strategy. Our experiments show that using the in-placeness strategy currently shipping with the LabVIEW compiler results in much faster and less memory intensive programs when compared to performing no computations in-place. The in-placeness algorithm presented in this paper results in a more robust performance, on par with the current LabVIEW strategy in most cases, while in some cases showing significant (orders of magnitude) performance improvement compared to the shipping LabVIEW compiler.

Our in-placeness algorithm saves both space and time in an executing program. It saves space by performing computations in-place. By performing in-place operations on aggregate data structures that only change a (small) part of the data structure, it saves the time needed for copying unchanged data.

The rest of the paper is organized as follows. Section 2 presents the LabVIEW data flow language and the in-placeness strategy used in its compiler. Section 3 states the in-placeness problem as a constrained optimization problem on a program graph. Section 4 describes our algorithm and analyzes its complexity. Section 5 presents the experimental results comparing our algorithm to the current LabVIEW strategy and program execution without any in-placeness optimizations. Section 6 describes the related work in register allocation and copy elimination in functional languages. Section 7 concludes the paper and suggests directions for future work.

2 The LabVIEW language

LabVIEW is a graphical data flow programming language from National Instruments Corporation (NI), widely used in industry for implementation of control and measurement systems and embedded applications [3]. In this section we will briefly describe the semantics of LabVIEW, give an example of an in-placeness optimization, and the NI heuristic for determining in-placeness.

2.1 Overview

A source program written in LabVIEW is referred to as a virtual instrument (VI). A VI consists of a front panel (graphical user interface) and a block diagram (a graphical data flow diagram) where icons and structures are used instead of textual instructions and wires are used instead of variables. Figure 1 shows a simple VI that computes the greatest common divisor (GCD) of the integer values provided to the controls (X and Y) on the front panel (on the right) or from a call to this VI from another VI, as part of a call chain.

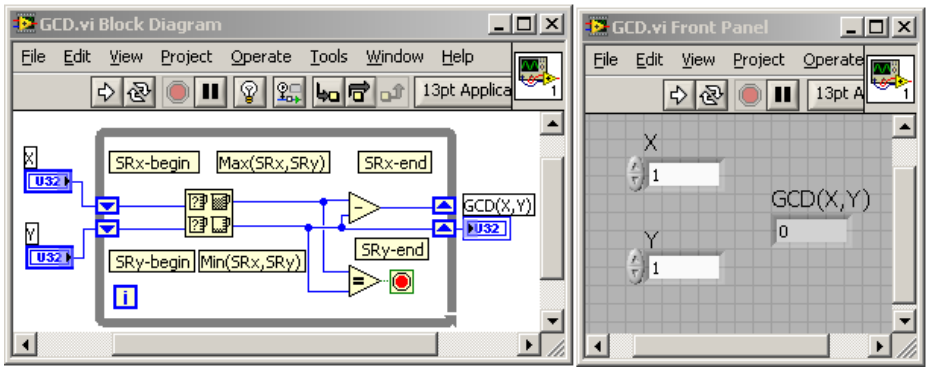

Fig. 1. A LabVIEW program for computing the greatest common divisor.

When the VI is executed it copies the values into the two shift registers (labeled SRx-begin and SRy-begin) on the while loop (the box structure). On each iteration of the while loop the value of SRx-begin and SRy-begin are copied into the Min/Max node. It returns the maximum (minimum) value on the top (bottom) wire. The minimum value is copied back into shift register SRy-end, on the right hand side of the loop, and the difference between the minimum and maximum value is copied into shift register SRx-end. If the minimum and maximum values are different then the loop will execute another iteration; otherwise the value of SRy-end is copied to the front panel (or to the calling environment, if used as sub-VI) as the GCD of the provided values. Note that the program itself does not specify the order of the subtraction and the comparison nodes; in principle they can be executed in any order, including simultaneously.

The LabVIEW compiler uses a technique called "clumping", where a selected set of data-flow dependent nodes are combined into a single schedulable unit (a clump). This allows the run-time system to schedule VIs at the clump level, instead of at the individual node level. The nodes inside a clump are sorted topologically according to their data flow dependencies, resulting in a total (sequential) execution order within a clump. When defining the topological sort of a clump, the compiler applies a weight to each node that is proportional to the amount of data that the node may copy, so that non-copying nodes get scheduled ahead of copying nodes when they have no data flow dependency.

