Construction of Thinned Gated Single-Assignment Form*

Paul Havlak

Center for Research on Parallel Computation,
Rice University, Houston TX 77251-1892 USA

Abstract. Analysis of symbolic expressions benefits from a suitable program representation. We show how to build thinned gated single-assignment (TGSA) form, a value-oriented program representation which is more complete than standard SSA form, defined on all reducible programs, and better for representing symbolic expressions than program dependence graphs or original GSA form. We present practical algorithms for constructing thinned GSA form from the control dependence graph and SSA form. Extensive experiments on large Fortran programs show these methods to take linear time and space in practice. Our implementation of value numbering on TGSA form drives scalar symbolic analysis in the ParaScope programming environment.

1 Introduction

Analysis of non-constant values yields significant benefits in a parallelizing compiler. The design of a symbolic analyzer requires careful choice of a program representation, that these benefits may be gained without using excessive time and space.

In symbolic analysis, we need to represent and compare both constant and non-constant expressions. This aids dependence testing, when we compare array subscripts; and dead code elimination, when we try to evaluate comparisons for branches. If our expression representation is self-contained, we can re-use the symbolic analysis system in multiple contexts, independent of other program representations or their absence.

Merges of program execution paths complicate the construction of symbolic expressions. In straight-line or branching code, we can simply replace each reference to a variable with the expression assigned at the unique reaching definition of that variable. This process of forward substitution yields a complete expression, in the usual form, with constants or external inputs at the leaves. Introduce merging, and we must find a way to represent the combination of values from multiple, conditionally executed assignments, or else give up on these cases.

* This work was supported in part by ARPA contract DABT63-92-C-0038 and NSF Cooperative Agreement No. CCR-9120008. Use of a Sun Microsystems Sparc 10 was provided under NSF Cooperative Agreement No. CDA-8619893. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Consider the problem of representing the value of `I` printed by the program in Figure 1.[2] The control flow graph (G_{CF}) provides complete information, but is inconveniently organized. We have the choice of running the program (if the input is available) to see the output, or of augmenting or replacing G_{CF} with another program form.

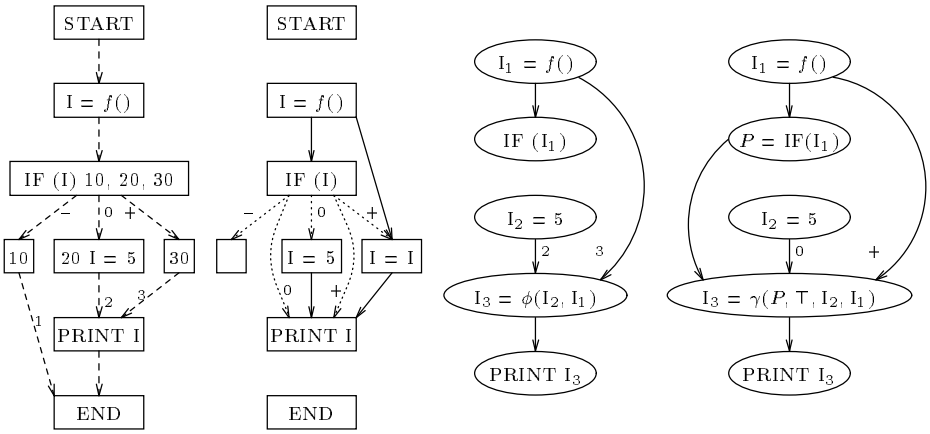

Fig. 1. Program Representations

One candidate representation is the program dependence graph (PDG). There are actually several varieties of PDG used in compilers and other program transformation systems [14, 18]. The PDG variant shown in Figure 1 was introduced by Cartwright and Felleisen [7]. The solid arrows represent the flow of data and the dotted arrows execution decisions.

This PDG may be interpreted by allowing each node to execute when it receives a control token from a control-dependence predecessor and for each variable reference, a data token from a data-dependence predecessor [7]. Assignments distribute data tokens to all their successors, but each branch sends control tokens only on edges whose labels correspond to the value of the predicate.[3]

While the PDG is a complete replacement for G_{CF}, its structure clashes with the requirements for a symbolic expression representation. It is far from clear how to write an expression for the merge that produces the printed value of `I`. And though the merge and the branch that controls it are close together in this example, in general they can be arbitrarily far apart in the PDG.

[2] We use the subscripted name $\mathtt{V}_i$ to indicate definition i of variable `V` or the value of that definition.

[3] We generalize the traditionally Boolean notion of predicate to include the multiway tests common in Fortran and C; *e.g.*, the branch in Figure 1 depending on `I`'s being negative ($-$), zero, or positive ($+$).

A second possibility is static single assignment (SSA) form [3]. In Figure 1, we give the def-use chains built on the SSA form of the program, where *pseudo-assignments* have been added at merges of values. $\mathtt{I}_3 = \phi(\mathtt{I}_2, \mathtt{I}_1)$ looks like a nice expression for the merge. However, the ϕ function is not referentially transparent; its result depends not only on its inputs but also on which control flow edge (2 or 3) executes. Were we to encounter a ϕ function with the same inputs elsewhere in the program, we could not assume their equivalence.

Gated single-assignment (GSA) form solves these problems by extending SSA form with new functions that encode control over value merges. GSA form was introduced as part of the program dependence web, and descends from the high-level variant of SSA form (defined only for structured code) [5, 3].

In Figure 1, the γ function takes the predicate controlling the merge as an explicit input. Depending on whether $\mathtt{I}_1$ is zero or positive, the new value for $\mathtt{I}_3$ is $\mathtt{I}_2$ (5) or $\mathtt{I}_1$, respectively. If $\mathtt{I}_1$ is negative, then $\mathtt{I}_3$ represents an unexecutable expression, denoted by $\top$. We can treat a $\top$ value as equivalent to any other when it executes, because it never does.

Handling of merges with referentially transparent functions and the resulting independence from G_{CF} together make GSA form an adequate representation for symbolic expressions. However, because original GSA form was designed with other purposes in mind, it contains some control information that we consider extraneous to symbolic analysis. We introduce thinned gated single-assignment (TGSA) form with the extra information deleted.[4] We show how to build TGSA form using SSA form and the unfactored control dependence graph.

Section 2 describes the control and data-flow information needed to build TGSA form. Thinned GSA form is defined in Section 3. The algorithms given in Section 4, for building TGSA form, are shown in Section 5 to require linear time and space under reasonable assumptions and throughout extensive experiments. Section 6 describes current and future uses for TGSA form, Section 7 covers related work, and and we conclude in Section 8.

2 Background and Notation

Building TGSA form for a procedure requires its control flow graph, control dependences and def-use chains for SSA form. This section describes the preliminary graphs and our notation.

