Refining and Defining the Program Dependence Web

Philip L. Campbell
Ksheerabdhi Krishna
Robert A. Ballance

Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131
Technical Report No. 93-6

February, 1993
Revised, March 1993

This work was supported in part by NSF under grant CCR-9110954 and by Sandia National Laboratories supported by the US Department of Energy under contract DE-AC04-76DP00789.

Abstract

The Program Dependence Web is a program intermediate representation which can be interpreted under control-driven, data-driven or demand-driven disciplines. An operational definition is given for the Program Dependence Web. This includes definitions for the nodes and arcs and a description of how webs are interpreted. The general structure for conditionals and loops is shown, accompanied by examples. The definition is a refinement of the original one in that a new node, the "β node," is introduced and two nodes, the μ node and the η^T node, are eliminated.

1.0 Introduction

To obtain peak performance it must be easy to match computations with machines. Today's world of MIMD supercomputing makes this difficult because it is divided into separate camps. There is the control-driven von Neumann massively parallel camp. There is the data-driven dataflow machine camp. And there is the demand-driven graph reduction machine camp. Each camp confines users to one language style: imperative, dataflow or functional. Moving computations between camps is difficult. We believe this is a barrier to performance and to productivity. The Program Dependence Web (PDW) [1] helps to eliminate this barrier.

The PDW is a program intermediate representation which can be interpreted under control-driven, data-driven or demand-driven disciplines. The PDW presents the opportunity of interpreting different parts of the same program in different ways, depending on the part being executed, the input being used, or both. The PDW thus subsumes various hybrids as well. The PDW is not confined to a single language style, just as it is not confined to machines of a particular execution style, because it is an intermediate represen-

tation. The PDW opens the development of architectures which can match computations with machines and this will lead to increased performance [6].

The PDW is an extension of the Program Dependence Graph (PDG) [5], a representation which unifies control- and data-dependences in a program within one framework. The PDG was conceived specifically for program optimization, accommodating standard optimizations as well as including control- and data-dependence information. The PDG, however, was not intended to be interpretable. Although it is possible to perform a control-driven interpretation of the PDG, for a data- and demand-driven interpretation the PDG must be augmented with new elements possessing appropriate semantics. The PDW extends the PDG by adding graph elements and by providing each element with a semantics appropriate to one or more models of interpretation.

The PDW has several advantages over other graph based representations:

1. It is interpretable, besides being a vehicle for performing optimizations.
2. It enables "cross-compilation" of programs written in a language suited to one execution paradigm onto a different one (e.g., mapping imperative programs on to data-flow architectures [1]).
3. It supports data- and demand-driven semantics within one representation, facilitating hybrid evaluation schemes like operator nets where some parts of the program graph may be evaluated eagerly (data-driven) and other parts lazily (demand-driven) depending on the "need" of the values [6] [9].
4. It is ideal for employing techniques like partial evaluation for code specialization. The PDW provides the kind of information (control-, data-flow) and flexibility (interpretable) required by a partial evaluator for code specialization.
5. It provides insight into the algorithm-to-architecture mapping problem for parallel architectures using different execution disciplines [13].

In their seminal paper Ballance et al. [1] present an informal definition of the PDW and algorithms for producing a PDW for a given program. This paper refines the definition of the PDW by replacing the μ node with a new node, the "β node," and eliminates the η^T node. The new β node eliminates the need for special loop priming constructs. This paper provides for the first time an operational definition of the PDW.

The report is organized as follows. Section 2.0 provides background to PDWs by pointing out how PDWs are related to PDGs and reviewing the original definitions. Section 3.0 refines the PDW by solving several problems in the original presentation. Section 4.0 presents an operational definition for the PDW. Section 5.0 provides for the first time a description of how a PDW is interpreted under control-, data- and demand-driven semantics. Section 6.0 concludes by providing the generic structure and concrete examples of PDWs for conditionals and loops. The appendix contains operational semantics for each of the PDW nodes.

2.0 Background

The Program Dependence Graph abstracts the control- and the data-dependences of a program. It encodes both control and data dependence information using directed edges

connecting graph vertices. The vertices in a PDG may represent several concepts: data operators, branch points in control flow, or regions of common control dependence. Edges represent any of the several varieties of data dependence or control dependence. Auxiliary edges may be present to indicate special kinds of data-flow, such as different kinds of procedure parameter passing. For practical purposes, every PDG is rooted at a distinct vertex named “Entry”.

The control dependence subgraph is typically built in four stages:

1. **Control flow graph:** Program is converted into an Abstract Syntax Tree/Control Flow Graph.
2. **Dominator tree construction:** Dominator information is extracted from the control flow graph.
3. **Control dependence calculation:** Dominator tree and the control flow graph are used to determine control dependences.
4. **Region node addition:** Nodes that have a common set of control dependence conditions, as defined by predicates, are factored into a region.

For structured programs the control dependence subgraph can be built in one step [2]. Note that the control dependence subgraph has, in addition to statement nodes, region nodes that summarize the conditions required for execution.

The data dependence subgraph is built in two stages:

1. **Data flow analysis:** Data dependence information on a per node level is obtained.
2. **Information propagation:** The data dependence subgraph is constructed by linking information obtained in the previous step.

A distinguished node, “undefined,” is a data-dependence predecessor of all nodes which might reference an uninitialized variable. If there is more than one reaching definition for an upwards exposed use, a merge node is created that is made dependent upon each reaching definition [5]. I/O is represented by treating file names similar to array names. A complete example of a PDG can be found elsewhere [8].

2.1 The Program Dependence Web

The original PDW [1] is derived from the PDG by adding graph elements (nodes) that have the proper semantics for interpretation. The additional graph elements can be classified into the following groups:

- **Gated Single Assignment (GSA) gating functions:** γ, μ, η^{T}, η^{F}

  The γ-function, $\gamma(P, v^{true}, v^{false})$ is similar to the conditonal expression: if P then v^{true} else v^{false}. The γ-function controls forward flow.

  The μ-function, $\mu(P, v^{init}, v^{iter})$ controls the mixing of “loop initialization” and “loop carried” flow. The μ-function is strict in predicate P. The predicate P determines whether control will pass into the loop body. The variable definition v^{init} represents those external definitions that can reach the loop head prior to the first iteration. The variable definition v^{iter} represents those internal definitions that can reach the loop head from within the loop following an iteration.

The η^T-function, $\eta^T(P, v)$ returns the value *v* when the predicate *P* is *true* and consumes *v* otherwise.

The η^F-function, $\eta^F(P, v)$ returns the value *v* when the predicate *P* is *false* and consumes *v* otherwise.

- **Loop priming Merge nodes**

  The Merge node is also known as the non-deterministic merge. It is similar to the "merge" node present in the PDG which is a stub for reaching definitions, a point where upwards exposed uses are linked with downwards exposed definitions [8]. The loop priming Merge node provides information only about *what* values may arrive at the entry of the loop, not about *how* those values get there.

