The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages

Robert A. Ballance† Arthur B. Maccabe† Karl J. Ottenstein‡

Abstract

The Program Dependence Web (PDW) is a program representation that can be directly interpreted using control-, data-, or demand-driven models of execution. A PDW combines a single-assignment version of the program with explicit operators that manage the flow of data values. The PDW can be viewed as an augmented *Program Dependence Graph*. Translation to the PDW representation provides the basis for projects to compile Fortran onto dynamic dataflow architectures and simulators. A second application of the PDW is the construction of various compositional semantics for program dependence graphs.

1 Introduction

The **Program Dependence Web** (PDW) is a new intermediate representation for programs that is suitable for control-driven, data-driven, or demand-driven interpretation. Program Dependence Webs are an extension of *Program Dependence Graphs* (PDG) [13] and *static single-assignment* (SSA) form [10]. Thus, the PDW also provides a basis for program optimization. This paper presents an efficient method for translating an imperative program into a Program Dependence Web.

Four application areas may benefit from adopting the PDW representation:

1. The PDW provides the ability to map existing imperative programs to dataflow architectures and demand-driven graph reducers.
2. The PDW includes the information necessary to construct compositional semantics [9, 23] for PDGs.
3. The PDW supports some optimizations better than SSA-form since the PDW incorporates control dependence information not present in SSA-form. For example, the PDW simplifies the methods of Alpern, Wegman, and Zadeck for detecting equivalent computations in the presence of control [2].
4. The PDW can be used to better analyze the performance of existing programs on conventional architectures.

Finally, some results from research concerning functional programming, such as algebraic optimization, partial evaluation, or proofs of program correctness, may be applicable to imperative programs that have been converted into a compositional form.

A PDW contains *all* of the information needed for control-driven, data-driven, or demand-driven interpretation. In practice, one or more **interpretable program graphs** (IPG) will be extracted from the PDW depending on the desired execution model.

Figure 1 illustrates the steps in translating a source program into a PDW. The secondary lines from the GSA-form PDG to the demand-driven and the control-driven IPGs indicate that the required extraction can be performed from the GSA-form PDG directly if so desired. Creation of a dynamic (or static) dataflow IPG requires the full PDW.

Figure 2 contains a code fragment written in an imperative pseudo-language that will be used to illustrate steps in the translation. Comments in Figure 2 provide a unique name for each new definition and for each control predicate.

The first step of the translation into a PDW (ϕ-*placement*) converts the source program into a SSA-form PDG. This conversion is based on a method by

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Los Alamos National Laboratory under contract W-7405-ENG-36, and by Sandia National Laboratory under contract 54-0910.

†Department of Computer Science, University of New Mexico, Albuquerque, NM 87131. ballance@unmvax.cs.unm.edu, maccabe@unmvax.cs.unm.edu

‡Los Alamos National Laboratory, C-3 MS/B265, Los Alamos, NM 87545. kjo@lanl.gov.

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

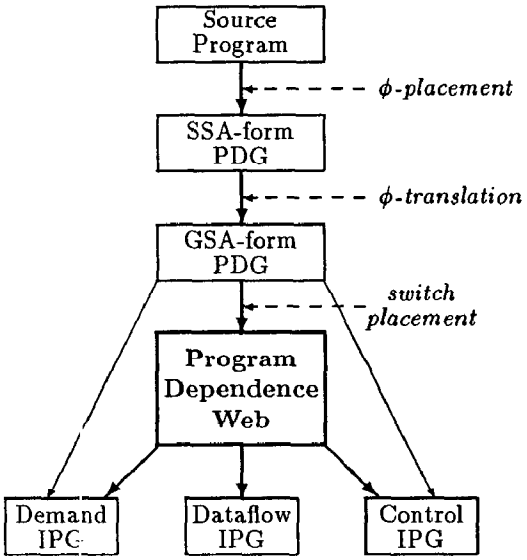

Figure 1: Translation Process

```
read(Y,X)                    /* X1 */
if Y ≥ 0 then                /* Predicate P */
    if Y = 0 then            /* Predicate Q */
        X := 0               /* X2 */
    endif
else
    X := −1                  /* X3 */
endif
... Use X ...
```

Figure 2: Program 1—Nested Conditionals

Cytron, Ferrante, Rosen, Wegman, and Zadeck for computing a minimal static single-assignment (SSA) form for an imperative program [10]. In that method, each definition of a variable is given a unique name. When several definitions of a variable reach a control confluence point, a *φ-function* for that variable is inserted. A ϕ-function creates a new definition for its corresponding variable. After placement of ϕ-functions, each use of a variable is renamed in terms of its single reaching definition. The published method establishes a minimal number of ϕ-functions for each source variable.

The SSA-form Program Dependence Graph for the program of Figure 2 appears in Figure 3. The control dependence subgraph is shown using bold edges. Nodes having a control dependence edge denoted "CD" are control dependent on the region that controls the entire fragment. Each definition of a variable is denoted using a new subscript; the added ϕ-function indicates the confluence of the three definitions X_1, X_2, and X_3 to create the new definition X_4.

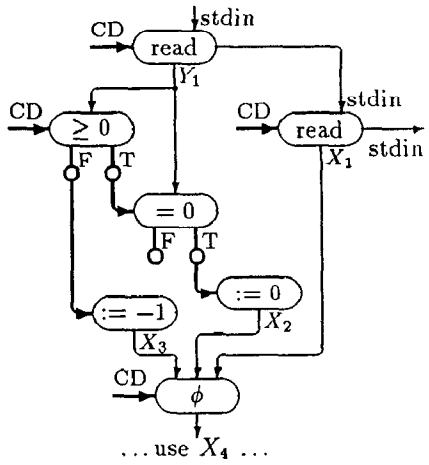

Figure 3: SSA-form PDG for Program 1

Unfortunately, the SSA-form PDG is not directly interpretable; there is no mechanism to discriminate among the various definitions that reach a given ϕ-function. The second step in the translation to a PDW rectifies this shortcoming by converting the SSA-form PDG to a *gated single-assignment* (GSA) form PDG. This step is called *φ-translation.*

In gated single-assignment form, the ϕ-functions are replaced by *gating functions.* Gating functions capture the control conditions that determine which of the definitions reaching a given ϕ-function will provide the value for the function. One such gating function, γ, can be read as a simple *if–then–else.* For example, the ϕ-function in Figure 3, $\phi(X_3, X_2, X_1)$, is replaced with $\gamma(P, \gamma(Q, X_2, X_1), X_3)$. Thus ϕ-translation factors the ϕ-functions and determines the correct control conditions for merging values.

The flow of definitions inside loops is also analyzed during ϕ-translation. Gating functions that manage (i) initial and loop-carried values (μ-functions) and (ii) loop-exiting values (η-functions) are then added to the GSA-form PDG. The resulting graph is immediately suitable for either demand-driven or control driven interpretation.

Figure 4 shows the GSA-form PDG corresponding to the example program. Two γ-nodes factor and replace the original ϕ-function appearing in the SSA-form PDG have been added. No μ- or η-nodes were added since there are no loops.