While the example VI above was explained in terms of copying values, it turns out that this VI can reuse the memory locations reserved for SRx-begin

and SRy-begin for all intermediate integer results on the wires inside the while loop. This is achieved by statically scheduling the comparison node before the subtraction node, since the comparison node cannot make its outputs in-place to its inputs due to a type (size) mismatch. By reusing the memory locations reserved for SRx-begin and SRy-begin the VI will use less memory and also execute faster since it executes fewer copy instructions. These savings are more dramatic when dealing with complex data structures such as arrays, where the computational node may only change the values of a subset of the array.

2.2 The NI In-placeness Heuristic

The in-place strategy currently used by LabVIEW is based on local and static decisions; binary operators, like the subtraction node in the GCD example, will make its output in-place to its top input if their types match. In the GCD example, this happened to be the best choice, but in general this approach is not optimal. The LabVIEW compiler is also hand-tuned for cases where this heuristic does not perform well. This ad-hoc implementation is difficult to maintain and requires expert knowledge to program applications and take full advantage of in-placeness opportunities available from the compiler. A more systematic approach to in-place computation is the main contribution of this paper.

3 Problem Description

First, we will formally encode in-placeness selection in LabVIEW as a constrained optimization problem on a program graph. We begin with a representation of the program as a directed acyclic graph (DAG). The input graph represents a set of data-flow dependent nodes that are part of a single schedulable unit (clump). The objective is to compute a schedule of the nodes of the clump in a single thread that maximizes the benefit from in-place computation.

3.1 Program Graph

We represent a LabVIEW program as a collection of two kinds of vertices. Let V be the set of computational nodes in the program and W be the collection of wires or memory locations. Each wire is the output of a single computational node but may be an input to an unbounded number of computational nodes. Thus we will assume that there are two sets of edges: E_V and E_W. $E_V \subseteq V \times W$ is the set of edges that connect a computational node to the wires that are produced as outputs from it. $E_W \subseteq W \times V$ is the set of edges that connect a wire to the computational nodes that use it as input. Note that the number of edges out of computational nodes is the same as the number of wires in total, since each wire is the output of a single computational node (i.e. $||W|| = ||E_V||$). As a notational convention, we will use the set names of the vertices 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 $O(||E_W||\ ||V||)$.

Figure 2 shows the program graph for the LabVIEW program from Figure 1 for computing the GCD of two integers. The set V of computational nodes

consists of the MAX/MIN, SUB, and EQ vertices (denoted by square vertices in Figure 2). The remaining vertices (denoted by circles) comprise the set W of wires or memory locations. Note that the wires $MIN(SRx, SRy)$ and $SRy-end$ correspond to the same memory location in the LabVIEW program in Figure 1, hence they are aggregated into one vertex in Figure 2. The edges preserve the data dependencies between the computational nodes in the LabVIEW program.

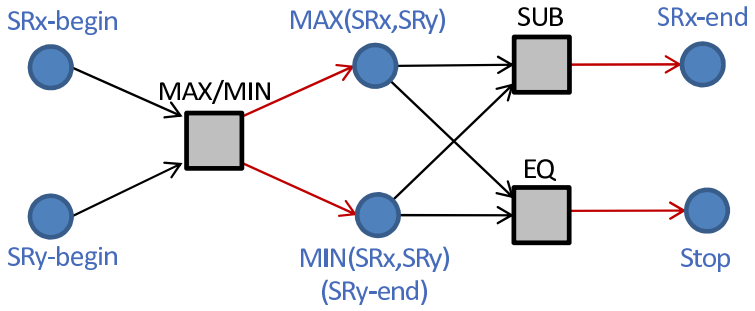

Fig. 2. Program graph for the LabVIEW program from Figure 1.

3.2 In-placeness Opportunities

The algorithm begins by constructing a set of *in-placeness opportunities* T, where

$$T \subseteq \{(w_1, v, w_2) \in W \times V \times W \mid (w_1, v) \in E_W, (v, w_2) \in E_V\}.$$

In other words, each in-placeness opportunity is a triple $t = (w_1, v, w_2)$ where wire w_1 is the input to computational node 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 computational node v' that also consumes wire w_1, the triple t cannot be in-placed.

For each triple $t \in T$ that represents an in-placeness opportunity, there will be a *benefit* representing the value of in-placing t. The benefit measures the advantage gained by avoiding a memory copy and performing the computation in-place. We denote this as $B(t)$. For example, if a computational node v updates one element of an input array $A1$ to produce the array $A2$ then the benefit $B(t)$ of in-placing the triple $t = (A1, v, A2)$ is $size(A1) - 1$.

3.3 Optimization Objective and Constraints

The objective is to select in-placeness opportunities from the set T to maximize the total benefit, while adhering to the following constraints: (a) any wire $w \in W$ is an input to (similarly, output of) at most one selected in-placeness opportunity, and (b) if $(w_1, v, w_2) \in T$ is selected for in-placeness, then all other consumers of w_1 must be scheduled before v.