- **Dataflow Switches: Switch^T, Switch^F**

  A Switch, Switch$^T(P, v)$ transmits the value *v*, if predicate *P* is *true*; when the predicate is *false* the behaviour of the Switch is unspecified.

  A Switch, Switch$^F(P, v)$ transmits the value *v*, if predicate *P* is *false*; when the predicate is *true* the behaviour of the Switch is unspecified.

Switches control the flow of values *into* a region whereas GSA gating functions control the flow of values *out of* a region. Section 3.0 discusses how these nodes are used in an interpretation.

Ballance et al. have provided algorithms to derive a PDW from a PDG [1]. They use as starting point an SSA-PDG. A program is defined to be in Static Single Assignment(SSA) form if each variable is a target of exactly one assignment statement in the program text [4]. Deriving an SSA form of a program requires placement of a special form of assignment called the ϕ-*function*. A ϕ-function provides information about *what* values may arrive at some point in a program. The ϕ-functions inserted in the SSA form replace the merge nodes in addition to reducing the size of the PDG since there may be unnecessary merge nodes [8]. Figure 1 shows how one ϕ-function obviates an extra merge node. Starting with an SSA-PDG the following steps are adequate for constructing the PDW:

1. **ϕ translation:** Replace the ϕ nodes with appropriate γ nodes. Replace loop predicates with appropriate μ–nodes and insert η^T,η^F nodes [Loop translation].
2. **Switch placement:** Place appropriate switches (S^T, S^F) to regulate flow of data.

FIGURE 1. (a) A program fragment (b) in SSA-form with ϕ-functions, (c) with merge nodes.

```
x = ...
if (P) {
  x = ...
  ....
}

if (Q) {
  ....
  if (R) {
    ... = x
  }
  ....
}

... = x
```

(a)

```
x1 = ...
if (P) {
  x2 = ...
  ....
}
x3 = ϕ(x1,x2)
if (Q) {
  ....
  if (R) {
    ... = x3
  }
  ....
}

... = x3
```

(b)

```
x = ...    /* Si */
if (P) {
  x = ...  /* Sj */
  ....
}

if (Q) {
  ....
  if (R) {
  merge(Si,Sj)
    ... = x
  }
  ....
}
  merge(Si,Sj)
... = x
```

(c)

However, it is not necessary to derive the PDW from an SSA-PDG . For structured programs it is possible to derive the PDW from a PDG with information obtained from reaching definitions. In structured programs, ϕ translation depends only on the inputs to the ϕ-function. Since merge nodes in a structured program mirror ϕ-functions, it is possible to extend ϕ translation to a PDG with merge node information. The algorithms for exact placement of PDW graph elements is beyond the scope of this paper.

The only remaining complexity in the construction of the PDW stems from the fact that computing the PDG involves several stages. Ballance and Maccabe have shown that for structured programs the task of constructing a PDG can be considerably simplified [2].

2.2 Interpretation

“Control-”, “data-” and “demand-” driven denote three different modes of program execution.

Under *control-driven execution* the next instruction to execute is specified by the address in the program counter.

Under *data-driven execution* there is no program counter. Any node (which denotes an operation) that has the required inputs to execute is a candidate for execution. The next instruction(s) to execute is chosen from these candidates. The choice is implementation dependent.

Under *demand-driven execution* there is likewise no program counter. The set of “demanded” instructions in this scheme forms a directed graph: nodes with zero in-degree represent the original demand from the world outside the program; the remaining nodes in the graph are instructions which have generated demands and still have at least one of those demands unfulfilled; the leaves (zero out-degree) are instructions which have either not yet issued any demands or have had all their demands fulfilled but have

not yet completed execution. How the next instruction(s) is chosen from the leaves of this graph is implementation dependent.

Compilers and interpreters may choose to interleave these modes. Data flow machines may require demand semantics (e.g., Lucid compiler [9], DTN data flow machine [6]) because in some instances a demand-driven evaluation outperforms data-driven evaluation.

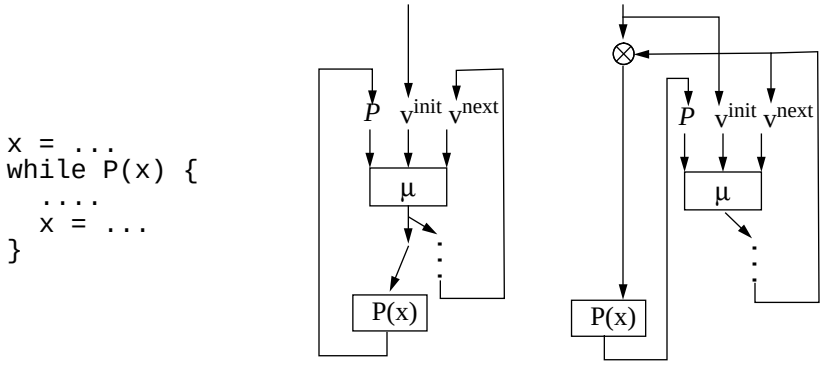
3.0 Refining the Original PDW

The original definition of the PDW [1] had two problems with loops and it had a superfluous node. These problems and their solution are briefly reviewed below.

3.1 Problems with loops

Figure 2(a) shows a program fragment containing a while loop. The loop controlling predicate depends upon a value x, within the loop. Figure 2(b) shows a PDW with the μ node in place. Recall that since the μ node is strict in the predicate *P*, the μ node cannot fire until the predicate produces a value but the predicate cannot fire until it receives a value from the μ node.

FIGURE 2. Pre-test loop and corresponding PDW fragments



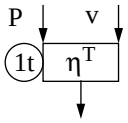
(a) Program fragment (b) Partial PDW without the Merge (c) with the Merge ⊗

The creators of the PDW broke this cyclic dependency by adding a non-deterministic Merge as shown in Figure 2(c). They however provided no placement algorithms for the Merge. Their suggested placement of the Merge as shown in Figure 2(c) works correctly only under control-driven interpretation. This is because a total of exactly one token will be waiting on the input arcs to the Merge node whenever it executes under control-driven interpretation. Unfortunately under data- and demand-driven interpretation the Merge fails since one or more tokens may be waiting on each of the input arcs to the Merge. The Merge is unable to hold back initial loop values while iteration values are still circulating, and thus it fails.

A second problem is that the μ node does not generate any output when the predicate evaluates to *false*. This forces the η^F nodes (not shown in Figure 2) to tap the v^{init} line of their associated μ nodes. Unfortunately this works only for zero trip executions of pre-test loops and single trip executions of post-test loops. In all other instances of loop execution, the η^F nodes are unable to propagate appropriate values!

This paper presents a revised definition of the μ node that handles both of the problems mentioned. The semantics of the μ node are sufficiently different that the μ node is replaced with a new node: the β node. The operational definition and semantics of β appear in Section 4.1.9.