An ordinary PDG uses a *merge node* to represent the set of definitions reaching a particular variable. Merge nodes, like the ϕ-functions in SSA form, provide information only about what values *may* arrive at some point in a program; not about *how* those values actually get there. Cartwright and Felleisen [9] noted that the separation between value flow and control flow rep-

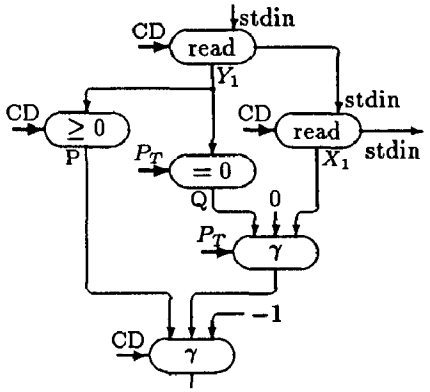

Figure 4: GSA-form PDG for Program 1

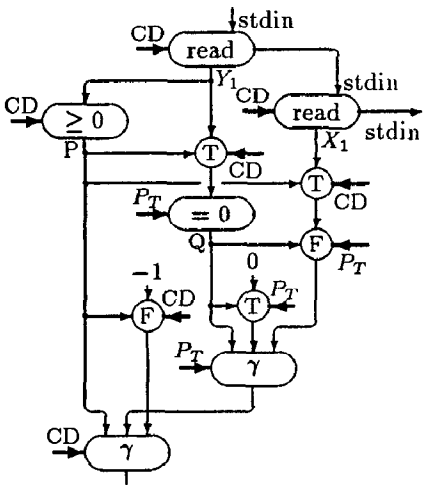

Figure 5: PDW for Program 1

resented a major stumbling block in their efforts to develop a denotational semantics for Program Dependence Graphs. To rectify this difficulty, they introduced *valve* nodes, which are essentially dataflow switches.

The third step of our translation process (*switch placement*) generalizes the placement of valve nodes. In this step, gating functions are added to control the flow of values *into* control regions in which those values are used. Such "flow control" is a function of the same predicates that determine whether the referencing computations will execute.

Switch placement is essential for a dataflow interpretation of the source program. The set of switches and γ-functions controlled by a single control condition delineate an "encapsulator" [25] for a conditional construct. Similarly, the set of μ-functions and η-functions controlled by a single loop-controlling predicate demarcate a loop-encapsulator. The PDW for the code in Figure 2 is shown in Figure 5. For simplicity, switches are indicated by circles labeled with "T" or "F".

Program Dependence Webs and the dataflow IPG form the basis of our efforts to compile Fortran programs for dynamic dataflow architectures. The initial target architectures for this work are Sandia National Laboratories' εpsilon-2 processor [14, 15], the MIT/Motorola Monsoon processor [22], and the MIT GITA interpreter [3, 5, 17].

Other projects have considered the translation of imperative languages such as Fortran to dataflow primitives. Thus far, these efforts have only succeeded for subsets of existing languages. The use of a global dependence graph to facilitate such translations was suggested by Ottenstein in 1978 [18, 19]. He did not specify algorithms for the mapping of control information, however, and array dependences were represented conservatively. In 1980, Allan and Oldehoeft presented a method for generating dataflow code from structured programs [1]. An important aspect of their work was the use of subscript analysis [21] to detect parallelism in array accesses. In 1981, Veen [26, 27] presented a method that appears to be similar to that of Allan and Oldehoeft but does not include subscript analysis. Ongoing work by Beck and Pingali [8] at Cornell and by Suhler [24] at IBM also addresses these issues. Beck and Pingali translate arbitrary code by inserting switches for every value flowing through a control flow branch point and later removing unnecessary switches. Suhler translates a structured subset of Fortran by wiring dependences in an inner-to-outer visitation of control structures.

The remainder of this paper is organized as follows. Section 2 describes those graph elements unique to the PDW and provides a more complete example involving loops. Interpretation of the PDW under the various execution models is discussed in Section 3. Section 4 presents and analyzes an efficient method for translating a SSA-form PDG into a GSA-form PDG. An efficient method for transforming a GSA-form PDG into a PDW using control dependence analysis appears in Section 5. Finally, Section 6 discusses the complications that arise with arrays and aliasing.

2 Unique PDW Elements

The Program Dependence Web contains switches and gating functions, elements not found in the PDG. Switches control the flow of values *into* a region. GSA gating functions control the flow of values *out* of a computation region. This section describes each of these graph elements in more detail. Section 3 describes how switches and GSA gating functions are used under the different execution models.

2.1 Switches

A switch is a binary function $S(p, v)$ with two output ports, denoted by S^T and S^F. The value v is transmitted to one of S^T or S^F according to the truth value of the predicate p [4]. Switches are used exclusively for data-driven interpretation. As noted in the introduction, the flow of values through control regions of a program is bracketed by switches to control entry to a region and γ-functions to control the exit of computed values.

2.2 GSA Gating Functions

The GSA-form PDG and the PDW contain three types of gating functions, denoted γ, μ and η. A γ-function controls forward flow. A μ-function controls the mixing of "loop carried" flow with loop initialization flow. An η-function controls the passage of values out of loop bodies into computations following the loop.

A γ-function $\gamma(P, v^{\mathsf{true}}, v^{\mathsf{false}})$ has three arguments: a predicate P, a true definition v^{true}, and a false definition v^{false}. The semantic definition of a γ-function is similar to the conditional expression: **if** P **then** v^{true} else v^{false}. A γ-function is strict only in the predicate P.

A μ-function $\mu(P, v^{\mathsf{init}}, v^{\mathsf{iter}})$ also has three arguments, a predicate and two variable definitions. The predicate determines whether control will pass into the loop body. Whenever the predicate evaluates to true the body of the loop will be executed. The variable definition v^{init} represents those external definitions that can reach the loop head prior to the first iteration. The second variable definition, v^{iter}, represents those internal definitions that can reach the loop head from within the loop following an iteration. In data-driven terms, the first arrival of a true token on the predicate input to a μ-function causes v^{init} to be returned for the first iteration of the loop. For a *while* loop, this token is produced by an initial evaluation of the loop predicate P. For a *repeat* loop, this token is obtained from the predicates controlling the region in which the loop is nested. The second value, v^{iter}, is returned on all subsequent iterations. When the predicate evaluates to false, the appropriate reaching value is consumed. (Note that v^{iter} may be defined using a γ-function involving v^{init}. Thus, the *value* of the external definition may in fact be used for multiple iterations of the loop. Also, circular dependences among the inputs to a predicate P and the outputs of μ-functions controlled by P entail special handling, as is discussed in the full paper.)

There are two kinds of η-functions, η^T and η^F, each of which takes two arguments, a loop predicate and a definition. A $\eta^T(P, v)$ returns the value v when the predicate P is true and consumes v otherwise. A η^F behaves similarly when P is false.

```
read(Z)                    /* Z1 */
W := 0                     /* W1 */
while Z ≥ 0 do             /* Predicate LP */
    /* CODE FROM PROGRAM 1 */
    read(Y,X)              /* X1 */
    if Y ≥ 0 then          /* Predicate P */
        if Y = 0 then      /* Predicate Q */
            X := 0         /* X2 */
        endif
    else
        X := -1            /* X3 */
    endif
    /* END OF PROGRAM 1 */
    W := X + W             /* W2 */
    Z := Z - 1             /* Z2 */
od
... Use W ...
```

Figure 6: Program 2—Simple Loop

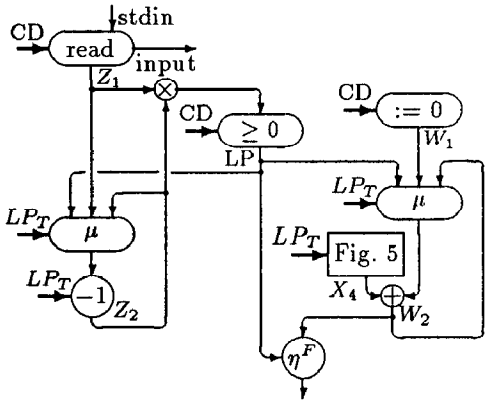

Figure 7: GSA-form PDG for Program 2

Consider the program shown in Figure 6. It consists of variable initializations, a loop, and a subsequent use of the variable W. The code from Program 1 (Figure 2) is nested within the body of the loop.