2.1 Control Flow

The nodes of the control-flow graph G_{CF} are basic blocks, each containing zero or more statements in straight-line sequence. Control flow can branch or merge only between basic blocks; wherever execution can flow from one block to another, a

[4] Our thinning consists of using fewer functions to replace a given ϕ than original GSA form. It should not be confused with *pruning*, which refers to the elimination of dead ϕ assignments in SSA form—those that reach no uses [8].

G_{CF} edge goes from the former block to the latter. A unique START node has no in-edges and an out-edge to every procedure entry; a unique END node has no out-edges and an in-edge from every procedure exit.

A node with multiple out-edges is a *branch*, with each out-edge labeled by the corresponding value of the branch predicate. A node with multiple in-edges is a *merge*, and its inedges are ordered.

From this graph we derive trees for dominator and loop nesting relationships. The immediate predominator of a block B, denoted $B.Ipredom$, is the last of the nodes ($\neq B$) that lie on every path from START to B. B's immediate postdominator, $B.Ipostdom$, is the first of the nodes ($\neq B$) lying on every path from B to END [21]. $B.Header$ lies on every path into and around the most deeply nested loop containing B.[5]

For some purposes we ignore back-edges (from a node to the header of a surrounding loop), leaving us with the forward control-flow graph F_{CF}. In this paper, *structured* applies to code for which F_{CF} has the fork-join property: each merge is the immediate postdominator of exactly one branch *and* all paths from a branch remain distinct until they join at the merge which immediately postdominates the branch. (Note that C-style `break` and `continue`, as well as `goto`s, can produce unstructured code.)

2.2 Control Dependence

The control dependence graph G_{CD} consists of the basic blocks from G_{CF} with a different set of edges.[6] Given a G_{CF} edge e with label L from a branch node B, control dependence edges, also labeled L, go from B to every node that must execute if e is taken [12].

Loop-carried control dependence edges run from conditional branches out of a loop to the header block of the loop. Ignoring these edges leaves the acyclic forward control-dependence graph F_{CD} [11].

2.3 Static Single-Assignment (SSA) Form

We focus on SSA form as an improved version of def-use chains. The nodes of the SSA-form def-use graph G_{DU}^{SSA} are the original program references plus pseudo-assignments, called ϕ nodes.

SSA construction entails adding the ϕ nodes at the beginning of merge blocks, so that only they, and no original references, are reached by multiple definitions [12]. The source and sink of each def-use chain (G_{DU}^{SSA} edge) are subscripted in examples with the definition number of the source.

Each chain e into a ϕ node is also tagged with the last control flow edge ($e.CfEdge$) by which its definition reaches the merge. The ϕ serves as a merge

[5] Our loops are Tarjan intervals (nested strongly-connected regions), computed with minor extensions to Tarjan's test for flow-graph reducibility [24]. Irreducible loops are handled conservatively.

[6] The original presentation of GSA form uses a factored control dependence graph with added region nodes [5, 14].

function, as in Figure 1, denoting that if G_{CF} edge 3 into the merge executes, then $\mathtt{I}_3 = \phi(\mathtt{I}_2, \mathtt{I}_1)$ assumes the value from definition $\mathtt{I}_1$.

Loop-carried G_{DU}^{SSA} edges go from a definitions inside a loop to a ϕ node at the loop header. Ignoring these edges leaves the forward def-use graph F_{DU}^{SSA}. Each node n in the graph is annotated with the basic block (G_{CF} node) containing it, $n.Block$.

3 Definition of TGSA Form

Alpern *et al.* introduced the first gated version of static single-assignment form as high-level SSA form [3]. Ballance *et al.* introduced the terminology of gated single-assignment form [5]. TGSA form is an extension of high-level SSA form to unstructured code, for which we use the more convenient GSA-form notation. Detailed comparisons with the prior versions are given in Section 7.

Building TGSA form involves adding pseudo-assignments for a variable V:

(γ) at a control-flow merge when disjoint paths from a conditional branch come together and at least one of the paths contains a definition of V;
(μ) at the header of each loop that contains at least one definition of V; and
(η) at every exit from each loop that contains at least one definition of V.

The first two cases, merges in forward control flow and at loop entry, are handled with ϕ nodes in SSA form. The third case, merging iterative values at loop exit, is ignored by SSA form.

3.1 Structure

Gamma: Merge with Predicate. The pseudo-assignment inserted for a data-flow merge in forward control flow employs a γ function. In a simple case, the fragment

$V_1 := ...$
if (P) **then** $V_2 := ...$

is followed by $V_3 := \gamma(P, V_2, V_1)$. This indicates that if control flow reaches the merge and P was true, then V has the value from definition V_2.

We insert γ functions only to replace ϕ functions at merges of forward control flow. Whenever otherwise disjoint F_{CF} paths from an n-ary conditional branch (with predicate P) come to a merge, and at least one path redefines a variable V, we insert a definition $V' := \gamma(P, V_1, ..., V_n)$ at the merge point. Each of the V_i is the last definition (possibly another pseudo-assignment) occurring on a path from subprogram entry, through the branch, to the merge. The V_i are called the value inputs, P the predicate input. A γ is strict in its predicate input only; when the predicate has value i, the quantity from the i^{th} value input is produced.

In unstructured code, paths from multiple branches may come together at a single merge. Replacing the ϕ then requires multiple γ functions, explicitly

arranged in a dag. Such a γ dag is structured so that the immediate predominator of the merge provides the predicate for the root of the dag, and paths in the dag from the root to the leaves pass through γ functions with predicates from branches in the same order that F_{CF} paths from the immediate predominator to the merge pass through the branches themselves.

If some edges from a branch cannot reach a merge, then the corresponding locations in γ functions are given as $\top$, indicating that no value is provided. We disallow the case where all non-predicate arguments save one are $\top$, as in $\gamma(P, \top, ..., \top, V_i, \top, ..., \top)$ — if execution reaches the merge, then definition V_i must be providing the value. This simplification is one difference between *thinned* GSA form and original GSA form, which we exploit in our algorithms for building TGSA form.

Mu: Loop Merge. A pseudo-assignment at the header of a loop uses a μ function. For each variable V defined in a loop body, we insert a definition $V' := \mu(V_{init}, V_{iter})$ at the loop header, where V_{init} is the *initial input* reaching the header from outside ($\mathtt{I}_0$ in Figure 2) and V_{iter} is the *iterative input*, reaching along the back-edge ($\mathtt{I}_2$). If every loop header has one entry edge and one back-edge, then every ϕ at a loop header can be replaced exactly by one μ (otherwise, a γ dag may also be required to assemble one or both of the two inputs).