The inputs to the optimization problem are a LabVIEW program $G = (V, W, E_V, E_W)$, a set of in-placeness opportunities T, and a benefit function

B. Let function $x : T \rightarrow \{0, 1\}$ denote whether $t \in T$ is selected for in-placeness. Also, let function $S : V \rightarrow \mathcal{Z}$ denote the position of $v \in V$ in a schedule of the program graph G. A valid solution to the optimization problem is characterized by functions x and S that satisfy the following four constraints:

$$
\begin{aligned}
&(a)\ \forall t = (w_1, v, w_2), t' = (w_1', v', w_2') \in T, t \neq t', w_1 = w_1', \\
&\qquad x(t) = 1 \ \Rightarrow\ x(t') = 0 \text{ (unique input in-placeness)} \\
&(b)\ \forall t = (w_1, v, w_2), t' = (w_1', v', w_2') \in T, t \neq t', w_2 = w_2', \\
&\qquad x(t) = 1 \ \Rightarrow\ x(t') = 0 \text{ (unique output in-placeness)} \\
&(c)\ \forall (v_1, w_1) \in E_V, \forall (w_2, v_2) \in E_W, \\
&\qquad w_1 = w_2 \ \Rightarrow\ S(v_2) > S(v_1) \text{ (program dependencies)} \\
&(d)\ \forall t = (w_1, v, w_2) \in T, \forall (w_1, v') \in E_W, v' \neq v, \\
&\qquad x(t) = 1 \ \Rightarrow\ S(v) > S(v') \text{ (ordering due to in-placeness)} .
\end{aligned}
$$

Constraint (a) specifies that if $t = (w_1, v, w_2) \in T$ is selected for in-placeness, then the opportunities $t' \in T$ which have w_1 as input are not selected. Constraint (b) imposes a similar condition on the output wire of an in-placeness opportunity. Constraint (c) enforces the dependencies between computational nodes from the program graph in the resulting schedule. Finally, constraint (d) connects in-placeness selections to new scheduling constraints, which are not implied by the dependencies in the original program graph. In particular, if $t = (w_1, v, w_2) \in T$ is selected for in-placeness, the constraint ensures that the computational node v is scheduled only after all other computational nodes $v' \neq v$ which consume w_1 have been scheduled. The optimization objective is to compute valid solutions to x and S, such that the total benefit $\Sigma_{t \in T} B(t)x(t)$ is maximized.

4 Greedy In-Place Algorithm

The main contribution of this paper is an efficient heuristic algorithm for choosing pairs of inputs and outputs to compute in-place. The greedy algorithm selects the triple with maximum benefit for in-placing, tests whether the in-placing is legal, and marks it as in-placed. When a particular triple is in-placed, it creates new scheduling constraints that can make other in-place choices illegal.

We model these effects by creating a *scheduling graph* $G_S = (V, E)$ in which the vertex set V is the set of computational nodes as before and each $(x, y) \in E$ indicates that computational node x must be scheduled *before* computational node y. The initial version of the scheduling graph is a straightforward translation of the computed program graph. The scheduling graph is a Direct Acyclic Graph (DAG), since legal LabVIEW program cannot have cycles, and every step of our algorithm ensures that there are no cycles introduced in it.

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

The steps of the greedy in-placeness algorithm are:

– Compute a priority queue T of in-place opportunities.
– Compute the initial scheduling graph G_S and the initial *Aft* relationship from the program graph (V, W, E_V, E_W).
– While T is non-empty, iteratively remove the highest benefit triple t. If it is legal to in-place, mark the triple as in-placed and update both G_S and *Aft* to reflect the new scheduling constraints introduced by in-placing the triple.

4.1 Constructing the Opportunities Heap

While the number of in-placeness opportunities may be large (a loose upper bound is $E_V E_W = W E_W$), some pruning will reduce it. For example, it is not likely that computing an array in place with a much smaller array or a scalar will be useful. We can reorganize a pruned set of opportunities T into a heap in $O(TlogT)$ time. Since the total number of triples that are *chosen* for in-placing will be far smaller than the number that are considered, it is very important that we be able to rapidly test for legality of in-placing a particular triple, and much of the machinery in this algorithm is designed to facilitate such a fast test.