Figure 7 shows the GSA-form PDG for Program 2. The GSA-form PDG in Figure 7 contains two μ-nodes and a single η-node. The η-node captures the flow of the definition of W from within the loop to a subsequent use. The two μ-nodes capture the flow of definitions of Z and W within the loop. A nondeterministic merge gate ($\otimes$) breaks the cyclic dependence between the predicate $Z \geq 0$ and the μ-node that defines the iteration-dependent value of Z. The sub-PDG from Figure 4 is not repeated. It is identical to that shown in Figure 4, except that its controlling region ("CD") becomes the region that represents the body of the loop.

PDW Element	Execution Model		
	Control	Data	Demand
Control Dep.		ign	ign
Switch	ign		ign
γ-function	ign	ign	
μ-function	ign		
η-function	ign		

Table 1: Interpretation of PDW Elements

Substituting the PDW of Figure 5 into Figure 7 yields the complete PDW for Program 2.

3 Interpretation of a PDW

A PDW embeds interpretable program graphs suitable for control-driven, data-driven, or demand-driven interpretation. Control dependence, switches, and γ-functions are used exclusively by control-driven, data-driven, and demand-driven interpretations, respectively. Both data- and demand-driven interpretations use μ-functions and η-functions, but in different ways. Table 1 lists those web components that are interpreted differently using the various models; all other data operators and data dependence edges in a PDW are required for each model of execution. The symbol ign marks a PDW element that is ignored under the corresponding interpretation.

For control-driven interpretation, the techniques of Ferrante and Mace [12] can be used to linearize the PDW to support a sequential, control-driven execution.

Data-driven interpretation of a PDW is straightforward since a dataflow program graph can be readily extracted from a PDW. First, all control dependence edges and related region nodes are deleted because control dependence information has already been fully incorporated into switches. Second, each γ-function is deleted after making its true and false inputs flow to each of the outputs of the γ-function, and deleting the predicate input edge, and any outgoing edges. The γ-functions are redundant because since switches are inserted symmetrically to γ-functions, assuring that each γ-function will only receive one of its true and false inputs. Third, the set of μ- and η-functions that bracket a loop body are translated into a loop encapsulator, or equivalently into operators to change and reset token color (iteration number) along with switches to control the flow of values into and exiting from the loop body. For the MIT architecture, these operators are *D* and *D-reset* [5].

In demand-driven interpretation, demand originates at program *outputs* and propagates *backwards* through a program graph. As shown in the table, neither switches nor control dependence are required for a demand-driven interpretation. Demand reaches the gating functions controlling the flow of values out of a computation region before it reaches the computations themselves. A demand for the value of a γ-function causes demand to be propagated to the predicate argument first. Based on the value returned, the appropriate of the true or false inputs is demanded. A γ-function that returns $\perp$ does not diverge; it simply returns that value. For simplicity, assume that loops will be interpreted in their entirety rather than slicing through the dependences based on one particular demand. Demand would first reach an η-function controlling the flow of a value out of the loop. By examining the children of the loop predicate, all other η-functions for the loop can be found. Demand then propagates to every corresponding μ-function, demanding first the initial input edge. As long as the loop exit condition is not satisfied, demand continues to be propagated to the μ-functions (and then to the iteration edge). Since demand-driven interpretation requires only the GSA gating functions, it can be applied to the GSA-form PDG as well as the PDW.

4 Creating the GSA-Form PDG

A SSA-form PDG is transformed into a GSA-form PDG by mapping each ϕ-function into a set of GSA gating functions. The initial stages of this ϕ-translation algorithm convert a single ϕ-function into a tree of γ- and μ-nodes. The tree factors the original ϕ-function to show the conditions under which the arguments to the ϕ-function are computed as well as the conditions under which the ϕ-function should be evaluated. The final stage of ϕ-translation adds η-nodes to control values that are computed within loops.

The ϕ-translation algorithm requires the dependence information (primarily control dependence) present in the Program Dependence Graph. The algorithm is restricted to programs having reducible flow graphs[1].

The ϕ-translation algorithm is presented in detail in the following subsections. Section 4.1 provides several necessary definitions. The actual algorithm along with motivating examples is presented in Section 4.2. Sections 4.3 and 4.4 discuss implementation details and analyze the complexity of the algorithm.

4.1 Definitions

For each PDG node n_ϕ representing a ϕ-function to be translated, the ϕ-translation algorithm identifies two subgraphs of the control dependence graph: the n_ϕ *data constraint subgraph*, denoted $\mathcal{DCS}(n_\phi)$, and the n_ϕ *control constraint subgraph*, denoted $\mathcal{CCS}(n_\phi)$. The

[1] This restriction is discussed in our technical report [7].

```
read(X)                          /* X1 */
if X ≥ 6.001 then goto L;/* Predicate P0 */
if Y ≥ 0 then                    /* Predicate Q0 */
    X := 0                       /* X2 */
L:  /* X3 := φ(X1,X2) */
    V := X + 47.0
endif
```

Figure 8: Unstructured Conditionals

ϕ-translation algorithm combines the information embedded in the data constraint and control constraint subgraphs in order to create the GSA expression for a given ϕ-function.

The conditions summarized by the $\mathcal{DCS}(n_\phi)$ control the calculation of the arguments to the ϕ-function.

Definition 1 Let n_ϕ be a node in the PDG G representing the ϕ-function $\phi(v_1, \ldots, v_k)$. For each $i, 1 \leq i \leq k$, let n_{v_i} be the node in G at which v_i is defined. (The node n_{v_i} must be a data predecessor of n_ϕ.) Let $\mathcal{CDG}(G)$ denote the control-dependence subgraph of G. The n_ϕ **data constraint subgraph of** G, $\mathcal{DCS}(n_\phi)$, is the subgraph of G induced by the set of nodes $N_D = \bigcup_{1 \leq i \leq k} \{ nd \in N \mid \exists$ a simple path in $\mathcal{CDG}(G)$ from nd to $n_{v_i} \}$. ∎

The conditions summarized by the $\mathcal{CCS}(n_\phi)$ control the actual execution of the ϕ-function. The gating tree into which the ϕ-function is mapped yields a value only if the conditions in $\mathcal{CCS}(n_\phi)$ hold.

Definition 2 Let n_ϕ be a node in the PDG G representing the ϕ-function. Let $\mathcal{CDG}(G)$ denote the control-dependence subgraph of G. The n_ϕ **control constraint subgraph of** G ($\mathcal{CCS}(n_\phi)$) is the subgraph of G induced by the set of nodes $N_C = \{ n \in N \mid \exists$ a simple path in $\mathcal{CDG}(G)$ from n to $n_\phi \}$. ∎

When the PDG is derived from a structured program, $\mathcal{CCS}(n_\phi)$ is a subgraph of $\mathcal{DCS}(n_\phi)$. For unstructured constructs, $\mathcal{CCS}(n_\phi)$ contains nodes not present in $\mathcal{DCS}(n_\phi)$. During ϕ-translation of unstructured code, the predicate nodes forming the frontier between $\mathcal{CCS}(n_\phi)$ and $\mathcal{DCS}(n_\phi)$ properly annotated.