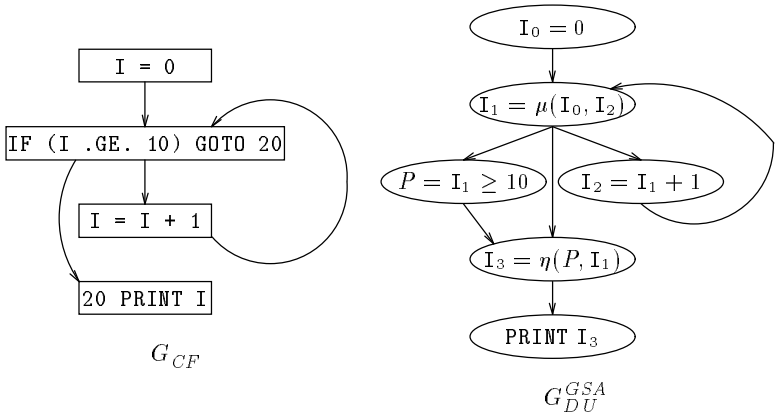

Fig. 2. Loop Representations

Since TGSA form is organized for demand-driven interpretation, there is no control over values flowing around a loop. A μ function can be viewed as producing an infinite sequence of values, one of which is selected by an η function at loop exit. (In original GSA form, μ functions have a predicate argument controlling iteration.)

Eta: Loop Value Selection. Pseudo-assignments inserted to govern values produced by loops use η functions. Given a loop that terminates when the predicate P is true, we split any def-use edge exiting the loop and insert a node $V' := \eta(P, V_{final})$, where V_{final} is the definition ($\mathtt{I}_1$ in Figure 2) reaching beyond the loop. In general, a control-flow edge may exit many loops at once; for a variable that has been modified in the outermost L loops exited, we must insert a list of L η functions at the sink of the edge. The η function in TGSA form corresponds to the η^T function in original GSA form.

Loop-variance Level. The *level* of a node in G_{DU}^{GSA} is the nesting depth of the innermost loop with which it varies (or zero, for loop-invariant values). The level of a μ function equals the depth of the loop containing it; the depth of an η function equals the depth of the loop exited less 1. For any other function, the level is the maximum level over all its inputs.

Level information is required to distinguish μ functions that vary with different loops in a nest. For other nodes, level information is just a convenient annotation.

3.2 Interpretation

We give no formal semantics for the interpretation of TGSA form, but define a structural notion of congruence. Under certain conditions, congruent nodes can be assumed to have equivalent values.

Congruence. Structural congruence is a form of graph isomorphism, defined on a value graph which combines G_{DU}^{GSA} with expression trees [3]. Initially, we can view the value graph as copied from these two forms, so that values flow from definitions to uses through G_{DU}^{GSA} edges and from uses to definitions through expression trees.

Collapsing uses of variables with their reaching definitions is the equivalent of forward substitution. When this is done, the value graph is an expression dag or forest, as in [3].[7] Leaf nodes are constants and external inputs, and internal nodes are program operations (*e.g.*, arithmetic) and TGSA functions.

Leaf nodes are congruent if they are the same; other value graph nodes are structurally congruent if they have the same function (remembering to distinguish μ functions with different levels) and congruent inputs.[8] Program expressions are called congruent if their corresponding value graph nodes are congruent.

[7] In our implementation, G_{DU}^{GSA} is attached to G_{CF}. We also keep a map from each G_{DU}^{GSA} node to its corresponding value graph node.

[8] Two methods of discovering congruence are value numbering by hashing [1] and partitioning [3]. Partitioning can discover congruent cycles that hashing cannot; however, hashing is better suited to arithmetic simplification and lazy evaluation. We use hashing in our implementation, with extensions for inductive value recognition.

Equivalence. TGSA form is designed so that congruence of expressions can be detected regardless of their execution context. However, care is needed in inferring equivalence of expressions from congruence of their corresponding value graph nodes.

In loop-free code, every expression executes once or not at all. Congruent expressions have the same value if both execute. In code with loops, congruent loop-variant expressions with level L can be treated as equivalent only when they both execute for the same iteration of their outer L surrounding loops. Valid applications of this principle include:

- Common subexpression elimination: If two level-L program expressions are congruent and have their outermost L loops in common, L surrounding loops in common, then they assume the same value on the each iteration of the common loops. If, for example, the G_{CF} block containing one expression predominates that containing the other, we may save the result of the first expression to replace the second.
- Test elision: If congruent subexpressions are tested for equality, then they execute in synchrony and the test must always yield **true**. The **false** G_{CF} edges from the branch may be deleted, along with any code that becomes unreachable.
- Dependence testing: when analyzing subscripts to test for dependence carried at level L, assume that all loops shallower than L are frozen at some arbitrary iteration. Congruent subexpressions with level less than L may be treated safely as equivalent.

4 Construction of TGSA Form

We present algorithms for the two crucial steps in converting from SSA form to TGSA form: replacing a ϕ function with a directed acyclic graph of γ functions and building the controlling predicate of a loop for use in its η functions.

4.1 Augmenting G_{CF} and G_{DU}^{SSA}

For convenience, we assume that every G_{CF} node is reachable from START and reaches END (neither unreachable nor dead-end code exists). Loops are assumed reducible, so that all cycles pass through a unique header. (In the implementation, irreducible loops are conservatively left alone.) We build tree representations of loop nesting and of the pre- and post-dominator relations [21, 24]. By saving a preorder numbering of each tree, we can test for transitive as well as immediate loop nesting and dominance in constant time.

We augment loops, as shown in Figure 3, to provide suitable places for the placement of μ and η nodes. For each loop we add a preheader node (`PH`), if the header has multiple forward in-edges, and a postbody node (`PB`), if the header has multiple backward inedges or if the source of the back-edge is in an inner loop. The postbody node for each loop then terminates each complete iteration,

and is post-dominated by the header. Wherever the sink of a loop-exit edge has other in-edges, we split the loop-exit edge with the addition of a postexit node (`PE`) [11]. These changes cause threefold growth in G_{CF} at worst.

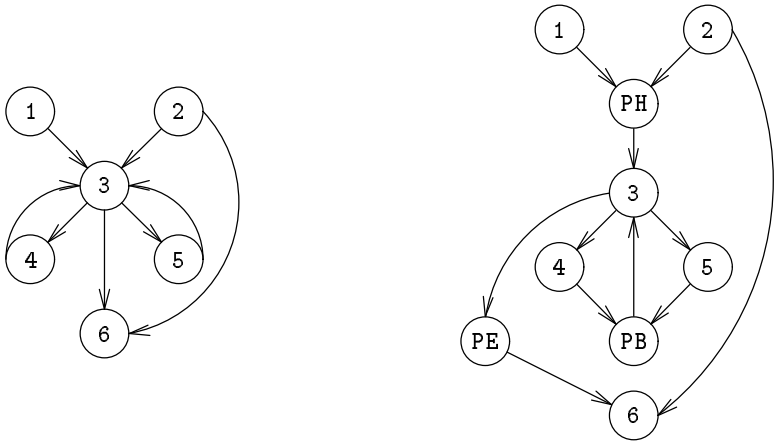

Fig. 3. Insertion of Preheader, Postbody and Postexit Nodes

The dominance and loop nesting trees are easily updated, and control dependence construction proceeds normally on the augmented G_{CF} [10]. As none of the new nodes are branches, control dependence edges among the original G_{CF} nodes are unaffected.