4.2 Constructing the Initial Graph

The algorithm *InitGraph* in Figure 4.2 constructs the initial scheduling graph G_S along with the side data structure *Aft*, which is a transitive closure of the initial scheduling graph. The upper bound on the time spent in this initialization is fairly straightforward: L1 is entered $O(V)$ times and the body of L2 is executed $O(E_W)$ times (the header of L2 and the enclosing loop count as a single loop.) So the entire time spent in loop L1 is $O(E_W + V)$. The loop at L3, which implements a transitive closure, 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 $O(EV)$.

4.3 Selecting In-placeness Opportunities

Once we have the priority queue T of triplets, organized by benefit B, we can iteratively select an in-placeness opportunity and test it for legality. A triple $t = (w_1, v, w_2)$ is legal to in-place if *neither* of the following conditions hold:

– There exists some $x \in suc[w_1] - \{v\}$ such that $x \in Aft[v]$. 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.

If in-placing the triple is legal, we introduce new scheduling constraints and update the side data structures. The pseuco code for this part is given in procedure *GreedyInplace* in Figure 4.2.

The execution time of this procedure, not counting the time spent in *InitGraph* and *UpdateGraph*, is $O(TlogT + E_W V)$. The heap operations take $O(TlogT)$. To avoid traversing the successors of an input wire w_1 every time a triple with

```
procedure InitGraph(V, E_V, E_W. E,
          suc, pred, Aft);
    for each v ∈ V do begin
     Aft[v] := ∅;
     count[v] := 0; pred[y] := suc[x] := ∅;
    end
    for each wire w ∈ W do
     for each v ∈ suc[w] do count[v]++;
    worklist := ∅; E := ∅;
    for each v ∈ V do
     if count[v] = 0 then worklist ∪ ={v};
L1: while worklist ≠ ∅ do begin
     remove element x from front of worklist;
     for each output wire w from x do begin
L2:    for each y ∈ suc[w] do begin
        if (x, y) ∉ E then begin
         E := E ∪ {(x, y)};
         pred[y] ∪ ={x}; suc[x] ∪ = {y};
        end
        count[y] − −;
        if count[y] = 0 then worklist ∪ = {y};
       end
      end
    end
    // Next compute the initial Aft
    // relationship, backing up through G_S
    for each v ∈ V do s_c[v] := 0;
    for each (x, y) ∈ E do s_c[x]++;
    for each v ∈ V do
     if s_c [v] = 0 then worklist ∪ = {v};
L3: while worklist ≠ ∅ do begin
     remove element y from front of worklist;
L4:  for each x ∈ pred[y] do begin
       Aft[x] ∪ =Aft[y];s_c[x] := s_c[x] − 1;
       if s_c[x] = 0 then worklist ∪ ={x};
      end
    end
end InitGraph
```

```
procedure GreedyInplace(V, E_V, E_W. E,
          suc, pred, Aft);
for each v ∈ V do I(v) := ∅;
wire_used := ∅;
while T ≠ ∅ do begin
 remove highest-benefit element t = (w_1, v, w_2)
    from the top of the heap, 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 := suc[w_1] − {v};
  while legal and other_inputs ≠ ∅ do begin
   remove an element x from other_inputs;
   if x ∈ Aft[v] then legal := false;
  end
 end
 if legal then begin
  mark t = (w_1, v, w_2) as in-placed;
  I[v] := I[v] ∪ {w_1} ∪ {w_2};
  wire_used = wire_used ∪{w_1};
  UpdateGraph(v, w_1, V, E, suc, pred, Aft);
 end
end
```

```
procedure UpdateGraph(v, w_1, V, E, suc, pred, Aft);
// v: vertex where in-placing happens, w_1: input, G_S = (V, E): graph being updated
// Actual updates occur to E, suc, pred, The side data structure Aft is also updated,
// newAft[x] = the set of vertices added to the Aft set of 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 ∈ suc[w_1] − {v} do begin
S1:  if (y, v) ∉ E then begin
       E ∪ = (y, v); suc[y] ∪ = {v}; pred[v] ∪ = {x}; newAft[y] := Aft[v] − Aft[y]; Aft[y] ∪ = Aft[v];
S2:    if newAft[y] ≠ ∅ then {worklist ∪ = {y}; processed ∪ = {y}; }
      end
    end
    // Update the Aft sets by backing up through the graph
L2: while worklist ≠ ∅ do begin
     remove an element y from the front of worklist;
L3:  for each x ∈ pred[y] − processed do begin
L4:   for each z ∈ newAft[y] do begin
S3:     if z ∉ Aft[x] then {Aft[x] ∪ = {z}; newAft[x] ∪ = {z}; }
S4:     if newAft[x] ≠ ∅ then {worklist ∪ = {x}; processed ∪ = {x}; }
       end
      end
    end
end UpdateGraph;
```