For example, consider the unstructured program fragment shown in Figure 8. This fragment has the control dependence graph shown in Figure 9. A ϕ-function for X has been placed at the head of the block labeled by L. For the node n_ϕ shown in Figure 9, the nodes defining $\mathcal{DCS}(n_\phi)$ are $\{ P_0, Q_0, r_1, r_2, r_3 \}$ and the nodes defining $\mathcal{CCS}(n_\phi)$ are $\{ P_0, Q_0, r_1, r_2, r_3, r_4, r_5 \}$. The predicates forming the frontier between $\mathcal{CCS}(n_\phi)$ and $\mathcal{DCS}(n_\phi)$ are P_0 and Q_0.

Along with the control and data subgraphs, ϕ-translation uses the notion of a *def(inition) confluence point*[2]. A def confluence point marks a predicate in the control dependence graph that (indirectly) controls two or more definitions reaching a single ϕ-function.

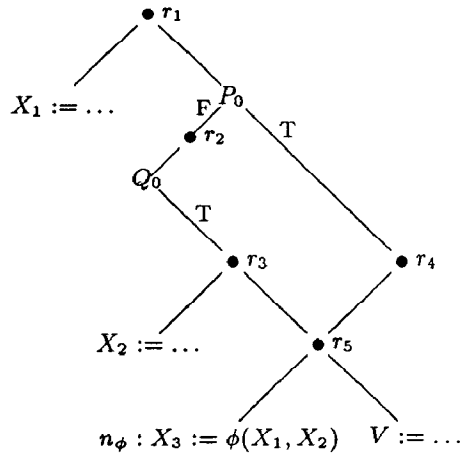

Figure 9: CDG for Figure 8

Definition 3 Let n_ϕ be the node representing a ϕ-function $\phi(v_1, \ldots, v_k)$ in the PDG G, and for each $i, 1 \leq i \leq k$, let n_{v_i} be the node in G at which v_i is defined. A node $n \in \mathcal{DCS}(n_\phi)$ is a **def confluence point** if it is the least common ancestor of $m \geq 2$ distinct nodes $n_1, \ldots, n_m \in \mathcal{DCS}(n_\phi)$, such that for each $j, 1 \leq j \leq m$, either $n_j = n_{v_i}$ for some $i, 1 \leq i \leq k$ or n_j is itself a def confluence-point. The integer m is called the **order** of the def confluence point. ∎

For example, the predicate P in Figure 3 and the region r_1 in Figure 9 are def confluence points of order 2. Predicate nodes that are def confluence points have order 2. Region nodes may have order 2 or 3.

4.2 The ϕ-translation Algorithm

The ϕ-translation converts a single ϕ-function represented by n_ϕ into a GSA gating tree. There are four steps in the algorithm:

1. `Preprocess-DCS`: The nodes in $\mathcal{DCS}(n_\phi)$ are computed and the def-confluence points are marked.
2. `Annotate-Frontier`: If $\mathcal{CCS}(n_\phi)$ contains nodes not present in $\mathcal{DCS}(n_\phi)$, initialize the predicate nodes appearing on the frontier between the two subgraphs with gate subtrees to be used in Step 3. (This case arises only in programs having unstructured control flow.)

[2] A def confluence point should not be confused with a control confluence point in the control flow graph.

3. `Process-DCS`: During a backwards walk over the data constraint subgraph $\mathcal{DCS}(n_\phi)$, map the existing control information into gate subtrees.

4. `Update-PDG`: Update the graph, replacing the ϕ-nodes with the resulting gate subtree and adding η-nodes as necessary.

Steps 2 and 3 combine control information into nested combinations of γ- and μ-nodes called *gate subtrees*. Gate subtrees integrate control and data information encountered during a backwards walk of the data constraint subgraph. As predicates in the subgraph are processed, they are annotated with gate subtrees.

The leaves of gate subtrees are either definitions reaching the ϕ-function or one of two special markers: empty or $\perp$. The placeholder empty is used whenever the correct value has not yet been determined. The special value $\perp$ signifies that control cannot flow to n_ϕ under the corresponding truth value for the γ predicate. All other nodes in the gate subtree represent applications of γ- or μ-functions.

Gate subtrees are built incrementally, from the bottom up. For example, the first time a def confluence point is processed, it is annotated with a gate subtree containing the empty placeholder representing the information about the (currently) unprocessed descendants of that predicate. Subsequent processing fills in the placeholder.

We assume that each region node r in the PDG that is reachable by a back-edge in the control dependence subgraph is marked as a *loop region node*, and that the set $SCR(r)$ is the set of nodes in the strongly-connected region including r. The construction of the control dependence graph ensures that nested loops are represented by distinct strongly-connected regions. By convention, $SCR(nd)$ is empty whenever nd is not a loop region node.

When a loop region node has n incident back-edges, the loop is controlled by n predicates. If $n > 1$, the CDG can be modified so that all n controlling predicates flow into a network of gates whose the single result controls the only back-edge in the CDG incident to the loop region node. The ϕ-translation algorithm assumes that this transformation has occurred. Preprocessing of loop region nodes, including the calculation of strongly-connected regions, is performed once, prior to applying ϕ-translation to any individual ϕ-function.

Figure 10 shows the notation used throughout the presentation of the algorithms.

4.2.1 Preprocessing the Data Constraint Subgraph

Figure 11 shows how the data constraint subgraph is preprocessed. From each unique data predecessor n_{v_i} of n_ϕ, `Preprocess-DCS` performs a depth-first search backwards along incident forward control edges. Each node encountered in the search is a node in $\mathcal{DCS}(n_\phi)$. The walk terminates whenever a node $nd \in \mathcal{DCS}(n_\phi)$ is encountered; the node nd is a def confluence point.

CFpred(n): The set of all predecessors p of n such that the edge $\langle p, n \rangle$ is a forward edge in the control dependence graph.

Cpred(n): The set of predecessors of n in the control dependence graph.

DefConPts(n_ϕ): The set of def confluence points.

Dpred(n): The set of predecessors of n in the data dependence graph.

Dsucc(n): The set of successors of n in the data dependence graph.

GSTree(n): The gate subtree associated with node n.

Op(n) : Returns the outermost gate of GSTree (n); one of def, μ, or γ. *Op*(n) returns def unless the operator for node n is γ or μ and n was inserted during the current ϕ-translation.

SCR(r): The strongly connected region including the loop region node r. If r is not a loop region node, then $SCR(r)$ is empty.

Visit-Ct(n): Calculated to be the order of a def confluence point in `Preprocess-DCS`, it is decremented at each visit during during `Process-DCS`. When it becomes 0, n can be removed from $\mathcal{DCS}(n_\phi)$ and *DefConPts*(n_ϕ) as well.

Figure 10: Notation used in Algorithms

4.2.2 Annotating the Frontier

When $\mathcal{CCS}(n_\phi)$ contains nodes not in $\mathcal{DCS}(n_\phi)$, the control conditions for n_ϕ are not fully determined by the conditions controlling the arguments to n_ϕ. Whenever this occurs, the predicates forming the frontier between the $\mathcal{CCS}(n_\phi)$ and $\mathcal{DCS}(n_\phi)$ must be annotated with gate subtrees that summarize the information unique to $\mathcal{CCS}(n_\phi)$. This information is incorporated into the gate subtrees built during the third phase walk over $\mathcal{DCS}(n_\phi)$. Figure 12 gives the algorithm for this step. The test $Cpred(n_\phi) \not\subseteq \mathcal{DCS}(n_\phi)$ determines whether $\mathcal{CCS}(n_\phi)$ contains any nodes not in $\mathcal{DCS}(n_\phi)$.

```
Preprocess-DCS (n_φ)
[
 DCS(n_φ) ← { }
 foreach p ∈ Dpred(n_φ) do
    Walk-DCS (n_φ, p)
]
Walk-DCS (n_φ, nd)
[
 INCREMENT(Visit-Ct(nd))
 if nd ∉ DCS(n_φ) then
       DCS(n_φ) ← DCS(n_φ) ∪ { nd }
       GSTree (nd) ← undefined
       foreach p ∈ CFpred(nd) do
          Walk-DCS (n_φ, p)
       od
 else
       DefConPts(n_φ) ← DefConPts(n_φ) ∪ { nd }
]
```

Figure 11: Preprocessing $\mathcal{DCS}(n_\phi)$

4.2.3 Propagating the GSA Subtree

`Process-DCS`, shown in Figure 13, pushes gate subtrees from the data predecessors of n_ϕ backwards along forward control edges in $\mathcal{DCS}(n_\phi)$. The algorithm operates by pushing a gate subtree backwards through the subgraph until a def confluence point p is reached. The subtree being pushed is merged with the gate subtrees already associated with p. If p has been visited fewer than m times, where m is the order of p, the traversal stops; otherwise, the resulting subtree associated with p is pushed on recursively.