Construction of SSA form proceeds normally except for tweaks to the ϕ placement phase. Having ensured that each loop header has exactly two inedges, we place μ nodes at loop headers instead of ϕ nodes. Every μ node added represents a variable modified in the loop, and we immediately add an $\breve{\eta}$ definition for the same variable at the postexit node for the loop.[9] These trivial pseudo-assignments ($V' := \breve{\eta}(V)$) hold the place for later creation of η functions with predicates, and ensure proper placement of ϕ assignments for merges of values from different exits.

Once ϕ, μ, and $\breve{\eta}$ assignments have been placed, the linking of definitions to uses (often referred to as *renaming* in the SSA literature) proceeds as for standard SSA form, producing an variant of G_{DU}^{SSA}.

4.2 Data Structures

With μ and $\breve{\eta}$ assignments out of the way, the main work of building TGSA form lies in converting the remaining ϕ functions to dags of γ functions, and building

[9] If the postexit node is shared with a surrounding loop, we add nothing, as the appropriate $\breve{\eta}$ will be added for the outer loop.

other γ dags to represent loop-termination conditions.[10] Our solutions for these similar problems share many data structures, described below.

The procedure *replace_ϕs()*, shown in Figure 4, examines one ϕ function at a time and replaces it with a dag of γ functions. The γ dag has the same dataflow predecessors and successors as the original ϕ, except that each γ function also takes the result of a branch predicate as input. All of the γ functions in the dag replacing a ϕ are placed at the beginning of the basic block where the merge happens (ϕ*.Block*).

Likewise, *build_loop_predicate()* proceeds one loop at a time. The γ functions making up the dag for the loop-termination condition are placed at the postbody for the loop.

Branch.Choices: for a G_{CF} node *Branch*, a vector of one choice per out-edge. Each choice represents a value that will reach the merge if the branch goes the right way. A choice is either a leaf definition (pointing to a value computed at another G_{CF} node), another branch (representing the value chosen there), or $\top$ (indicating a choice to avoid the merge node).

In ϕ-replacement, a leaf definition is the choice if all the paths through the corresponding out-edge to the ϕ merge point pass through the same final definition. This chosen definition may occur before or after *Branch*. If there are instead multiple final definitions on these paths, then the branch choosing between them is specified as the choice for this out-edge. Such a branch must lie between *Branch* and the ϕ merge point. If no path through an out-edge reaches the ϕ merge point, then its corresponding choice entry is $\top$.

When building loop predicates, a leaf choice is **true** if an outedge must lead to the postbody or **false** if it cannot. If some paths through the outedge lead to the postbody and some do not, then a branch choice involved in the decision is specified.

Selectors: a set of G_{CF} branch nodes whose predicates will be used in building the γ dag for the current loop predicate or ϕ.

In ϕ-replacement, a branch is added to *Selectors* if and only if it affects, directly or indirectly, the choice of definition reaching the control flow merge. We add the branch to *Selectors* when we detect that two of its *Choices* are different and not $\top$. More precisely, the selectors of a ϕ assignment comprise the set of $n \in N_{CF}$ such that (1) $\exists$ at least two paths in F_{CF}, from n to ϕ*.Block*, disjoint except for their endpoints, and (2) $\exists$ such a path containing a definition of the ϕ's variable not at n or at ϕ*.Block*.

When building loop predicates, *Selectors* contains all branches inside the loop whose immediate postdominator is outside the loop (i.e., the nodes within the loop on which the postbody is transitively control dependent).

Branch.FirstChoice: used only in *replace_ϕs()*; it saves the first choice propagated back to the branch. Each choice propagated back to *Branch* is compared against this field. If the *FirstChoice* field already holds a different

[10] These algorithms are described in terms of building G_{DU}^{GSA}; they also can be used to build the GSA-form value graph on the fly.

```
procedure replace_φs(Merge):
    Predom := Merge•Ipredom;
    foreach φ at Merge do
        foreach CfEdge entering Merge in F_CF do init_walk(CfEdge•Source);
        Selectors := {};
        foreach DefEdge entering φ in F_DU do
            CfEdge := DefEdge•CfEdge;    Def := DefEdge•Source;
            process_choice(Def, CfEdge•Label, CfEdge•Source);
        recursively build a dag γ from Predom.Choices;
        replace φ with γ;

procedure init_walk(Node):
    if (Node•Fanout > 1) then
        Node•Visits++;
        if (Node•Visits == 1) then Node•Choices[*] := Node•FirstChoice := ⊤;
        if (Node == Predom) or (Node•Visits ≠ 1) then return;
    /* on our first visit to non-Predom */
    foreach edge Cd entering Node in F_CD do init_walk(Cd•Source);

procedure process_choice(Choice, Label, Node):
    NewChoice := Choice;
    if (Node•Fanout > 1) then
        Node•Choices[Label] := Choice;    Node•Visits--;
        if (Node•FirstChoice == ⊤) then Node•FirstChoice := Choice;
                                   else if (Node•FirstChoice ≠ Choice) then
                                            Selectors += Node;
        if (Node == Predom) or (Node•Visits ≠ 0) then return;
        if (Node ∈ Selectors) or
          ((not Thinned) and (Merge ∉ postdominators(Node))) then
            NewChoice := Node;
    /* on our last visit to non-Predom */
    foreach edge Cd entering Node in F_CD do
        process_choice(NewChoice, Cd•Label, Cd•Source);
```

Fig. 4. Replacing a ϕ with a dag of γs

value besides $\top$, then another choice has already been propagated, and we add *Branch* to *Selectors*.

Node•Visits: a counter, used only in replacing a ϕ. Initialized to zero, the multiple calls to *init_walk* increment *Visits* for branch nodes until it reaches the number of edges from branch that can lead to the merge. Later, calls to *process_choice* decrement *Visits* each time an edge from the branch node is processed. When *Visits* reaches zero again, then all paths from the branch to the merge have been examined. We then propagate the appropriate choice to the branch's G_{CD} predecessors: the branch itself, if it is a selector, else the one non-$\top$ value from its choices.

CfNode•Exits This is the γ dag representing when the loop exits. While we could compute distinct conditions for each exit, it is sufficient to determine termination test for the entire loop.

Miscellaneous:

CfNode•Ipredom = immediate predominator;
SsaDuEdge•CfEdge = final G_{CF} edge along which a definition reaches a ϕ;
CfEdge•Label distinguishes edges sourced in the same conditional branch;
CfNode•FanOut = the number of outedges from a G_{CF} node.
Thinned is true for all examples given here; when false, extra $\top$ entries are left in γ functions.