Fig. 3. Algorithms

that wire as an input is processed, we use the side data structure *wire_used*, containing all input wires that have already been in-placed (once a wire has been in-placed at some vertex, it cannot be in-placed at any other vertex, since that would create a scheduling cycle). Since we interrogate *wire_used* first, we traverse the successors of a wire at most once for every vertex to which it is an input that might be in-placed. Overall the total time spent traversing the successors of an input wire is $O(E_W V)$. Observe that if there is no pruning of the set of triples T,

then $T = O(E_W V)$ so the entire process, aside from the graph updating, takes $O(TlogT)$ time. However, assuming that significant pruning is done, it is useful to separate the two terms to yield $O(TlogT + E_W V)$.

4.4 Updating the Scheduling Graph

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

The procedure begins by inserting new edges between all the other computational nodes to which the input wire w_1 is also an input and updating the predecessor and successor lists. Then, the algorithm must update the *Aft* data structure. We add a new vertex to *Aft*[v] only once for each v. This requires backing up through the predecessors *pred* of all the vertices with new edges (other inputs of w_1) while maintaining a new data structure called *newAft*, which gets reduced whenever there already exists a path to some element in *Aft* for the predecessor. The algorithm *UpdateGraph* in Figure 4.2 describes this process.

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 S1 eliminates duplicate edges, the body of the conditional is executed at most E times over the entire program.

Even though E is smaller than E_W initially, it grows during the execution. However, we can still establish a bound on the size of E in terms of E_W after the algorithm is done, since 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, which is at most $E_W - W$ edges. The total number of edges in E after all in-placing steps is no more than $2E_W - W = O(E_W)$.

In loop L1, the most expensive operations are the set unions and differences, each taking $O(V)$ time, so the entire cost of loop L1 is $O(E_W + EV) = O(E_W V)$. 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 *Aft* set, which is not more than V times.

Now 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, so the total number of executions of the body of L3 is EV. Even though the header of loop L4 is executed EV times, we can charge each execution of the body to a new element of the *Aft* set for y, which is only once per vertex. The total number of executions of the body of L4 is $O(V^2)$. Since the body of the *if* statement S3 takes constant time, the

aggregate time over the entire algorithm for L4 is $O(V^2)$. The aggregate time for loop L2 is therefore $O(EV + V^2)$. Since $E = O(E_W)$, we have established that the running time for the entire algorithm is bounded by $O(TlogT + E_W V + V^2)$.

Note that the algorithm can be safely stopped at any time, since no node is ever *unmarked* for in-place computation. Running the algorithm to completion only affects the quality of the result, not the correctness.

4.5 Loops and Shift Registers

In LabVIEW, *shift registers* are used to represent the loop-carried dependences; they are equivalent to induction variables in an imperative language. Each shift register has a source and a sink and at the end of each iteration of a loop the value of the sink of the shift register is copied into the source of the same shift register. In the LabVIEW example in figure 1 there are two shift registers, labelled SRx and SRy. They are used to carry the state of the GCD computation from one iteration of the while loop to the next iteration.

The in-placeness algorithm as we have described it in this section so far only works on straight-line code. Since loops and shift registers (especially shift registers that transfer aggregate data structures) can have an enormous impact on performance, we treat loops and shift registers separately.

Our strategy is to make the in-placeness decisions for loops in three steps; First, all copies on the back edges of a loop are eliminated by in-placing the source of the shift register with its sink. Second, we apply the greedy in-placeness algorithm presented earlier in this section to the body of the loop. Finally, we replace the loop by dummy operations that model the input and output tunnels connecting the loop to the enclosing VI. Then we run our algorithm on the enclosing VI, using the dummy operations to decide on the in-placeness of the inputs and outputs of a loop. For the GCD example in figure 1 the input pairs is Y and SRy-begin and the output pair is SRy-end and GCD(X,Y). The input/output pairs of the dummy are added to the opportunities heap of the enclosing VI as in-placeness opportunities.

In-placing all the shift registers on the back edges of the loop may force some explicit copies to be inserted inside of the body of the loop. If there is a direct link from one shift register's source to a different shift register's sink, for example, then an explicit copy will have to be inserted along that edge. This does not affect the overall goal of the algorithm, as a copy involving that data structure would have to be performed anyway, either on the back edge or on some forward edge throughout the body of the loop.