During the walk, a μ-subtree is created whenever the walk encounters a loop region node on which n_ϕ is control dependent. Otherwise, γ-subtrees are created the first time a predicate node is encountered. Subsequent visits to that node merge the extant gate subtree with the subtree currently being pushed. Subtrees are merged at def confluence points and at those predicates annotated during Step 2.

It is *not* necessary to create γ-subtrees for every newly-encountered predicate. In a structured program, a ϕ-function will be dependent upon some region node in $\mathcal{DCS}(n_\phi)$. If the walk encounters the region node on which the ϕ is control dependent, the most-recently visited predicate must be control dependent on that region as well. Since a γ-function is strict in its predicate argument, any control dependences for the predicate effectively govern the execution of the γ. Thus, if there are predicates nodes in the n_ϕ data constraint subgraph above the ϕ region node, there is no need to create γ's for those additional predicates. If a def confluence point exists higher in the n_ϕ data constraint subgraph, the backwards walk will have to continue to that point, however. Similarly, once a μ-subtree has been constructed, no new γ-nodes should be created during the subsequent backward walk from that loop region node. The algorithm in the next section uses the variable *Wrap-γ?* to control the creation of new γ-nodes.

```
Annotate-Frontier (n_φ)
[
 if Cpred(n_φ) ⊈ DCS(n_φ) then
    Walk-CCS (n_φ, empty)
]
Walk-CCS (nd, GateTree)
[
 foreach p ∈ CFpred(nd) do
    if p is a predicate node then
       if p ∈ DCS(n_φ) then
          Annotate-Predicate (n_φ, p, nd, GateTree)
       else /* p ∉ DCS(n_φ) */
          if nd is the true successor of p then
             Walk-CCS (p, γ(p, GateTree, ⊥))
          else
             Walk-CCS (p, γ(p, ⊥, GateTree))
    elsif p is not ENTRY then
       /* p is a region node */
       Walk-CCS (p, GateTree)
 od
]
Annotate-Predicate (n_φ, p, nd, GateTree)
[
 if p ∈ DefConPts(n_φ) then
    if nd is the true successor of p then
       GSTree (p) ← γ(p, GateTree, empty)
    else
       GSTree (p) ← γ(p, empty, GateTree)
 else /* p is not a def confluence point */
    if (nd ∉ DCS(n_φ)) ∨ (GateTree ≠ empty) then
       if nd is the true successor of p then
          GSTree (p) ← γ(p, GateTree, ⊥)
       else
          GSTree (p) ← γ(p, ⊥, GateTree)
]
```

Figure 12: Annotating the Frontier

For example, consider the following fragment of code:

X_0 := ...
...
if P **then**
  X_t := ...
else
endif
X_ϕ := $\phi(X_0, X_t)$

where the definition of X_0 occurs somewhere above the conditional in the control flow graph, and the renaming and ϕ-function are explicitly shown. Single-assignment

```
Process-DCS (n_φ)
[
 foreach p ∈ Dpred(n_φ) do
    Push-Tree (n_φ, p, p, true)
 od
]
Push-Tree (n_φ, nd, GateTree, Wrap-γ?)
[
 foreach p ∈ CFpred(nd) do
    if p ∈ Cpred(n_φ) then
       Wrap-γ? ← false
    if Cpred(n_φ) ⊆ SCR(p) then
       Visit-Loop (n_φ, p, GateTree, Wrap-γ?)
    elsif p is a region node then
       Visit-Region (n_φ, p, GateTree, Wrap-γ?)
    elsif nd is the true successor of p then
       Visit-T-Predicate (n_φ, p, GateTree, Wrap-γ?)
    else /* nd is the false successor of p */
       Visit-F-Predicate (n_φ, p, GateTree, Wrap-γ?)
 od
]
```

Figure 13: Processing $\mathcal{DCS}(n_\phi)$

form guarantees that only one definition X_0 can pass through the definition-clear path created by the empty *else* clause. The ϕ-function and P have the same control dependences and so depend upon the same region node in the PDG. No matter how deeply the *if* statement is nested, only P is required to choose between X_0 and X_t.

`Visit-Loop` and `Visit-T-Predicate` in Figure 14 are quite straightforward, creating new μ- and γ-nodes respectively. In those routines, the actual creation of new (γ or μ) nodes is implicit. A parameterized γ or μ is a shorthand for a new PDW node representing the indicated gate. The new node is given data predecessors corresponding to the gate parameters and is made control dependent upon $Cpred(n_\phi)$. For conciseness, `Visit-F-Region`, a routine similar to `Visit-T-Region` has been omitted.

`Visit-Region` is complicated by the possibility that the order of a def confluence point is ≥ 2.

`Merge-Tree` is called when propagating a gate subtree *GateTree* to a region node r that has already been marked with another gate subtree. `Merge-Tree` takes two arguments: a gate subtree and a PDG node *nd* with gate subtree GSTree (*nd*). It returns a single, combined gate subtree. The γ and μ subtrees contain some leaves marked empty. Figure 15 shows the possible cases.

```
Visit-Loop (n_φ, p, GateTree, Wrap-γ?)
[
 Let P be the predicate controlling the sole back-edge
 Push-Tree (n_φ, p, μ(P, empty, GateTree), false)
]
Visit-Region (n_φ, p, GateTree, Wrap-γ?)
[
 if p ∈ DefConPts(n_φ) then
    DECREMENT(Visit-Ct(p))
    if GSTree (p) = undefined then
       GSTree (p) ← GateTree
    else
       GSTree (p) ← Merge-Tree (p, GateTree)
       if Visit-Ct(p) = 0 then
          DefConPts(n_φ) ← DefConPts(n_φ) − { p }
          Push-Tree (n_φ, p, GSTree (p), Wrap-γ?)
 else /* p is not a def confluence point */
    Push-Tree (n_φ, p, GateTree, Wrap-γ?)
]
Visit-T-Predicate (n_φ, p, GateTree, Wrap-γ?)
[
 if GSTree (p) = undefined then
    if p ∈ DefConPts(n_φ) then
       DECREMENT(Visit-Ct(p))
       GSTree (p) ← γ(p, GateTree, empty)
    else
       if Wrap-γ? then
          Push-Tree (n_φ, p,
                     γ(p, GateTree, empty), Wrap-γ?)
       else
          Push-Tree (n_φ, p, GateTree, Wrap-γ?)
 else
    DECREMENT(Visit-Ct(p))
    gate ← GSTree (p)
    Let T denote the true subtree of gate.
    T ← Fill-Tree (T, GateTree)
    if Visit-Ct(p) = 0 then
       DefConPts(n_φ) ← DefConPts(n_φ) − { p }
       if DefConPts(n_φ) ≠ { } then
          Push-Tree (n_φ, p, gate, Wrap-γ?)
]
```