3.2 Eliminating the η^T node



The η^T node is intended to allow values to exit a loop. For each *false* token it receives on its *P* arc, it consumes the next data token on its *v* arc; for each *true* token it receives on its *P* arc, it lets the next data token on its *v* arc through to its output. Unfortunately, loops continue to cycle while the predicate remains *true*, terminating when the predicate turns *false*, exactly the opposite of what the η^T node expects. Thus there is no place for the η^T node and it can be eliminated. The superscript of the η^F node is retained not so much to distinguish it from the η^T node but to preserve the clarity of its own functionality.

Implicit in our assumptions is that loops follow one style for termination. Pascal-style `REPEAT-body-UNTIL(P)` statements are semantically equivalent to `DO-body-WHILE(not P)` statements. That is, the loop continues to iterate while the predicate "`not P`" remains *true*, terminating when it becomes *false*. Since the Switch η^F and β nodes all presume this arrangement, directly representing Pascal-style `REPEAT-body-UNTIL(P)` loops would require a second set of these nodes. One set would be used in loops which terminate when the predicate turns *false*; and another used in loops which terminate when the predicate turns *true*. Since this complicates matters and is, as noted above, not necessary, it is eliminated.

4.0 Defining the PDW

The Program Dependence Web is a directed graph G = (V,A) where,

V = {Entry, Exit, operators, predicate, read/write, switch^T/switch^F, η^F, γ, β}
A = {control dependence^{true}, control dependence^{false}, data, other data-dependences}.

4.1 Nodes

Table 1 summarizes the use of PDW elements and their behavior. Unqualified names in the Table denote primary definitions. For example, Switches are included to handle aspects of data-driven interpretation. They behave differently under all three interpreta-

tions. Section 9.0 contains the pseudo code (operational semantics) for each of the nodes discussed in the following subsections.

TABLE 1. Node use in a PDW

Node Type	Interpretation		
	Control-Driven	**Data-Driven**	**Demand-Driven**
Entry	(specifies initial control dependence region)	(not used)	
Exit	(specifies end of computation)		
operator	The behavior of these nodes under the three interpretations is straightforward and is thus not summarized. They are included here for completeness' sake.		
predicate			
read, write			
Switch	PassThrough-IgnoreP[a]	Switch	$Switch_{demand}$
η^F		$Switch^F$	η^F
g		γ_{data}	g
b		b	β_{demand}

a. That is, ignore the token waiting on the input arc *P* and pass through to output the one token waiting on the other arc(s). Under control-driven interpretation there is guaranteed to be exactly one token waiting on exactly one arc for each of these nodes when it is their turn to execute. The pseudo-code is shown in Section 9.4.

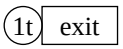
Each element is described below in detail and accompanied by a graphical representation. Control dependences are shown via circles to the left and/or right of the node. The circle to the left indicates the control dependence region the node is in. A circle to the right of the node, if present, indicates the control dependence region for which the node is the root. Many of the arc labels in the diagrams are abbreviated to save space. The arcs shown in the diagrams are data arcs. In all cases the input arc labelled *P* is connected to the output of the predicate for the conditional or the loop.

4.1.1 Entry

Entry is the "first" node of the PDW. The Entry node is the source for one control dependence arc, labelled *true* by default. Only Entry and predicates presented below, are sources of control dependence arcs. Entry is neither the source nor the target of a data arc.

Under control-driven interpretation, execution begins with Entry and follows the control dependence arcs. Under data- and demand-driven interpretation Entry is not used.

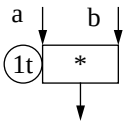
4.1.2 Exit



Exit is the "last" node of a PDW. Unlike Entry, Exit is the target of one control dependence arc and the source of none. Like Entry, Exit is neither the source nor the target of a data arc.

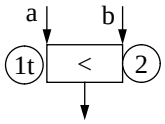
Under control-driven interpretation, encountering the Exit node signals the end of the computation. Under data- and demand-driven computation the Exit node is not used.

4.1.3 Operators



Binary operators are the target of two data arcs and they are the source of one data arc; unary operators are the target and source of one data arc. Pseudo-code for the *multiply* (strict) and the *or* (non-strict) operator is shown in Section 9.1.

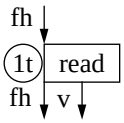
4.1.4 Predicate



A predicate is the target of two data arcs and the source of one data arc. Predicates represent the usual relational and boolean operators.

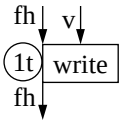
A PDW for a one-arm conditional statement, such as `IF-pred-THEN-body`, has vertices in the control dependence^{true} region for which the predicate is the source but no vertices in the corresponding control dependence^{false} region. A PDW for a two-arm conditional statement, such as `IF-pred-THEN-body-ELSE-body`, has vertices in both regions. PDW loops may have vertices in both of the predicate's output control dependence regions.

4.1.5 Read & Write



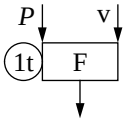
The read node is the target for one data arc, labelled *fh* (for "file handle"), and is the source of two data arcs, labelled *fh* and *v*. The *fh* arc connects I/O nodes on the same file.

The write node is the target for two data arcs, labelled *fh* and *v*, and is the source of one data arc, labelled *fh*.



Under data-driven interpretation the *fh* arc sequentializes I/O. Under control- and demand-driven interpretation the *fh* arc is not needed for sequentialization but to distinguish I/O on one file from I/O on another. This distinguishing information could be hard-coded into the node or it could be passed in the token traveling along the *fh* arc. This paper assumes the latter. As a consequence in the pseudo-code in Section 10.3 and 10.4 the appropriate *wait*, *demand*, *consume*, and *output* functions are provided.

4.1.6 Switches: Switch^{T} & Switch^{F}



Switch nodes are the target of two data arcs, labelled *P* and *v*, and are the source of one data arc. The primary purpose of the Switch node is to control the flow of values in to an area of the graph under data-driven interpretation. In the original definition the behaviour of the Switch^{T} (Switch^{F}) node when the predicate is *false* (*true*) is unspecified. In our definition the Switch^{T} (Switch^{F}) node consumes values and produces no output when the predicate is *false* (*true*).

Under control-driven interpretation the Switch nodes pass the input on *v* to their output, ignoring *P.*

Under data-driven interpretation the Switch^F (Switch^T) node copies *v* to its output only when *P* is *false* (*true*); otherwise it generates no output.

Under demand-driven interpretation the Switch nodes pass the input on *v* to their output, ignoring *P,* but they must first discard unsolicited values. In conditionals these unsolicited values accrue on Switch^T (Switch^F) nodes when the source which feeds the *v* arc in addition feeds

i) the predicate of the conditional, or

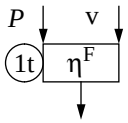
ii) a Switch^F (Switch^T) node in that same conditional, or

iii) both i) and ii).