5 Experimental Results

In our experimental study, we evaluated our heuristic for in-placeness optimization on two sets of benchmarks. The first set of benchmarks consisted on random program graphs in the form presented in Section 3. The second set of benchmarks consisted of 7 real-world LabVIEW applications. The platform we used for running these experiments was Intel (R), Pentium (R) 4 CPU 2.80 GHz. 2.79 GHz, 504 MB of RAM, running Microsoft window XP.

5.1 Random Graph Benchmarks

Our first set of benchmarks were random program graphs with a varying number of computational nodes. The largest instance in this set contained 75 computational nodes (which corresponds to the set V in the program graph from Section 3). This is a practical limit for what the optimal constraint-based solvers can handle in a reasonable amount of time. Figure 4 compares the percentage difference from the optimal in-placeness result of our heuristic and the NI LabVIEW heuristic on the random graph instances. The optimum results were computed using the Spear constraint solver that internally employs an exact branch-and-bound method to test satisfiability of problem constraints [4].

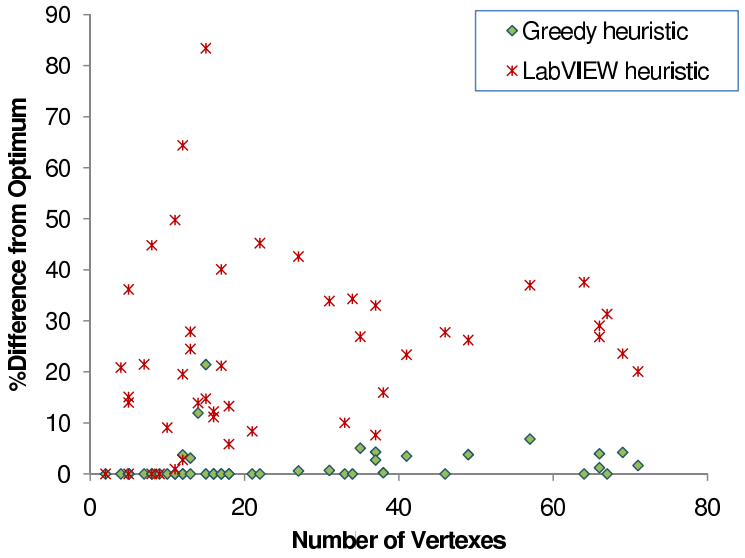

Fig. 4. Graph comparing the optimality gap of the LabVIEW heuristic and our heuristic for in-placeness on random program graphs. Lower is better.

Figure 4 indicates that our greedy heuristic is consistently close to the optimal result across a varying number of vertices in the program graph. On average, our heuristic is within 2% of the optimum for these instances, while the LabVIEW heuristic is over 25% below the optimum. The times taken to complete execution of the current LabVIEW heuristics and our algorithm are both on the order of milliseconds. In contrast, the exact solver method takes up to 3 minutes on some problem instances. Thus, our heuristic achieves a much higher quality of results while matching the efficiency of the NI LabVIEW heuristic for in-placeness.

5.2 LabVIEW Application Benchmarks

Tables 1 and 2 summarize the performance of programs compiled using our heuristic and the NI LabVIEW heuristic on several benchmarks. In four of the benchmarks our algorithm made the same in-placeness decisions as the LabVIEW compiler. The slight differences in the running times are due to different

default schedules produced by the two algorithms. In the other programs, the code generated by our algorithm significantly outperforms the code generated by the current LabVIEW compiler.

	Sample VI	No In-placeness	NI In-placeness	Greedy In-placeness
1	Standard Div	110 ms	32 ms	32 ms
2	Original Unpack	> 8 $hours$	745 ms	733 ms
3	Simple Unpack	> 8 $hours$	695 ms	605 ms
4	Split Unpack	> 8 $hours$	353 ms	358 ms
5	Sine Generator	$2,064,041$ ms	94 ms	62 ms
6.a	Updating Cluster	> 8 $hours$	$132,796$ ms	10 ms
6.b	Tuned Updating Cluster	> 8 $hours$	10 ms	10 ms
7.a	Mandelbrot	$10,478,956$ ms	$44,425$ ms	$6,888$ ms
7.b	Rewired Mandelbrot	$11,937,131$ ms	$42,958$ ms	$6,970$ ms

Table 1. Running times for a set of LabVIEW benchmarks.

	Sample VI	NI In-placeness Vs. No In-placeness	Greedy In-placeness Vs. No In-placeness	Greedy In-placeness Vs. NI In-placeness
1	Standard Div	3	3	1.0
2	Original Unpack	$> 40,000$	$> 40,000$	1.016
3	Simple Unpack	$> 40,000$	$> 50,000$	1.149
4	Split Unpack	$> 80,000$	$> 80,000$	0.99
5	Sine Generator	$21,958$	$33,291$	1.516
6.a	Updating Cluster	> 200	$> 300,000$	$13,280$
6.b	Tuned Updating Cluster	$> 300,000$	$> 300,000$	1.0
7.a	Mandelbrot	236	$1,521$	6.45
7.b	Rewired Mandelbrot	278	$1,713$	6.16

Table 2. Speedup factors for the set of LabVIEW benchmarks from Table 1.