Figure 14: Processing $\mathcal{DCS}(n_\phi)$ (continued)

4.2.4 PDG Update

The final phase of the ϕ-translation algorithm has two steps. First, n_ϕ is replaced by the gating tree built by `Process-DCS`. The second step is taken only if the gating tree corresponds to a loop, i.e., the tree is rooted by a μ-node. Here, η functions are inserted along any edge flowing from the μ to a destination outside of the corresponding loop. Figure 16 shows these steps.

```
Merge-Tree (nd, GateTree)
[
 ndtree ← GSTree (nd)
 case (Op(ndtree), Op(GateTree)) of
    (def, γ):
        GSTree (nd) ← Fill-Tree (GateTree, ndtree)
    (γ, def):
        GSTree (nd) ← Fill-Tree (ndtree, GateTree)
    (def, μ):
        Let L denote the left operand of the μ
        L ← Fill-Tree (L, ndtree)
        GSTree (nd) ← GateTree
    (μ, def):
        Let L denote the left operand of the μ
        L ← Fill-Tree (L, GateTree)
        GSTree (nd) ← ndtree
    (γ, μ):
        Let L denote the v^init operand of the μ
        Replace L with Fill-Tree (ndtree, L)
    (μ, γ):
        Let L denote the v^init operand of the μ
        Replace L with Fill-Tree (GateTree, L)
    endcase
 return GSTree (nd)
]
Fill-Tree (tree, val)
[
 Replace all occurrences of empty in tree with val and
 return the result
]
```

Figure 15: Merge-Tree and Fill-Tree

4.3 Implementation Considerations

The complexity of the ϕ-translation algorithm can best be addressed after clarifying several implementation details. Assume that the PDG is built using a doubly-linked graph structure [11, 20]. From any node nd it is a constant time operation to examine the immediate

```
Update-PDG (n_φ, GateTree)
[
 Dsucc(GateTree) ← Dsucc(n_φ)
 delete all edges (p,n_φ)
 delete n_φ
 if Op(GateTree) = μ then
    foreach s ∈ Dsucc(gatetree) do
       if SCR(Cpred(s)) ≠ SCR(Cpred(GateTree)) then
          insert η along edge (s,GateTree)
    od
]
```

Figure 16: Update-PDG

N: the number of nodes in the CDG

E: the number of edges in the CDG

D: the maximum depth of the CDG

V: the number of variables in the program

P: the number of ϕ-functions inserted

Figure 17: Notation used in Algorithm Analysis

control predecessor(s) of nd. Examining the sibling region successors of a predicate is also a constant time operation.

Of the algorithms appearing in the paper by Cytron *et al.* [10], only the ϕ-placement algorithm is required in practice. A subsequent dataflow analysis step that treats each ϕ-function as both a use and a definition can properly compute the input values to ϕ's. Definition points can then be linked to the ϕ-function, eliminating the need for the merge nodes that had been used to summarize sets of reaching definitions for basic blocks in earlier versions of the PDG. No renaming is necessary as each unique definition corresponds to a unique graph node.

Two single-bit flags are required in each node: one to mark whether the node is in $\mathcal{DCS}(n_\phi)$ and one to indicate whether a node is a def confluence point.

Each region and predicate node must have a pointer field to support being marked with another node (γ or μ); it might be more efficient to maintain a table with this information. A parallel implementation becomes complicated since ϕ-subgraphs can overlap; separate mark fields would be needed for each virtual processor. To keep the size of a node small, a parallel implementation could maintain per-processor tables of marks for nodes in the subgraph.

In `Visit-Loop`, if all of the leaves of *GateTree* are the same definition D, it suffices to call Push-Tree $(n_\phi, p, \mu(P, \text{empty}, D), \text{false})$. A mechanism to avoid constructing *GateTree* and just push D up the graph is discussed in the full report [7] along with other efficiency improvements.

4.4 Complexity of ϕ-Translation

Using the notation in Figure 17, a worst-case analysis of the ϕ-translation algorithm suggests a bound of $O(E)$. `Preprocess-DCS` requires $O(E)$ time, while visiting the subgraph again to push definitions and gates through it is also $O(E)$ and all other operations are essentially constant time. Filling in the empty operand positions of a gate tree is at worst proportional to the total number of predicates in the program. Realistically, one can

expect the size of this tree to be bounded by a small constant. The total complexity for translating all ϕ's is therefore $O(PE)$. Since P can be $O(VN)$ and E is $O(N)$, this bound might be stated as $O(VN^2)$. Based on experience with similar algorithms on dependence graphs, greater than quadratic complexity seems very unlikely.

A somewhat tighter bound can be expressed in terms of the depth D of the CDG, which corresponds to the most deeply nested control structure. For most programs, the depth is fairly small and the CDG is quite wide. While of course D is $O(E)$ in the worst case, expressing the complexity in terms of D results in a complexity bound that can be more easily thought of in terms of expected costs. The subgraph walk for each operand of a ϕ is $O(D)$; arguably, the number of ϕ operands is typically bounded by a small constant, so we might take $O(D)$ as the cost of the entire walk. Total translation costs then are $O(PD)$. It can be argued that D is typically bounded by a small constant, too, as conditionals and loops are not nested arbitrarily. The expected complexity is then $O(P)$.

5 From GSA-Form PDG to PDW

The GSA gating functions γ, μ, and η control the flow of values *out* of control regions. But a data-driven interpretation requires that the flow of values *into* regions be controlled as well. In a PDW, this control is exercised by switches that are placed on data edges in the GSA-form PDG.

Switches are required whenever a data value controlled by a region R_1 flows into a different region R_2. Since R_2 can be embedded arbitrarily deeply within R_1, a single data edge may require several switches—one for each predicate on each forward path in the control dependence graph from R_1 to R_2.

The switch insertion algorithm of Figure 18 performs a breadth-first search of the region nodes in the control dependence subgraph. This visitation order bounds the complexity of switch insertion by placing switches in outermost to innermost-predicate order. While switch insertion results in a larger PDW, both with more nodes and longer paths, the inserted switches and edges will not increase the running time of the switch insertion algorithm since no walk will have to traverse new edges.

For each region *reg* encountered in the breadth-first search, `Place-All-Switches` examines all of the γ-nodes whose boolean control is the result of a predicate control dependent upon *reg*. The control information summarized by a γ-function must be propagated into control over the inputs to the node's v^{true} and v^{false} computations. This is performed by the function `Switch-`γ of Figure 20.

```
Place-All-Switches ()
[
 foreach region node R in breadth-first order do
    foreach non-region node N controlled by R do
       if N is a predicate then
          foreach γ gate nd_γ = γ(N, arg1, arg2) do
             /* N is the predicate appearing in nd_γ */
             Switch-γ (N, nd_γ)
          od
       foreach data successor s of N do
          if Cpred(s) ≠ R ∧ s is not a GSA gate then
             sw ← Build-Switches (Cpred(s), R)
             Insert sw on edge ⟨N, s⟩
       od
    od
 od
]
```

Figure 18: Switch Placement Algorithm

Unfortunately, merely inserting switches that correspond to the γ-functions is not sufficient to support a data-driven interpretation. For example, in the code shown in Figure 19, the values used in (and therefore

```
if X > Y then
    if f(X) > g(Z) then
        DivergingComputation()
```

Figure 19: The Need to Switch Upwards Exposed Uses

the result of) the test $f(X) > g(Z)$ must be switched as well. In general, one must switch all of the uses of a definition that reach into a region from outside that region. The second loop in `Place-All-Switches` does the work needed to control values used by non-γ data operators appearing within a region. When an upwards-exposed use of a value is found, the function `Build-Switches` (shown in Figure 21) is invoked to create the switching network. The values used by predicates that control γ-functions are switched in this phase.