In each case a value arrives at a Switch node every time the conditional executes, regardless of how the predicate evaluates. In order for the Switch to operate correctly it must first discard these unsolicited values.

In loops a *false* token is always left on each Switch^T node when the loop terminates. Switch^F nodes are never used in loops. In certain situations there may be many *false* tokens from the predicate waiting on the Switch^T nodes and there may be as many data tokens on their *v* input arcs. These data tokens represent unwanted values. The Switch^T nodes are not demanded until the predicate finally evaluates to *true*. When it does, the Switch^T nodes discard all of the waiting *false* tokens and unwanted values, if any. Section 6.2.4 provides an illustration of one of these situations.

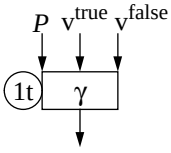
4.1.7 η^F node



The η^F node controls the passing of final values out of a loop. The η^F node is the target of two data arcs, labelled *P* and *v,* and is the source of one data arc.

Under control-driven interpretation the η^F node passes the input on *v* to its output, ignoring *P.* Under data-driven interpretation the η^F node behaves like the Switch^F node. Under demand-driven interpretation the η^F node generates demands until it receives a *false* on *P,* discarding intermediate values on *v.* The capability of generating demands is what separates the η^F nodes from the Switch^F nodes.

4.1.8 γ node



The γ node is placed at the output of a conditional. The γ node is the target of three data arcs labelled *P,* v^{true} and v^{false}, and is the source of one data arc.

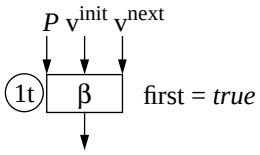
Under control-driven interpretation the γ node passes the incoming value on v^{true} or v^{false} to its output, ignoring *P.*

Under data-driven interpretation the γ node creates an output stream from its v^{true} and v^{false} input streams using the input stream on *P* as a guide. PDW data arcs act as queues under data-driven interpretation (see Section 4.2) so it is possible for multiple values to

be waiting at the γ nodes' input arcs. This input needs to be passed on in the order specified by the input on the *P* arc.

Under demand-driven interpretation the γ node generates a demand for the evaluation of a predicate. Based on the result of this demand, the γ node generates a demand on either the *true* arm, via v^{true}, or the *false* arm, via v^{false}.

4.1.9 β node



The β node controls the initial and subsequent values of a variable used in a loop. The β node is the target of three data arcs, labelled *P*, v^{init} and v^{next}, and is the source of one data arc; it also has a boolean state variable, named "first," which is initially set to *true*. The state variable is needed to keep track of the iteration[1] of the loop under data-driven interpretation. The β node is not strict in the value of the predicate *P*. The input arc v^{init} holds the initial value of a variable before the loop begins execution; the arc v^{next} holds the value of the variable at the beginning of a subsequent iteration of the loop. The placement of the β node is identical to the μ except that it is control dependent on the region containing the predicate of the loop.

Under control-driven interpretation the β node passes the incoming values on v^{init} or v^{next} to its output, ignoring *P*.

Under data-driven interpretation the β node behaves as described below in Table 2. Interpretation begins with checking the value of the state variable "first". Initially, since first is *true*, the β node is waiting in **state 1** for a value to arrive on its v^{init} line. When a value arrives, the β node consumes the value, outputs the same value, and sets first to *false*. This is the initial value for an execution of the loop. Each succeeding *true* token on the predicate line indicates the start of another iteration of that same execution of the loop, so the β node moves to **state 2**: it consumes the value on *P* and consumes and outputs the value on v^{next}, leaving first set to *false*. The next *false* token on the *P* line indicates that the current execution of the loop has finished its last iteration, so the β node moves to **state 3**: it consumes the value on *P*, generates no output, and sets first to *true*, ready now to let another initial value in to the loop

Under demand-interpretation the β node does not use its state variable "first". There will always be a token waiting on the node's *P* input arc except for the very first time that the node ever executes in the current invocation of the PDW. On that very first time, recognizable by an absence of a token on the *P* input arc, the node demands a value from v^{init}. Otherwise there will be a token waiting on the *P* input arc. A *false* token on this arc indicates that the previous execution of the loop has terminated and that the current execution wants an initial value: the β node consumes *P* and demands a value from v^{init}. A

[1] In this paper *iteration* refers to a single execution of a loop body. A loop *execution* includes all iterations, of which there may be none if the loop is a zero-trip.

TABLE 2. State transitions of the β node under data-driven interpretation

value of predicate P	value of state variable "first"	
	true	false
true	**state 1**: first time[a] consume (v^{init}) [ignore v^{next}] [ignore P][b] first = *false* output (v^{init})	**state 2**: other times [ignore v^{init}] consume (v^{next}) consume (P) [first stays *false*] output (v^{next})
false		**state 3**: last time [ignore v^{init}] [ignore v^{next}] consume (P) first = *true* [no output]

a. The β node is in this state when it executes for the "first time" in any execution of the loop.

b. The term "ignored" in this context means that if a token is waiting on the associated arc, then that token is not consumed and its value is not considered.

true token, on the other hand, indicates that the current execution of the loop needs a subsequent value: the β node consumes *P* and demands a value from v^{next}.

4.2 Arcs

There are two types of arcs in a PDW: control dependence arcs and data arcs. The former connect nodes sharing common control dependence regions, the latter carry data from one node to another. These arcs behave differently under the three interpretation schemes, as presented in Table 3. The data arcs of concern for interpretation are flow

data-dependences which act as data conduits. The PDW also contains output-, anti- [5] and def-order [7] dependences which are carried over from the PDG.

TABLE 3. Arc Behaviour and Usage

Arc Type	Interpretation		
	Control-Driven	**Data-Driven**	**Demand-Driven**
control dependence	determine regions which are candidates for execution	(not used)	
data[a]	behaves like an ordinary arc[b]; not used, except to communicate data between nodes [c] representing operations	behaves like an unbounded queue;	behaves like an unbounded queue; used to direct demands (i.e., presumes the capability to proceed "backwards" from target to source)

a. Under all interpretations, data arcs provide data communication paths between nodes and they are the *only* means of doing so in a PDW.

b. At most one value can exist on the arc. Placing a subsequent value on the arc overwrites the current value.

c. They can be used to determine the valid execution sequence within a control dependence region.

5.0 Interpretation Mechanics

Interpreting the PDW under control-, data- or demand-driven disciplines is achieved by attributing different behaviour for arcs and by providing a different interpretation mechanics (semantics) to each node. A summary is presented below in Table 4.

TABLE 4. Interpretation Mechanics

Question	Interpretation		
	Control-Driven	**Data-Driven**	**Demand-Driven**
Which node is the first to execute?	the Entry node	any node which has enough of its inputs to execute	the demanded nodes
What characterizes executable node(s) at each subsequent execution step?[a]	within a control dependence region, any node which has its data dependences satisfied		any leaf of the demand graph
When is execution complete?	when the Exit node is encountered	when no node can execute	when the original demands are fulfilled

a. We presume under all interpretations that transmission along data arcs is always faster than the determination of which node is to execute next.