4.3 γ Conversion

The routines in Figure 4 show how to replace a ϕ function with a dag of γ functions. We call *replace_ϕs()* for each node in topological order on F_{CF}, and it replaces any ϕ functions located there. The replacement proceeds one ϕ at a time. Auxiliary routines visit each control-dependence ancestor of F_{CF} predecessors of ϕ•*Block* in two passes. Consider their effects during *replace_ϕs(*`A3`*)* for the only ϕ assignment in Figure 5.

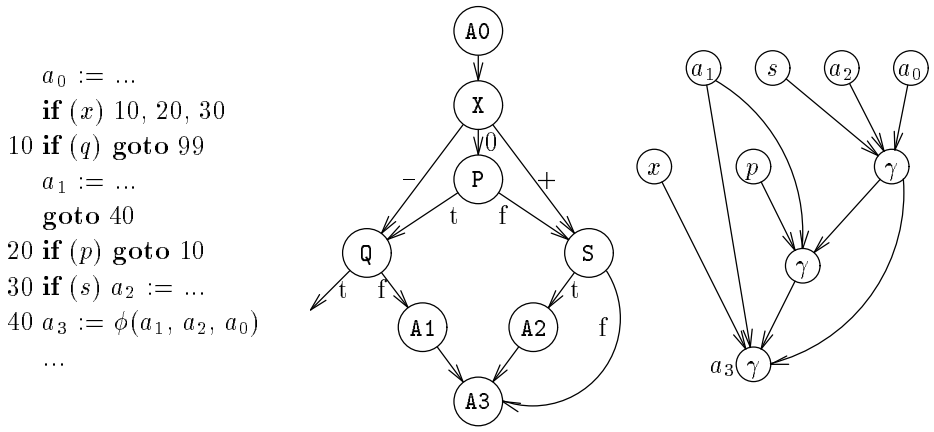

Fig. 5. SSA-form Source, G_{CF}, and γ dag for Example

The first pass, by recursive calls to *init_walk()*, initializes fields for each node that might be a branch point for distinct paths to the merge. *Node•Visits* counts the number of visits in this walk; we will count back down on the next walk. At the end of this pass, the values for *Visits* in the example are: 1 for `Q` (because the other outedge cannot reach `A3`), 2 for `S`, 2 for `P`, and 3 for `X`. All choice vectors and *FirstChoice* fields are $\top$.

The second walk routine, *process_choice()*, propagates each reaching definition backwards from the ϕ until a branch is encountered. It is then entered as

a choice for that branch. If another, different choice has already been entered (*Node*•*FirstChoice*), then the branch *Node* is added to *Selectors* for this ϕ. When we have finished all visits to a branch, we then push a choice (the branch itself, if it has been marked a selector) upwards from this branch to its controlling branch.

If we process G_{DU} edge (a_1, a_3) first, we execute *process_choice*(a_1, **nil**, A1). As this recurses, we decrement Q•*Visits* to 0, and continue propagating a_1 as the choice, since that was Q's first and only choice. When we return to the top level, we have P•*Choices* $= [a_1, \top]$ with one visit left, and X•*Choices* $= [a_1, \top, \top]$ with two visits left. (However, *Visits* is not always the same as the number of $\top$ choices left.)

If we next process G_{DU} edge (a_2, a_3), we execute *process_choice*(a_2, **nil**, A2). We then call *process_choice*(a_2, **true**, S). We return having S•*Choices* $= [a_2, \top]$, with one visit one visit left.

The last edge processed at the top level is (a_0, a_3), and causes us to call *process_choice*(a_0, **true**, S). This finishes off S's visits and makes it a selector, and finishes off P and X directly. We are left with *Selectors* $= \{\mathtt{S}, \mathtt{P}, \mathtt{X}\}$, with their choice vectors $[a_2, a_0]$, $[a_1, \mathtt{S}]$, and $[a_1, \mathtt{P}, \mathtt{S}]$, respectively.

When the second walk is done, *Node*•*Choices* is fully defined for each selector. We then read off the γ dag starting at *Predom*. The γ function at the root of the dag takes its predicate input from the predicate controlling *Predom*'s branching, and its value inputs from the corresponding entries in the choice vector. Choices that are leaf definitions are made direct data inputs to the γ; a choice that is a selector is replaced with a γ function built recursively, in the same fashion, from the predicate and choice vector for the selector. The resulting γ dag for a_3 is shown rightmost in Figure 5.

We give no formal proof for the correctness of this procedure. Informally, one can easily show that *Selectors* is properly computed. To show correctness of the γ dags, we prove a one-to-one correspondence between paths in the γ dag replacing a ϕ and sequences of selectors encountered in execution paths reaching ϕ•*Block* in G_{CF}, based on the labels produced by the predicates at the selectors. The last definition on a G_{CF} path from START to ϕ•*Block* is the same as the leaf definition selected by the corresponding path through the γ dag.

4.4 η Construction

Figure 6 shows the procedures to build the γ dag for a loop-termination predicate. For each loop header, we pass its unique back-edge predecessor (its post-body node) to *build_loop_predicate()*. The γ dag built represents the negation of its control conditions, which correspond to the conditions for loop exit. We store this condition as *Loop*•*Exits*.

We now expand each $\check{\eta}$ — inserted earlier, one per loop-exit G_{CF} edge — so that there is one η per loop exited, and make the predicate input of each η refer to the loop-termination predicate for the corresponding loop. Each η takes input from the η from the next-inner loop exited on the same G_{CF} edge, and

```
procedure build_loop_predicate(End):
    Loop := End•Header;    Selectors := {};
    process_cds(End);
    Predom := Loop;
    while (Predom•Ipostdom predominates End) do Predom := Predom•Ipostdom;
    recursively build a dag from Predom.Choices;
    save negated dag as Loop•Exits;

procedure process_cds(Node):
    foreach edge Cd entering Node in F_CD do
        Pred := Cd•Source;
        if (Pred == Loop) or (Loop predominates Pred) then
            if (Pred ∉ Selectors) then /* assume all branches exit */
                Pred•Choices[*] := true;
                Selectors += Pred;
                process_cds(Pred);
            if (Node == End) then Pred•Choices[Cd•Label] := false; /* non-exit */
                             else /* possible exit path */
                                  Pred•Choices[Cd•Label] := Node;
```

Fig. 6. Building a loop-termination predicate

feeds the η for the next-outer loop exited, except that the innermost η gets the input of the original $\breve{\eta}$ and the outermost η feeds the original uses of the $\breve{\eta}$.