5.1 Controlling Inputs to γ-Functions

A γ-function controls the propagation of the results of computations for its v^{true} and v^{false} arguments. This control must be transformed into control over the inputs to those computations. Switch insertion for a single true or false input for a single γ is shown in Figure 20. The algorithm looks backward up the *data dependence* subgraph of the PDG searching for data edges that cross into the region controlled, directly or indirectly, by the region that controls P. The following definition formal-

```
Switch-γ (F, nd_γ)
[
  Obtain a new mark for this traversal
  Place-γ-Switch (truepred(nd_γ), nd_γ, P, true, Cpred(P))
  Obtain a new mark for this traversal
  Place-γ-Switch (falsepred(nd_γ), nd_γ, P, false, Cpred(P))
]

Place-γ-Switch (n_src, n_target, P, boolval, r)
[
 if n_src has not been visited then
    case: n_src is a switch (P, boolval, r):
          /* No action necessary */
    case: n_src is a switch (P, ¬boolval, r) :
          add arc from boolval-output of n_src to n_target
    case: Cpred(n_src) ⊈ R*(r):
          Insert-γ-Switch (n_src, n_target, P, boolval, r)
    case: Cpred(n_src) ⊆ R*(r):
          mark n_src visited
          foreach node n ∈ Dpred(n_src) do
                Place-γ-Switch (n, n_src, P, boolval, r)
          od
    endcase
]

Insert-γ-Switch (n_src, n_target, P, boolval, r)
[
  Sw ← a new switch controlled by P
    with Cpred(Sw) = r
  Insert Sw in edge ⟨ n_src, n_target ⟩
    with boolval-output linked to n_target
]
```

Figure 20: Switch Insertion for Inputs to γ-Nodes

izes the notion of "the region controlled directly or indirectly" by a region node:

Definition 4 Let n be a node in the control dependence subgraph of a PDG G. The set of nodes in the control dependence subgraph that are reachable from n is defined by $\mathcal{R}^*(n) = \{ n_j \mid \exists$ a simple path in $\mathcal{CDG}(G)$ from n to $n_j \}$. ∎

A single depth-first search over the forward edges in the CDG suffices to calculate $\mathcal{R}^*(n)$ for each region in the CDG. Each region node *reg* reached more than once during this depth-first search has multiple control dependence predecessors. The node *reg* can be tagged with the least-common ancestor in the CDG of its forward control dependence predecessors, denoted *region-lca*(*reg*), during the calculation of $\mathcal{R}^*$. The portion of the CDG demarcated by *reg* and *region-lca*(*reg*) represents an unstructured construct in the original program. The value *region-lca*(*reg*) is used in the function `Build-Switches`.

It is possible that two switches controlled by the same predicate P can be placed onto the same original edge

```
Build-Switches (reg, top)
[
 if reg = top then return nil
 if ControllingPred(reg) = { P^boolval } then
    make-AND-net (P, boolval,
                  Build-Switches (CFpred(P), top))
 else
    D ← region-lca(reg)
    make-AND-net (Build-OR-Tree (reg, D), true,
                  Build-Switches (CFpred(D), top))
]

/* Build-OR-Tree is called when reg has */
/* more than 1 controlling predicate */
Build-OR-Tree (reg, top)
[
 if reg = top then return nil
 if CFpred(reg) = { X } then
    return Build-OR-Tree (X, top)
 else
    Let { r_1, ... r_k } = CFpred(reg)
    return make-OR-net (Build-Switches (r_1, top),
                        ...,
                        Build-Switches (r_k, top))
]
```

Figure 21: Switch Placement

in the PDG: one for P^{true} and one for P^{false}. Similarly, several identical switches can be placed onto the (multiple) outputs of a single node. These switches can be merged into a single switch whose outputs fan out to the combined destinations.

5.2 Controlling Upwards Exposed Uses

Controlling the input to non-γ operators is simplified once all of the γ-nodes have been handled. For each data operator node *nd* having an input that does not originate in the same region that controls *nd*, the algorithm builds up a switch network to control the flow of that input. Code for this is shown in Figure 21.

Like `Switch-γ`, the function `Build-Switches` traverses the CDG backwards along forward control dependence edges. The traversal ends whenever the region controlling the source of the value to be switched is encountered. `Build-Switches` collects linear chains of predicates into linear sequences of switches representing nested conditional execution. Whenever a region has multiple controlling predicates, however, the original program was (locally) unstructured. In this case, a disjunction of control paths must be created as shown in `Build-OR-Tree`.

In Figure 21, the function *ControllingPred*(R) returns the set of predicates that immediately control the region

node X. Each predicate is superscripted by the sense of the predicate (true, false) that controls X. In a structured program, $||ControllingPred(X)|| \leq 1$.

`Build-OR-Tree` handles unstructured control flow. Since the γ-nodes handle the case of multiple definitions reaching a single control confluence point, any definition that flows into an unstructured component must come from outside that component. In particular, it must come from a region that (directly or indirectly) controls the entire unstructured component. Thus, `Build-OR-Tree` creates a switching network representing the entire unstructured component. It does so by constructing a disjunction over all forward paths in the CDG up to, but not including, the predicate that is the LCA in the forward CDG of the regions within the unstructured components.

The functions `make-AND-net` and `make-OR-net` construct single-input, single-output networks composed of switches. Networks of switches, rather than AND and OR operators, are required to correctly implement a data-driven semantics. (Assume that `make-AND-net` and `make-OR-net` handle a nil argument appropriately.) Pseudo-code for these functions appears in the full report.

The execution time of `Place-Switches` and the size of the generated switch network will be decreased by caching copies of the switch networks built by `Build-Switches`.

5.3 Analysis

The breadth-first visit of CDG predicate nodes in the switch placement algorithm causes switches to be inserted in an outermost-to-inner nesting order. While `Place-All-Switches` can increase the size of the graph, none of the newly added nodes and edges will be visited by subsequent calls to `Place-All-Switches` as the search frontier is always at most the first new node encountered.

Using the terminology from the analysis of Algorithm ϕ-Translation in Section 4.4, a worst case complexity bound can be constructed with $O(N)$ for the breadth-first search to provide the ordering visitation for the $O(PD)$ γ-functions. Each such visit causes in invocation of `Place-All-Switches` requiring time $O(D)$. The cost of switching upwards exposed uses is $O(ND)$. This gives a bound of $O(N + PD^2 + ND)$. Since the number of γ-functions can be $O(VN)$, and D can be $O(N)$, this worst-case bound can be $O(VN^2)$. Since D is typically bounded by a small constant, switch placement is expected to exhibit linear complexity of $O(N + P)$.

6 Arrays and Aliasing

So far, our examples have involved only scalar variables and constants. This section briefly presents the complications that arise due to arrays and aliasing. The full paper discusses these issues in more detail.

Arrays can be naïvely handled in the same way as scalar variables. The cost of naïveté is that demand- and data-driven interpreters must copy the entire array after every update in order to maintain single-assignment semantics. This problem is well-known in functional programming. The advantage in translating an imperative program into a PDW is that a correct sequential interpretation using a single copy of each array is known to exist. Another extreme is to sequentialize the updates to each array. Between these extremes, subscript analysis [21] can be employed to determine some cases where accesses and updates are independent and therefore can remain parallel, or when I-structures [6] can be substituted for imperative arrays. Cost analyses can determine when small regions of array accesses should be kept sequential and others should involve copying to increase parallelism.