5.1 Control-Driven

The Entry node is the first node to execute because it is the first control dependence region. Nodes in the same region that share no data dependences can execute concurrently or in any order. Nodes data dependent on other nodes must follow their data-dependence predecessor(s) in execution. There are techniques available to extract a correct execution ordering among nodes [12]. When the Exit node is encountered, computation is terminated.

5.2 Data-Driven

The first node to execute and the candidates for execution at each subsequent execution step are the nodes which have enough of their inputs to execute. As long as at least one node can execute, execution is not complete.

5.3 Demand-Driven

Demand-driven execution begins because the output of some nodes are “demanded” from the world outside the program. The demanded nodes under this scheme form a

graph. The nodes with in-degree of zero are the first demanded nodes; the interior nodes are those which have generated demands and still have at least one of those demands unfulfilled; the leaves are nodes which have either not yet issued any demands, or have had all their demands fulfilled but have not yet executed. Any leaf is a candidate for execution. When all the nodes with in-degree zero have had their demand fulfilled execution can halt.

6.0 Generic Structures and Concrete Examples

This section presents the generic structures (topology) and concrete examples for a conditional and loop constructs in a PDW.

Arc junctions in the figures represent a fan-out: under all interpretations, tokens traveling along the arc are duplicated to all branches of the junction. This is like the *split* operator available in dataflow machines [9]. Arc labels, such as *P*, v^{init}, and so on, are not shown in the figures below; however the relative ordering is preserved. That is, for example, the input arcs of a β node are always from left to right *P*, v^{init}, and v^{next}. Data constants are shown explicitly. For example, the input for the rightmost arc to the predicate node in Figure 4 is a constant. Under control- and data-driven execution the constant will keep a supply of at least one zero on that data arc; under demand-driven execution it will produce a zero only when a value is demanded on that arc. The highlighted portions in the figures are meant to reflect similarities between PDW nodes and *encapsulators* [11].

6.1 Conditionals

6.1.1 Generic Conditional Structure

Shown below in Figure 3 is the generic structure of PDW conditionals. The generic structure for conditionals is a predicate, a row of $Switch^T$ nodes one for each variable used in the *true body* and $Switch^F$ nodes one for each variable used in the *false* body, a *true* body and a *false* body, and finally a row of γ nodes, one for each variable which could be changed by the conditional and is used afterwards. Note that the control dependence region for the $Switch^T$ nodes and the *true* arm is 2t, and for the $Switch^F$ nodes and the *false* arm it is 2f. Note also that the control dependence region for the γ nodes is 1t. The *v* input lines for the Switch nodes are not extended: this is intended to suggest the possibility of either one or more of these nodes getting their *v* input from the same source as the predicate and/or a $Switch^T$ node and a $Switch^F$ node getting their input from the same source. In either case such a node is guaranteed to receive input every time the conditional executes. This is of interest under demand-driven interpretation, as is pointed out in the example further below.

FIGURE 3. Generic Conditional structure

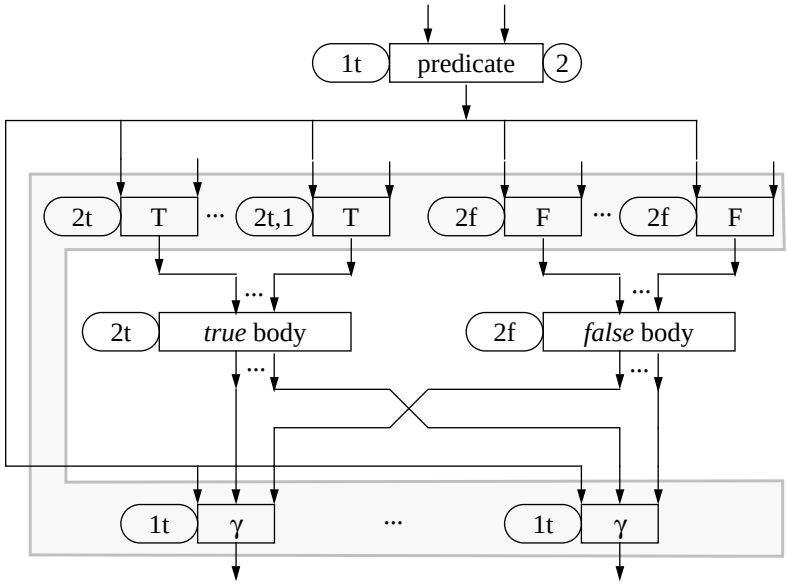


FIGURE 4. PDW fragment for Conditional Example

```
x = ...;
y = ...;
if (y > 0)
   x = x + y;
else
   x = y - 5;
... = x * y;
```

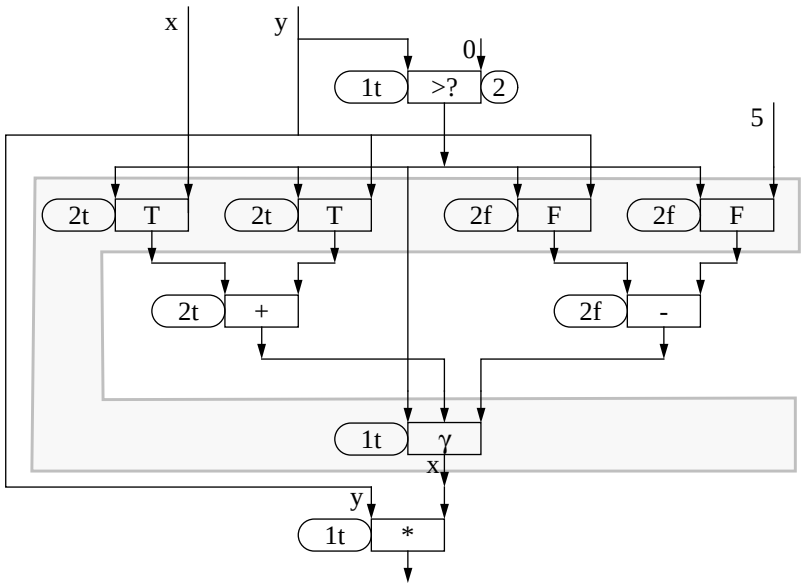


Figure 4 shows an `IF-pred-THEN-body-ELSE-body` statement and the corresponding PDW fragment.

6.1.2 Control-Driven Interpretation

Under control-driven interpretation execution starts in the region 1t which encloses the conditional. Suppose the predicate evaluates to *true*. Control enters region 2t. The Switch^{T} nodes are the first to execute. Since they share no data dependences they can execute in either order (or in parallel). They transmit their *v* input, ignoring their *P* input. The [+] node, which is data-dependent on the Switch, can thus execute, passing the sum to the γ node. Control returns to region 1t. Ignoring its *P* input, the γ node passes its input on its v^{true} arc out of the conditional. Whichever way the predicate evaluates, after the execution of the conditional completes, tokens will be left remaining at one of the group of Switch nodes, either the Switch^{T} group or the Switch^{F} group. This is not a problem under control-driven interpretation because data arcs act as arcs, not queues, so no more than one token will ever accumulate on any arc -- the new values overwrite the old ones before the Switch nodes execute again.