5 Theoretical and Experimental Complexity

We implemented these algorithms in the ParaScope programming environment [20] and tested them on 1227 procedures from 33 programs in the NAS, Perfect, SPEC, and RiCEPS benchmark suites [4, 9, 25, 22].[11] All told, these Fortran applications contain around 70,000 executable statements. The 23 largest procedures have over 400 G_{CF} nodes; the largest has 812.

5.1 Performance Parameters

The time and space required to build TGSA form are closely related to the size of SSA form, especially if reasonable bounds hold for a few program control-flow characteristics: the fan-out of conditional branches (c_b), the depth of loop nesting (c_l), the number of exits from any single loop (c_x), and the length of any F_{CD} path from the immediate predominator of a merge node to an F_{CF} predecessor of the merge (also referred to here as "the depth of unstructured control," c_u).

[11] RiCEPS is available for anonymous ftp at `cs.rice.edu` in the directory `public/riceps`.

Our notation reflects the belief that these quantities can be treated as small constants.

Our experiments indicate that these metrics usually have small values. The gross outliers from linear time and space requirements also exhibited large values for c_b, c_x, or c_u.

Auxiliary Analyses and Data Structures. Efficient methods exist for computing each of pre- and post-dominators, Tarjan intervals and control dependences [21, 24, 12, 10]. While the asymptotic time bounds range from almost linear to quadratic, for practical purposes, these methods are linear in the number of G_{CF} edges (E_{CF}).

The addition of blocks and edges to G_{CF} is bounded by the number of loop headers and loop-exit edges. The total observed growth in G_{CF} nodes was eight percent; the maximum growth in any procedure was a factor of two.

The addition of ϕ functions for SSA form affects the size of the dataflow graph. While the number of additional ϕ assignments *can* be quadratic in the number of original references, it is linear throughout extensive practical experiments [12]. The number of edges in G_{DU}^{SSA} is linear in the number of references if the arity of merges is constant. In practice, G_{DU}^{SSA} should be far smaller than traditional def-use chains, which are often quadratic in the number of references. Our results agree with the prior literature here.

Auxiliary data structures required to build GSA form from SSA form, such as the vectors of choices, are reused by different calls to *build_loop_predicate()* and *replace_ϕs()*. Their total size is $O(E_{CF})$.

μ and η Functions. There is one μ per loop per variable defined in the loop, which is exactly the number of ϕ functions at loop headers in standard SSA form. There are two cases of η functions. For variables defined in a loop, we need create only one η per level exited, so those η functions are at most $c_x c_l$ times as numerous as the μ functions. Other η functions must be created for predicates, when multiple loop-exit branches merge outside the loop. At the very worst, there are $c_x c_l$ of these η functions per γ dag (*i.e.*, per old ϕ). So the total number of η functions is $O(c_x c_l(\#\mu))$. Our experiments confirm this tight linear relationship, with an average of 1.5 η functions per μ. However, there is wide variation in the number of μ and η functions per loop, presumably because of differences in loop size.

The predicate input of an η refers to the corresponding loop's termination predicate, a γ dag with at most $(c_b - 1)$ γ functions for each block in the loop. (This worst case would require that every block end with a a multi-way loop exit.) The termination predicate is built in time proportional to its size. Since each block can be in at most c_l loops, the total size of all loop-termination predicates is $O(c_b c_l N_{CF})$. Our measurements do not distinguish γ functions in loop predicates from those replacing ϕs.

γ Functions. In structured code, there is only one merge point for each branch (i.e., paths from a branch merge only at its postdominator). In this situation, the γ dag replacing each ϕ has exactly the same inputs and space requirements as the ϕ, except for the addition of the predicate inputs.

In unstructured code, every branch between a ϕ's merge point and its predominator can potentially contribute to the size of the γ dag. A coarse bound for the maximum dag size in this case is $min(N_{CF}, c_b{}^{c_u})$. Therefore, the time and space to replace all the ϕ functions is linear for structured code and quadratic ($O(N_{CF} * N_{DU}^{SSA})$) for the worst case.

5.2 Overall Performance

Figure 7 plots the total size of G_{DU}^{SSA} (nodes and edges, with μ and $\check{\eta}$, but with not γ conversion) against the number of plain variable references (ordinary definitions and uses). While there is some spread, the relationship appears to be linear. The significant outlier is `master` from the RiCEPS program `boast`, with over 30 SSA nodes and edges per plain references. This routine contains one loop with an exceptionally large number of loop exit branches and edges.

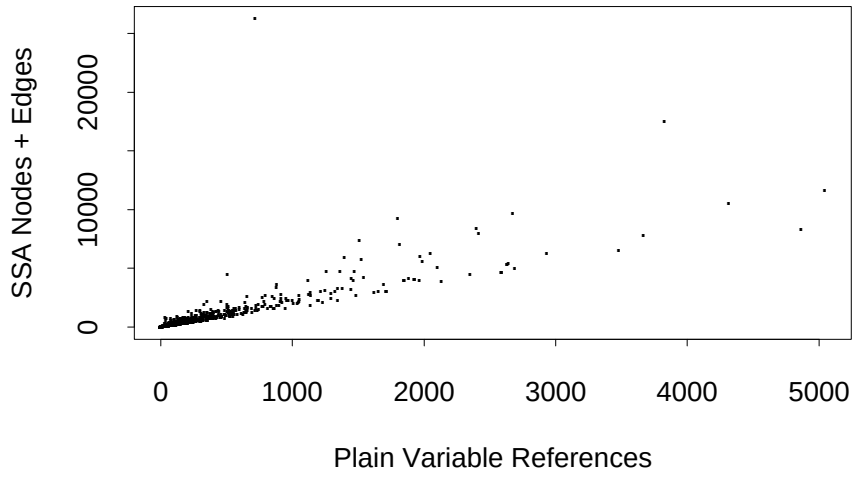

Fig. 7. SSA size vs. Plain References

Figure 8 compares the total size (nodes and edges) of G_{DU}^{GSA} and G_{DU}^{SSA}. The total growth over all procedures was 18 percent. The outlier `mosfet`, from the

SPEC version of `spice`, had a high maximum depth of unstructured control (c_u) — although a high c_u did not always cause such growth.