The results presented in Tables 1 and 2 illustrate the importance of having a systematic in-place computation strategy in a LabVIEW implementation. For all the above programs, the difference in running times between no in-place computation and some in-place computation is enormous. Updating Cluster, for example, takes 10 milliseconds to compute using our in-place algorithm but it takes more than 8 hours (we terminate our experiments after 8 hours for practical reasons) with no in-place computation.

In summary, we have shown on a large collection of random graphs that our algorithm consistently finds a better solution, and on a collection of real-world LabVIEW programs it at least matches, and in several cases significantly outperforms the current LabVIEW compiler.

6 Related Work

Hudak and Bloss address the problem of updating aggregates in-place to avoid unnecessary copies in functional languages [5]. If the last use of an aggregate is a write, they perform the write in place. Our algorithm checks the legality of allowing a use to be done in-place before adding constraints, and uses a cost

model to to in-place the uses that will be most beneficial to performance. Their algorithm relies on run-time reference counting to find the inplaceness opportunities, while we make all the decisions statically. In [6], Goyal and Paige gave a solution that combines dynamic reference counting and lazy copying. Their algorithm uses static analysis to enhance and improve the use of reference counting. They implement the optimization for the programming language SETL.

For fixed evaluation order, Bloss developed an algorithm that statically computes evaluation paths in non-strict functional programs [7], while Kirkham and Li developed a copy avoidance algorithm to improve the performance of the programming language *UFO* [8] and Gudjónsson and Winsborough developed an algorithm that introduces update-in-place operations to the logic programming language Prolog to allow it to update recursive data structures as in an imperative programming language [9]. These heuristics, as in the first step of Hudak and Bloss [5], update an aggregate in place if the last use is a write operation.

Sarkar and Cann built an optimized SISAL compiler that includes an update-in-place analysis phase that tackles the aggregate incremental update problem by extending the approach by Hudak and Bloss to additionally consider general iteration, function call boundaries, and nested aggregates [10]. However, as with Hudak and Bloss, they do not evaluate the benefits in choosing which nodes to modify in-place, and cannot make all the decisions during compilation.

Gopinath and Hennessy proposed "Targeting", an algorithm to reduce intermediate copies in divide and conquer problems [11] by properly selecting a storage area for expression evaluatiion. They eliminate copies in a given and fixed computing evaluation sequence. We allow changes to the evaluation sequence in order to find more opportunities for in-place updating. Unlike our heuristic, their algorithm constrains the arrays to have restricted bounds, and is unable to detect values whose lifetime cannot be computed at compile time.

Debray focuses on reusing dead data structures [1]. As in our algorithm, his heuristic chooses between interfering data structures based on a cost model, but does not describe how to derive these costs.

The problem of excessive copying appears in register allocation. The second stage of the Chaitin's [12] algorithm consists of coalescing nodes in the graph to use the same storage (machine register). If there is a copy from R_i to R_j, and R_i and R_j do not otherwise interfere, then R_i and R_j can share storage. Briggs *et al.* add a tradeoff between coalescing and spilling register values to memory [13]. In another related algorithm, Briggs *et al.* present an algorithm for inserting copies to replace ϕ-functions when translating SSA form to sequential code [14], which involves a similar problem with cycles of copies. lić *et al.* present an algorithm that performs coalescing without building an interference graph, but by using liveness and dominance information to model interference [15].

While copy avoidance problems are closely related to the one we are addressing in this paper, none of the results described above focus on a unique characteristic of the programs with aggregate data structures: it is much more profitable to perform an in-place computation on a data structure when such a computation only changes a small part of the data.

7 Conclusions and Future Work

Copy avoidance through in-place computation is extremely important for languages with copy semantics such as LabVIEW. The performance of LabVIEW programs with a methodical in-place computation strategy can improve by several orders of magnitude over programs with naïve implementations that allocate a new memory location for the result of every computation. This is especially true for programs with loops and aggregate (array and structure) data.

In this paper, we present a systematic greedy algorithm for deciding which computations should be performed in-place for LabVIEW programs. We show that our algorithm runs in $O(TlogT + E_W V + V^2)$ time, where T is the number of in-placeness opportunities, E_W is the aggregate number of edges and V is the number of computational nodes in a program graph.