6.1.3 Data-Driven Interpretation

Under data-driven interpretation all of the nodes are data dependent (eventually) on the predicate node. After the predicate node executes, all of the Switch nodes can execute, assuming a value for x is available (a value for y will be available since the predicate executed, and the "5" constant keeps its output line supplied with a 5). Supposing that the predicate evaluates to *true* the Switch^{T} nodes pass on their data while the Switch^{F} nodes absorb theirs, allowing data to arrive only at the [+] node. This node executes and passes the sum to the γ node, which, ignoring its *P* input, passes its input on out of the conditional. Note that the [-] node did not execute.

6.1.4 Demand-Driven Interpretation

Under demand-driven interpretation execution of the conditional begins with the [*] node. The [*] node demands[2] the γ node which demands the predicate which demands the node which provides a value for y and the "0" constant. The node providing the value for y is outside the Figure. When this value is provided, the predicate can complete its execution.

At this point, immediately after the predicate executes, there is at least one token from the predicate on the *P* input arc of all of the Switch nodes and exactly one token on the *P* arc of the γ node. There is also at least one data token (the latest value for y) on the *v* input arc of each of the innermost Switch nodes and exactly one token (with the same value) on the left input arc of the [*] node. There are no tokens waiting on the *v* input arc of the outermost Switch nodes.

The γ node now continues execution. It is presumed again that the predicate resolves to *true*. The γ node demands the [+] node which demands the two Switch^{T} nodes which

2. This is shorthand for "places a demand on the output of."

demand the predicate[3] the node providing the value for x and the node providing the value for y.

When the Switch^T nodes have had their demands fulfilled and have completed execution, the [+] node, the γ node, and the [*] node can, in sequence, complete execution, completing execution of the conditional and fulfilling the original demand. Note that there may be tokens left waiting in the conditional.

Suppose that for the past *n* executions of this conditional the predicate has evaluated to *true* every time. As a result there would be exactly *n true* tokens waiting on the two Switch^F nodes in the Figure. Since the leftmost Switch^F node receives a value on its *v* input arc each time the conditional executes, there would also be exactly *n* tokens on that arc of that node. But note that there would not be any tokens waiting on the *v* input arc of the rightmost Switch^F node. Since the "5" constant has not been demanded in any of the past *n* executions, it has not produced any output.

Finally, suppose that on the next execution the predicate evaluates to false. When the Switch^F nodes are demanded, they have to discard unwanted values. The leftmost Switch^F node discards all *n* of its values; the rightmost Switch^F node has none to discard. The reader is encouraged to examine the pseudo-code for demand driven interpretation of Switch nodes.

6.2 Loops

6.2.1 Generic Loop Structure

Shown below in Figure 5 and Figure 6 are the generic PDW structures for pre-test and post-test loops, respectively. The constituting elements are the same in both cases: there is a row of β nodes, a row containing both Switch^T nodes (one for each variable which can be used in the loop) and η^F nodes (one for each variable that is both used after the loop and which could be changed by the loop), a predicate and a loop body. In both cases the row of β nodes precedes the other loop elements. In pre-test loops the predicate precedes the Switch^T and η^F nodes which in turn precede the body. In post-test loops the body precedes the predicate which in turn precedes the Switch^T and η^F nodes. The box around "body" in both Figures is a simplification.

[3] Actually, one η^F node places the demand; the demand from the other η^F node is discarded since by that time the predicate has already been demanded. The rationale for this is that when the predicate executes it will automatically fulfill both demands. Thus demands do not accumulate.

FIGURE 5. Generic Structure for Pre-Test Loops

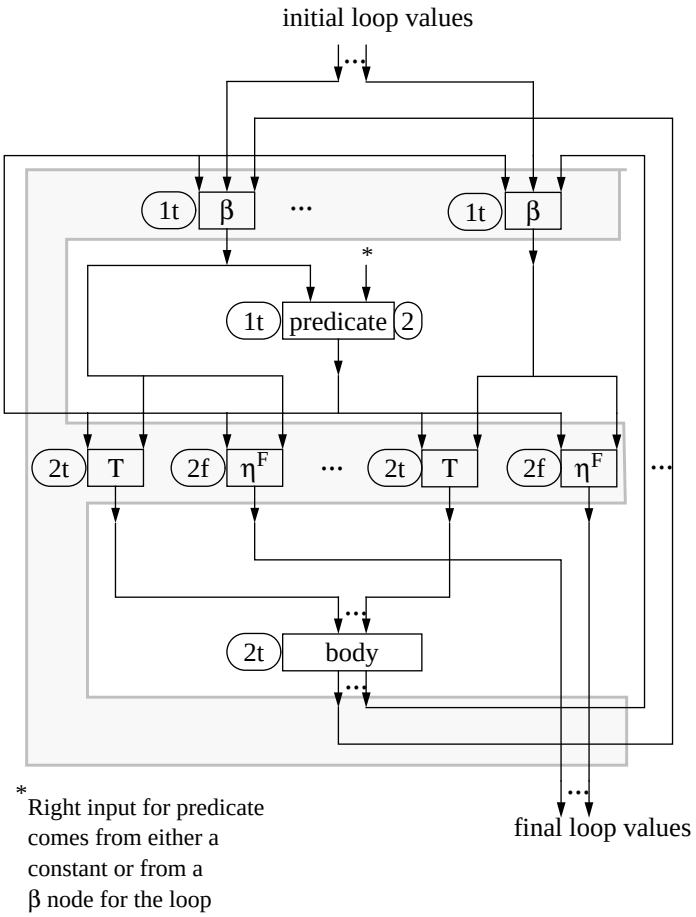
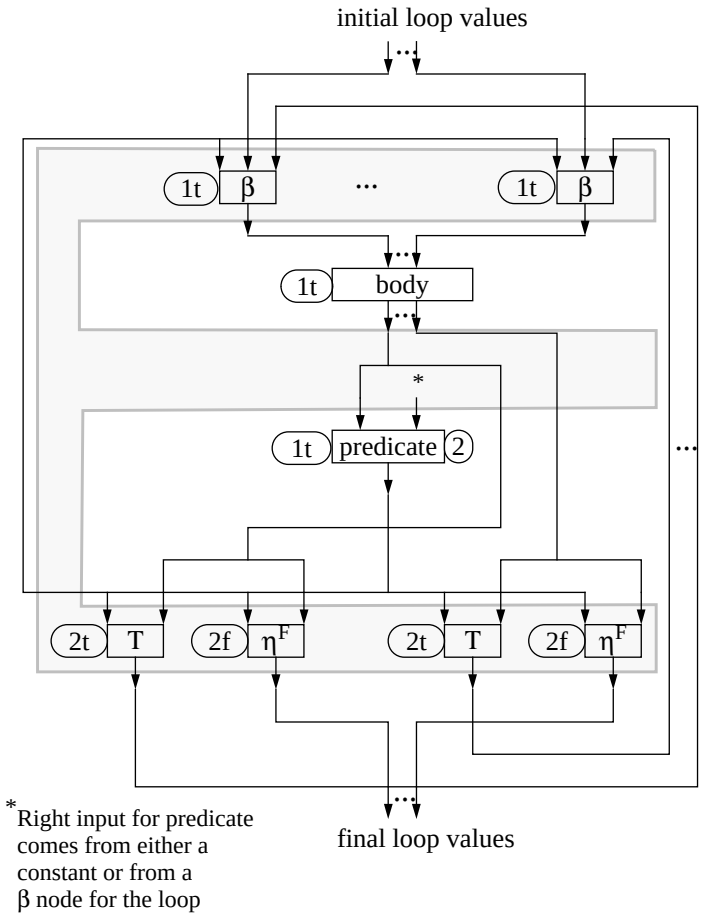