In Fortran, aliasing only becomes interesting at an interprocedural level. Consider the aliasing of scalars through call-by-reference parameters and COMMON blocks. Interprocedural dataflow analysis can provide sets of *may* and *must* aliases on a per-call-site and summary basis. In the case of *may* aliases, sequentializing switches can be inserted that are predicated on a run-time test of the associated variables' addresses. A form of node-splitting can be performed to create several different versions of a procedure based on the alias relationships. For example, if there are some call sites that do not induce aliases, a version of the procedure body can be generated that does not contain the overhead of run-time tests and switches. The additional complication engendered by arrays is that aliases may overlap: array to array or array to scalar. Subscript analysis and sequentialization are required for a correct program representation.

7 Summary and Future Work

A new program representation, the Program Dependence Web, has been introduced along with a means for its interpretation under control, data, and demand-driven evaluation models. Efficient algorithms have been presented for translating an imperative program into a PDW.

The next step is to implement the ideas presented in this paper. We are modifying an experimental front-end [11, 20] that maps a Fortran-77 program into an operator-level Program Dependence Graph. The PDG will be transformed into a corresponding MIT dataflow

program graph to be passed to the back-end of the *MIT Id compiler* [16, 25]. The Id back-end produces machine graphs for the Monsoon architecture; a version producing machine graphs for the epsilon-2 architecture [14, 15] is being developed at Sandia National Laboratory. The Fortran front-end performs interprocedural dataflow analysis to find procedural side effects. Global references are converted into explicit parameters of call nodes in the PDG and hence in the dataflow program graph.

The potential benefits of this project include not only the porting of existing Fortran programs to dataflow architectures, but also the ability to better analyze the performance of those programs on conventional architectures. The speedup curve for an idealized execution on the MIT simulator under a varying number of processors provides a realistic asymptote for the potential parallelism in the executing program. This curve will be much more useful in gauging the effectiveness of hand or automatic parallelization on conventional architectures than is the overly optimistic linear speedup asymptote. Ultimately, a compiler might be developed that uses performance information gleaned from dataflow simulations in order to support better automatic partitioning for conventional MIMD architectures. Translation of imperative programs into a compositional form may also permit the application to imperative programs of functional language results concerning algebraic optimization and program proving.

8 Acknowledgements

We appreciate motivational conversations with Arvind, Rishiyur Nikhil and Ken Traub, particularly Nikhil and Ken's clarification of the construction of the MIT dynamic dataflow program graphs. Kattamuri Ekanadham suggested the idea of combining our separate data-driven and demand-driven graph forms into a single representation. Mark Wegman noted the improved optimization capabilities offered by the PDW and Paul Suhler spied several editorial errors in an earlier draft.

References

[1] ALLAN, S. J., AND OLDEHOEFT, A. E. A flow analysis procedure for the translation of high level languages to a data flow language. *IEEE Trans. on Computers TC-29*, 9 (Sept. 1980), 826–831.

[2] ALPERN, B., WEGMAN, M. N., AND ZADECK, F. K. Detecting equality of variables in programs. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages* (San Diego, California, January 13–15, 1988), ACM, pp. 1–11.

[3] ARVIND, DERTOUZOS, M. L., NIKHIL, R. S., AND PAPADOPOULOS, G. M. Project Dataflow: A parallel computing system based on the Monsoon architecture and the Id programming language. Computation Structures Group Memo 285, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, March 25, 1988.

[4] ARVIND, AND GOSTELOW, K. P. The U-interpreter. *IEEE Computer 15*, 2 (Feb. 1982), 42–50.

[5] ARVIND, AND NIKHIL, R. S. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. on Computers TC-39*, 3 (March 1990), 300–318.

[6] ARVIND, NIKHIL, R. S., AND PINGALI, K. K. I-Structures: Data structures for parallel computing. *ACM Trans. on Programming Languages and Systems 11*, 4 (Oct. 1989), 598–632.

[7] BALLANCE, R. A., MACCABE, A. B., AND OTTENSTEIN, K. J. The program dependence web: A representation supporting control, data, and demand-driven interpretation of imperative languages. Technical Report LA-UR-89-3654, Los Alamos National Laboratory, 1990.

[8] BECK, M., AND PINGALI, K. From control flow to dataflow. Tech. Rep. 89-1050, Cornell University, Dept. of Computer Science, Oct. 1989.

[9] CARTWRIGHT, R., AND FELLEISEN, M. The semantics of program dependence. In *Proc. of the SIGPLAN '89 Conference on Programming Language Design and Implementation* (Portland, Oregon, June 21–23, 1989), ACM, pp. 13–27. Appeared as Sigplan Notices, 24(7), July 1989.

[10] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. An efficient method of computing static single assignment form. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages* (Los Angeles, January 17-19, 1989), ACM, pp. 25–35.

[11] ELLCEY, S. J. The program dependence graph: Interprocedural information representation and general space requirements. M.S. thesis, Michigan Technological University, Jan. 1985.

[12] FERRANTE, J., AND MACE, M. On linearizing parallel code. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, Louisiana, January 14–16, 1985), ACM, pp. 179–190.

[13] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems 9*, 3 (July 1987), 319–349.

[14] GRAFE, V. G., DAVIDSON, G. S., HOCH, J. E., AND HOLMES, V. P. The epsilon dataflow processor. In *Proc. 16th International Symposium on Computer Architecture* (1989).

[15] GRAFE, V. G., HOCH, J. E., AND DAVIDSON, G. S. Eps'88: Combining the best features of von Neumann and dataflow computing. Sandia Report SAND88-3128, Sandia National Laboratories, Albuquerque, New Mexico, 87185, Jan. 1989.

[16] NIKHIL, R. S. Id (version 88.0) reference manual. Computation Structures Group Memo 284, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, March 25, 1988.

[17] NIKHIL, R. S., FENSTERMACHER, P. R., AND HICKS, J. E. Id World reference manual (for LISP machines). Computation structures group memo, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, August 23, 1988.

[18] OTTENSTEIN, K. J. *Data-Flow Graphs as an Intermediate Program Form.* PhD thesis, Purdue University, Jan. 1978. Available through University Microfilms, Ann Arbor, MI.

[19] OTTENSTEIN, K. J. Summary of presentation. Tech. Rep. TM-136, Massachusetts Institute of Technology Laboratory for Computer Science, June 1979. (Report on the Second Workshop on Data Flow Computer and Program Organization, M.I.T., July 1978).

[20] OTTENSTEIN, K. J., AND ELLCEY, S. J. Experience compiling Fortran to program dependence graphs. Tech. rep., Los Alamos National Laboratory, 1989. In progress.

[21] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Communications of the ACM 29*, 12 (Dec. 1986), 1184–1201.

[22] PAPADOPOULOS, G. M. Implementation of a general-purpose dataflow multiprocessor. Tech. rep., Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, Aug. 1988.

[23] SELKE, R. P. A rewriting semantics for program dependence graphs. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages* (Los Angeles, January 17-19, 1989), ACM, pp. 12–24.

[24] SUHLER, P. Personal communications.

[25] TRAUB, K. R. A compiler for the MIT tagged-token dataflow architecture. Tech. rep., Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, Aug. 1986.

[26] VEEN, A. H. Reconciling data flow machines and conventional languages. In *Proc. Conf. on Analysing Problem Classes and Programming for Parallel Computing* (Nurnberg, 1981), W. Handler, Ed., no. 111 in LNCS, Springer-Verlag, pp. 127–140.

[27] VEEN, A. H. *The Misconstrued Semicolon: Reconciling Imperative Languages and Dataflow Machines*, vol. CWI Tract 26. Centrum voor Wiskunde en Informatica, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 1986.