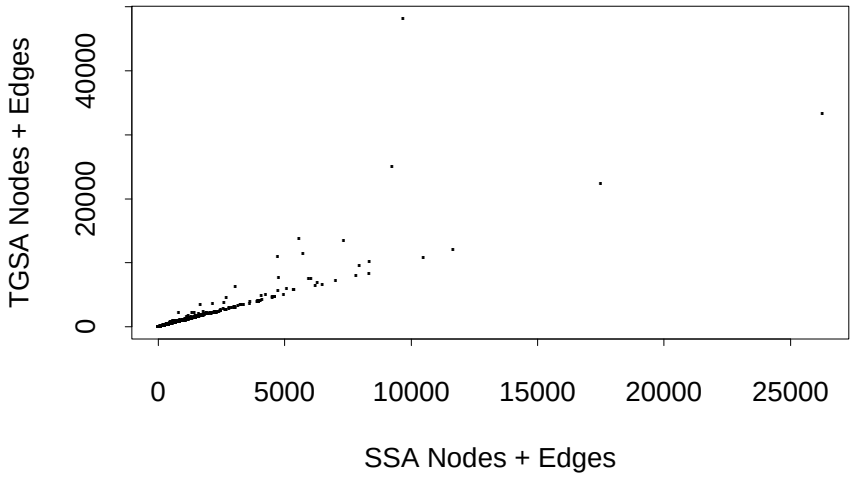

Fig. 8. Comparing size of TGSA and SSA forms

Figure 9 plots the total size of G_{DU}^{GSA} divided by the number of plain variable references. Only a very few small procedures have more than 20 TGSA elements per plain reference. The trend on this graph seems flat, implying a linear relationship between the size of TGSA form and the size of the program.

The total time required to build everything from G_{CF} to G_{DU}^{GSA} is plotted as milliseconds per G_{CF} node and edge in Figure 10. This graph is also quite flat, suggesting a linear relationship between analysis time and program size. The average time required per G_{CF} element on the entire sample was 2.4 ms; the average of the procedure averages was 3.1 ms.[12]

6 Applications and Future Work

We use TGSA form mostly in the analysis of scalar symbolic values, such as array subscripts and branch predicates. The implementation includes a conservative representation of array operations, which is used by other researcher [13].

[12] These measurements were taken inside the ParaScope programming environment, optimized with `gcc` version 2.4.5, running on a Sun MicroSystems Sparc 10 with 64 Mbytes of memory.

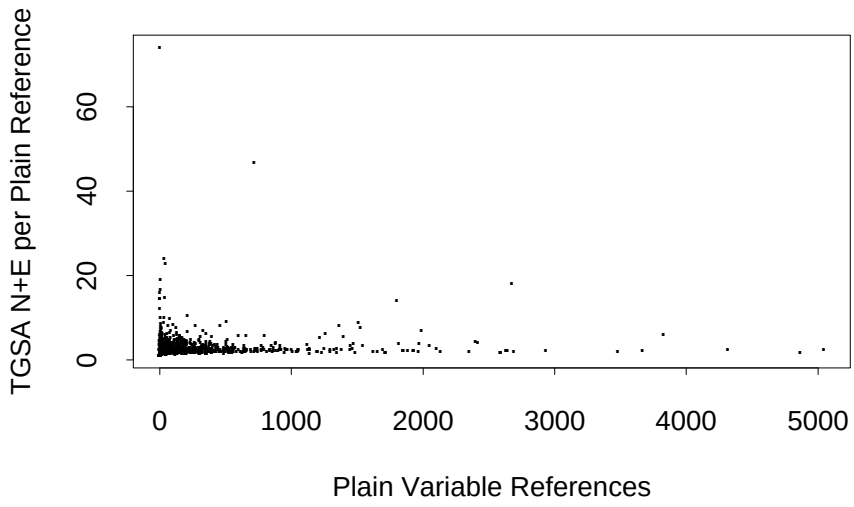

Fig. 9. TGSA Elements per Plain Reference

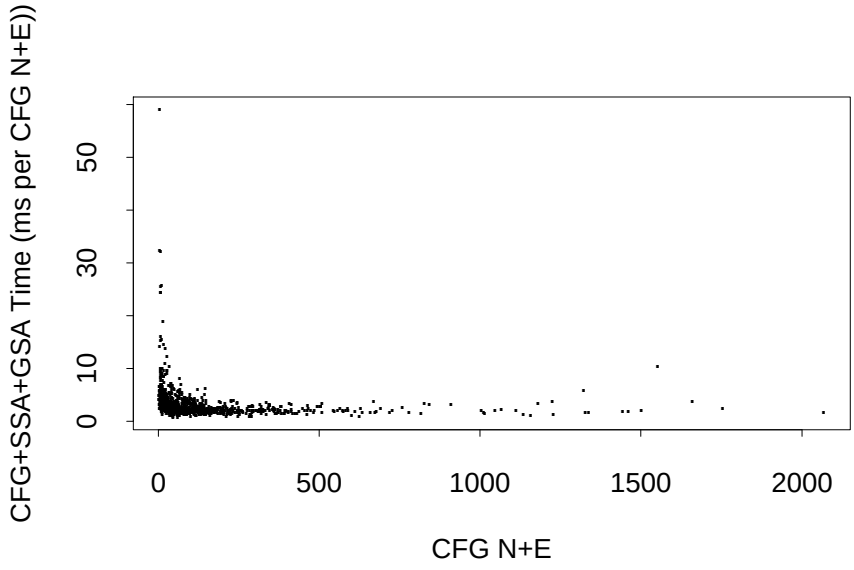

Fig. 10. Total Analysis Time per CFG Element

6.1 Symbolic Analysis.

We have implemented value numbering by hashing on TGSA form in the ParaScope programming environment [20]. The symbolic analyzer builds portions of the hashed value graph when requested by the dependence analyzer. The TGSA-form value graph represents symbolic expressions with a minimal amount of control information, which helps in comparing expressions from different contexts. However, extensions are needed to compute differences, such as dependence distances.

As expression dags are built, they are normalized on the fly by distributing multiplication over addition, extracting constant terms and coefficients, and sorting. Expressions are compared by constructing their difference, simplifying, and checking for a constant result. The power of this method subsumes that of the forward substitution used in earlier vectorizers [2], once further extensions are made for inductive value recognition [26]. While we have yet to complete our experiments with symbolic analysis, we can expect at least a 13 percent increase in dependences disproven, based on prior experience [16].

Related work on symbolic analysis has emphasized symbolic expressions [17] or linear inequalities [19]. The excessive worst-case bounds (cubic to exponential to undecidable) of symbolic analysis make experimental comparisons vitally important. Preliminary experiments with our implementation indicate that it takes time linear in the number of expressions analyzed.

6.2 Semantics

Recent work by John Field gives a formal treatment of graph rewriting on a representation resembling original GSA form [15]. The program representation graph of Horwitz *et al.* also resembles GSA form [27]. Further work is needed to establish a comparably solid formal foundation for TGSA form.

7 Related Work

7.1 High-level SSA form

Alpern *et al.* presented SSA form and proposed extensions to improve handling of conditional merges and loops [3]. For structured programs, their $\phi_{\texttt{if}}$, $\phi_{\texttt{enter}}$, and $\phi_{\texttt{exit}}$ are exactly equivalent to the TGSA-form versions of γ, μ, and η, respectively. They showed how to find the partition of nodes in a *value graph* based on SSA form that gives the maximum number of congruent nodes (equivalent expressions) without rewriting the graph.

TGSA form is identical to high-level SSA form for structured code. The same value partitioning methods apply to TGSA form, extending these results to unstructured programs.