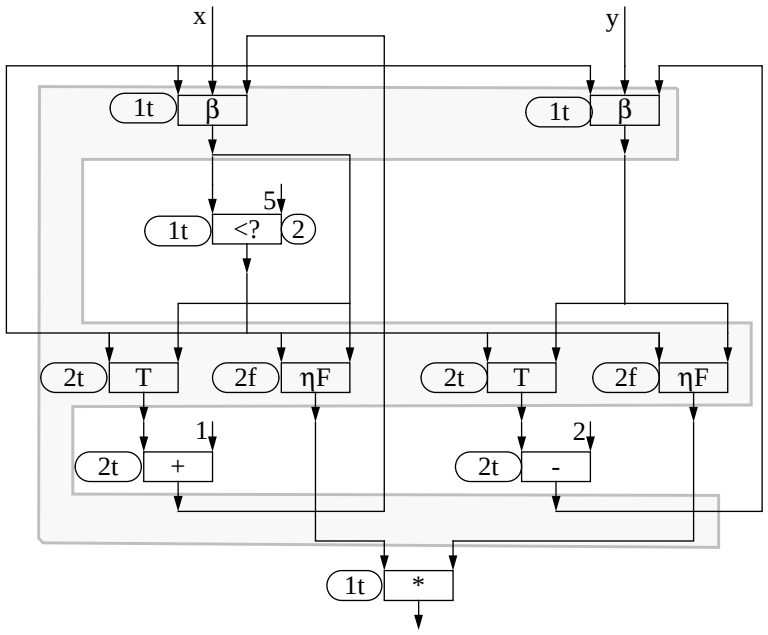
FIGURE 6. Generic Structure for Post-Test Loops



Consider the pre-test loop and the PDW fragment shown in Figure 7. Interpretation under all three modes for this example is explained below.

FIGURE 7. PDW fragment for Pre-Test Loop Example

```
x = ...;
y = ...;
while (x < 5) {
  x = x + 1;
  y = y - 2;
}
... = x * y;
```



6.2.2 Control-Driven Interpretation

Control-driven execution begins in the region 1t containing the β nodes and the predicate.

If the predicate evaluates to *true*, execution proceeds as described in the next paragraph; otherwise execution terminates as described in the second paragraph below.

The predicate has evaluated to *true*, control enters region 2t. The Switch^T nodes execute, followed by the [+] and [-] nodes (this is the body of the loop), followed then by the β nodes, and followed finally by the predicate again. This concludes a loop iteration. Execution proceeds as described in the paragraph above.

The predicate evaluated to *false* so control enters nested region 2f. The η^F nodes execute. Control returns to the region prior to 2f, which is 1t. The [*] node, data dependent on the η^F nodes can execute.

6.2.3 Data-Driven Interpretation

Data-driven execution is straightforward. The β nodes allow in one initial loop value, then circulate subsequent iteration values while the predicate continues to resolve to *true*. When the predicate resolves to *false* the η^F nodes pass the final loop values out of the loop and the β nodes reset to prepare for the initial value of the next execution of the loop.

Both the Switch^T and η^F nodes *are* necessary. If the former were absent, then, at the end of each loop's execution, an additional set of data tokens would be added to a pool of extraneous tokens which would continue in the loop and would wreck havoc with the semantics of the loop. If the latter were absent, then data tokens would never leave the loop. Note finally that in the example, if values were available for x, then the loop could proceed through many loop executions, each with many iterations, even if values for y were not available. Values from the predicate will queue up on the appropriate β, Switch^T, and η^F nodes. When values for y subsequently became available, these queued tokens will cause this part of the loop to iterate for the same number of iterations per execution as the loop did with values of x.

6.2.4 Demand-Driven Interpretation

To simplify the description below, a kind of lock-step parallel execution of the PDW is employed. However, since the demanded nodes form a graph and any leaf is executable, many other execution sequences are possible. The reader may find it instructive, for example, to pursue demands in a depth-first fashion, following the rightmost β node first; note that tokens arriving unsolicited at not-yet-demanding nodes guarantee the same computation as the one traced with our lock-step parallel approach.

Under demand-driven interpretation execution begins with a demand on the [*] node which demands each of the η^F nodes which in turn demand the predicate and each of the β nodes. The predicate demands the leftmost β node (already demanded) and the "5" constant which fulfills its demand immediately. The β nodes demand the nodes at the other end of their v^{init} arcs. These last nodes are outside the Figure. When these demands are fulfilled, the β nodes and the predicate, in sequence, can complete execution.

At this point all of the demands inside the loop have been fulfilled and both of the η^F nodes are ready to continue execution. There is exactly one token from the predicate waiting on the *P* arc of both of the β nodes and both of the η^F nodes, and at least one token from the predicate waiting on the *P* arc of both of the Switch nodes. There is also at least one value token waiting on the *v* input arc of both of the Switch nodes and exactly one value token waiting on both of the η^F nodes. If the predicate evaluates to *true*, then execution proceeds as described in the next paragraph; otherwise execution terminates as described in the second paragraph below.

The predicate has evaluated to *true* the η^F nodes demand the predicate and each of the β nodes. The predicate in turn demands the leftmost β node (already demanded) and the "5" constant, all as they did the first time. The β nodes, however, now demand the predicate. These demands are fulfilled by the *true* tokens from the previous execution of the predicate which are already waiting for each β node. Since these tokens are *true*, the β nodes demand the [+] and [-] nodes at the end of their v^{next} arcs. These nodes in turn demand both Switch nodes and the "1" and "2" constants, respectively. The Switch nodes then demand the predicate and the β nodes -- all four of these demands are fulfilled by tokens waiting from the previous execution of the predicate and β nodes. All of the tokens which were waiting on loop nodes, as described in the previous paragraph, have now been consumed. Since the token from the predicate is *true*, the Switch nodes pass on their input from the β nodes. The [+] and [-] nodes, the β nodes and the pred-

icate can, in sequence, complete execution. Execution proceeds as described starting with the previous paragraph.