Our heuristic computes near-optimum (within 2% on average) solutions for a large collection of randomly generated graphs, compared to the current LabVIEW compiler heuristic which is more than 25% below the optimum. Our algorithm achieves this while still running in time competitive to the current LabVIEW compiler (order of milliseconds for random graphs of up to 75 nodes). It is much faster than optimal constraint-based solver strategies, which are impractical even for modestly large programs.

On a collection of LabVIEW programs, our algorithm produces in-placeness decisions that generate code that is at least competitive, and in several cases much faster than the code generated by the current LabVIEW compiler. This efficient and effective in-place computation strategy should prove itself a valuable addition to any implementation of languages with copy semantics.

In the future, we will investigate an adaptation of our algorithm to an interprocedural, modular compiler using the Telescoping Languages approach [16], which includes a size-inference algorithm that infers sizes of procedure variables in terms of the sizes of input arguments. This information will be important for determining benefits for in-placeness when the whole program is not available. This approach will summarize the in-placeness analysis results for each program and create different versions of programs based on different in-placeness contexts. Second, we will experiment with algorithms for splitting of aggregate data structures and in-place computation for parts of an aggregate data structure. Finally, we will explore heuristics for estimating the trade-off between more in-place computation and more parallelism available (especially for multicore platforms) that will attempt to balance the in-place computation, parallelism and scheduling to achieve faster running times.

8 Acknowledgments

This work was supported in part by National Instruments and by NSF grant CCF-0444465. We would like to thank Jeff Kodosky, Brent Schwan and Duncan Hudson for their comments and support during this project. We would also like to thank Keith Cooper and Tim Harvey for their insights and discussions

concerning relevant register allocation topics, and Vivek Sarkar for his assistance in understanding copy elimination in SISAL and other data-flow languages.

References

1. Debray, S.K.: On copy avoidance in single assignment languages. In: International Conference on Logic Programming. (1993) 393–407
2. Abu-Mahmeed, S., McCosh, C., Budimlić, Z., Kennedy, K., Ravindran, K., Hogan, K., Austin, P., Rogers, S., Kornerup, J.: Scheduling tasks to maximize usage of aggregate variables in place. Technical report, Rice University, TR09-01 (2009)
3. National Instruments Corporation: LabVIEW^{TM}User Manual. (August 2007)
4. Babic, D., Hutter, F.: Spear theorem prover. In: SAT 2007 Competition. (Jan 2007)
5. Hudak, P., Bloss, A.: The aggregate update problem in functional programming systems. In: POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. (1984)
6. Goyal, D., Paige, R.: A new solution to the hidden copy problem. In: Proceedings of the 5th International Static Analysis Symposium, volume 1503 of Lecture Notes in Computer Science, Springer-Verlag (1998) 327–348
7. Bloss, A.: Update analysis and the efficient implementation of functional aggregates. In: FPCA '89: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. (1989)
8. Li, Z., Kirkham, C.: Efficient implementation of aggregates in united functions and objects. In: ACM-SE 33: Proceedings of the 33rd Annual Southeast Regional Conference. (1995)
9. Gudjónsson, G., Winsborough, W.H.: Compile-time memory reuse in logic programming languages through update in place. ACM Trans. Program. Lang. Syst (1999)
10. Sarkar, V., Cann, D.: Posc–a partitioning and optimizing sisal compiler. In: ICS '90: Proceedings of the 4th international conference on Supercomputing, New York, NY, USA, ACM (1990) 148–164
11. Gopinath, K., Hennessy, J.L.: Copy elimination in functional languages. Technical report, Computer Systems Laboratory, CSL-TR-88-370. Stanford, CA: Stanford University. (1989)
12. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer Languages **6** (1981) 47–57
13. Briggs, P., Cooper, K.D., Torczon, L.: Rematerialization. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (17-19 June 1992) 311–321
14. Briggs, P., Cooper, K.D., Harvey, T.J., Taylor Simpson, L.: Practical improvements to the construction and destruction of static single assignment form. Software Practice and Experience (July 1998)
15. Budimlic, Z., Cooper, K.D., Harvey, T.J., Kennedy, K., Oberg, T.S., Reeves, S.W.: Fast copy coalescing and live-range identification. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. (2002)
16. Kennedy, K., Broom, B., Chauhan, A., Fowler, R., Garvin, J., Koelbel, C., McCosh, C., Mellor-Crummey, J.: Telescoping languages: a system for automatic generation of domain languages. Proceedings of the IEEE **93**(2) (Feb 2005) 387–408