7.2 Original GSA form

Ballance *et al.* introduced GSA form as a component of their program dependence web (PDW) [5]. GSA form seems to have been inspired both by high-level SSA form [3] and PDGs with valve nodes [7].

Original GSA form has an third input to the μ function, the loop-continuation predicate. In addition, it allows γ functions to have a single non-$\top$ value input, as in Figure 11.[13]

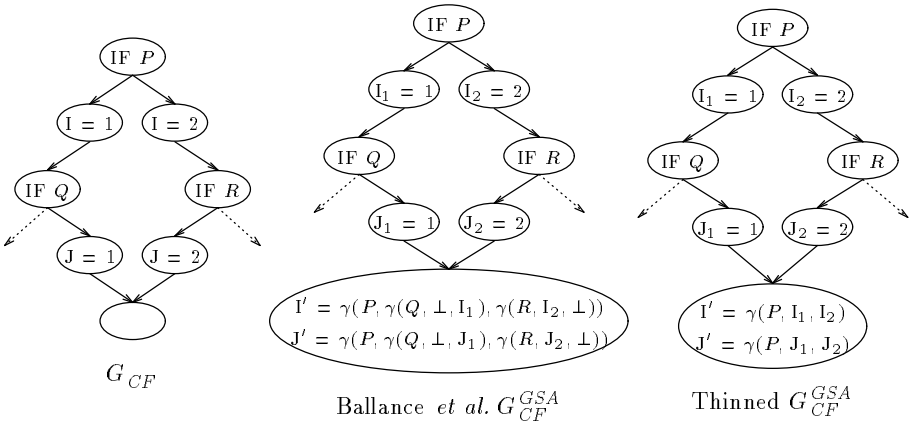

Fig. 11. Different versions of GSA form on G_{CF}

Ballance *et al.* need this additional information to drive the insertion of switches, creating the data-driven form of their PDW. For symbolic analysis, it is sufficient to control values flowing out of a loop, and the predicates for η functions in thinned GSA form satisfy this purpose. Omitting the predicates on μ functions clarifies the independence of iterative sequences, such as induction variables, from the iteration count.

Extra γ functions can obscure equivalences. The original form's γ dags include both the predicates determining which definition reaches the merge *and* those determining whether or not the merge would execute.[14] In Figure 11, if $\mathtt{I}_1$ and $\mathtt{I}_2$ were equal, it would be much easier to notice the simplification for $\mathtt{I}'$ in TGSA form than in the original GSA form.

Original GSA form can be converted to thinned GSA form by globally simplifying the extra γ functions and ignoring the predicate inputs to μ functions.

[13] Actually, they use $-$, which makes sense in their dataflow interpretation, where unexecutable expressions are treated as divergent. In a demand-driven interpretation, unexecutable expressions can be assumed to have any value ($\top$), since that value will never be demanded.

[14] Because conditions for branches preceding the predominator of the merge are implied through the predicate of the γ, the extra functions occur only for unstructured merges.

While both algorithms for building GSA form seem to have the same asymptotic complexity, ours is simpler to implement, especially when working with an unfactored control dependence graph.

The Program Dependence Web researchers have recently developed another revised definition of GSA form, which retains the major differences between with thinned GSA form described above [6].

7.3 Program Dependence Graphs

Ferrante, Ottenstein and Warren introduced the program dependence graph, comprising data dependences and their now-standard formulation of control dependence [14]. Groups at Wisconsin and Rice have developed semantics for PDGs [18, 7, 23]. Selke gives semantics for programs with arrays and arbitrary control flow, and shows how to insert valve nodes efficiently.

Despite the greater maturity of PDGs as a program representation, they are inferior to TGSA form for value numbering. The inputs to each node include the control dependences under which that node executes, which would make value numbering overly conservative for dependence testing. However, ignoring control dependences in value numbering would produce results similar to those for SSA form; we would be unable to compare merged values from different parts of the program.

Perhaps rewriting systems can be defined for PDGs that have the same power as value numbering, but the practical efficiency of such an approach is uncertain.

8 Conclusion

We have introduced thinned gated single-assignment form, which is superior to previous program forms for value numbering. Our original algorithms for building TGSA form take linear time, asymptotically under reasonable assumptions and experimentally on large suite of Fortran applications. Value numbering on our implementation of TGSA form serves as the backbone of cheap yet powerful scalar symbolic analysis in the ParaScope programming environment.

Acknowledgements

I would like to thank Ken Kennedy and my other colleagues on the PFC and ParaScope projects at Rice. Bwolen Yang helped refine the algorithms through his persistent implementation efforts and questions. Robert Cartwright, Rebecca Parsons, and Mary Hall provided substantial help through discussions and comments on earlier drafts of this paper. Karl Ottenstein thoughtfully fielded my questions while attempting to refine the comparison between full and thinned GSA form.

References

1. Alfred V. Aho, Ravi I. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.
2. J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*. PhD thesis, Rice University, April 1983.
3. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 1–11, San Diego, CA, January 1988.
4. David Bailey, Eric Barszcz, Leonardo Dagum, and Horst Simon. NAS parallel benchmark results. Technical Report RNR-92-002, NASA Ames Research Center, February 1993.
5. R. Ballance, A. Maccabe, and K. Ottenstein. The Program Dependence Web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, White Plains, New York, June 1990.
6. Philip L. Campbell, Ksheerabdhi Krishna, and Robert A. Ballance. Refining and defining the Program Dependence Web. Technical Report TR 93-6, Department of Computer Science, University of New Mexico, 1993.
7. Robert S. Cartwright and Matthias Felleisen. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989.
8. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth ACM Symposium on the Principles of Programming Languages*, pages 55–66, January 1991.
9. G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.
10. R. Cytron, J. Ferrante, and V. Sarkar. Compact representations for control dependence. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 337–351, White Plains, New York, June 1990.
11. R. Cytron, J. Ferrante, and V. Sarkar. Experience using control dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.
12. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
13. Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In *Preliminary Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.
14. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
15. John Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, San Francisco, California, June 1992.

16. G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.
17. M. R. Haghighat and C. D. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In *Preliminary Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.
18. S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 146–157, San Diego, CA, January 1988.
19. Francois Irigoin. Interprocedural analyses for programming environments. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*. Elsevier Science Publishers, 1993.
20. K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.
21. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1:121–141, 1979.
22. Allan Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989. Available as Rice COMP TR88-93.
23. Rebecca P. Selke. *A Semantic Framework for Program Dependence*. PhD thesis, Rice University, 1992.
24. R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.
25. J. Uniejewski. SPEC Benchmark Suite: designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.
26. Michael Wolfe. Beyond induction variables. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, California, June 1992.
27. Wuu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical Report 840, Computer Sciences Department, University of Wisconsin-Madison, April 1989.

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