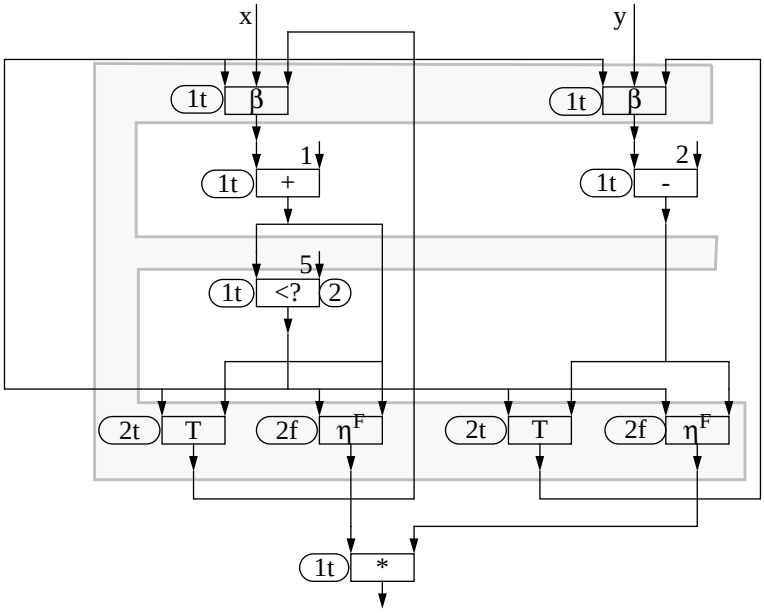
The predicate has evaluated to *false* so the η^F nodes pass their *v* input on to their output and the loop terminates. The demands on the two η^F nodes have been fulfilled and the $\boxed{*}$ node can then execute. Note, however, that there are *false* tokens left waiting, exactly one total for each of the β nodes and at least one for each of the Switch nodes. In addition there will also be at least one value tokens waiting for each of the Switch nodes. When the β nodes next execute they will consume the *false* token waiting on their *P* input arc and proceed to demand a value on their v^{init} arc. β nodes do not use their state variable "first" under demand-driven interpretation. When the Switch nodes next execute they will use the *false* tokens waiting on their *P* arc to discard the old and now-unwanted values on their *v* arc.

To see how there could be more than one set of tokens waiting on the Switch nodes when execution of the loop begins, consider the following. Suppose that the past *n* executions of this loop were zero trip. In this case the predicate was evaluated only once for each execution and it evaluated to *false* each time. On each of these executions one *false* token and one value token was sent to each of the two Switch nodes in the Figure, but, because the predicate evaluated to *false* each time, neither of these Switch nodes were demanded on any of those *n* executions. As a result, *n* pairs of tokens were left waiting on these two nodes. If the next execution of the loop is not zero trip, the predicate will evaluate to *true* and the Switch nodes will discard all of their waiting *false* tokens and the associated value tokens before generating another demand.

Figure 8 shows the code and the corresponding PDW for a post-test loop. The previous section explained how a pre-test loop executes under control-, data- and demand-driven interpretation. With appropriate changes that discussion is applicable to the post-test loop example presented here.

FIGURE 8. PDW fragment for Post-Test Loop Example

```
x =...;
y =...;
do {
 x = x + 1;
 y = y - 2;
} while (x < 5);
... = x * y;
```



7.0 Conclusions and Future Work

This paper presents an operational definition of the PDW elements. To alleviate problems in the original papers, the μ node has been replaced by the β operator. Using the revised definition a front-end that maps C-like programs (with parallel constructs) into a PDW is being developed at the University of New Mexico. This front-end will be used to generate both a partial evaluator for scientific codes and a static performance predictor that will match computations with machines. The front-end may be used for other experimentation and analysis of the PDW.

The potential benefit of these projects include a better analysis of the performance of programs on conventional architectures by simulating their execution in data- and demand-driven fashion. The speedup curve for an idealized execution on a simulator under a varying number of processors provides a realistic asymptote for the potential parallelism in executing a program. This curve will be much more useful in gauging the effectiveness of manual or automatic parallelization on conventional architectures than is the overly optimistic linear speedup asymptote. We hope to lay the foundation of a compiler that uses performance information gleaned from simulations in order to support better automatic partitioning for conventional MIMD architectures.

8.0 Acknowledgments

Many thanks to Dr. Arthur B. Maccabe for invaluable clarifications regarding the PDW.

9.0 References

[1] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages, In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 257-271, June 1990.

[2] Robert A. Ballance and Arthur B. Maccabe. Program Dependence Graphs for the Rest of Us, UNM Technical Report No. CS92-10, October 28, 1992, University of New Mexico, Albuquerque, New Mexico.

[3] Robert Cartwright and M. Felleisen. The Semantics of Program Dependence, Revision of paper which appeared in *Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 13-27, 1989.

[4] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 451-490, July 1991.

[5] Jeanne Ferrante, Karl J. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization, *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319-349, July 1987.

[6] Jean-Luc Gaudiot and Lubomir Bic. Advanced Topics in Data-Flow Computing, Prentice-Hall, New Jersey, 1991.

[7] S. Horwitz, J. Prins, and T. Reps. On the Adequacy of Program Dependence Graphs for Representing Programs, In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pp. 146-157, 1988.

[8] Karl J. Ottenstein and Steven J. Ellecy. Experience Compiling Fortran to Program Dependence Graphs, *Software-Practice and Experience*, Vol. 22(1), pp. 41-62, January 1992.

[9] David Skillicorn. Techniques for Compiling and Executing Dataflow Graphs, *Information Processing*, pp 27-32, 1989.

[10] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. Data-Driven and Demand-Driven Computer Architecture, *Computing Surveys*, Vol. 14, No. 1, pp. 93-143, March 1982.

[11] Kenneth R. Traub. Implementation of Non-Strict Functional Programming Languages, The MIT Press, Cambridge, 1991.

[12] Jeanne Ferrante, Mary Mace, and Barbara Simmons. Generating Sequential Code from Parallel Code, In *International Conference of Supercomputing*, St. Malo, France, pp. 582-592, 1988.

[13] Leah H. Jamieson. Using Algorithm Characteristics to Evaluate Parallel Architectures, In *Performance Evaluation of Supercomputers*, J.L. Martin (editor), Elsevier Science Publishers B.V., North Holland, pp. 21-49, 1988.

10.0 Appendix: The Pseudo-Code

Following is a brief description of the functions used in the pseudo-code:

`wait-for-token-on(a,b)` waits at this statement until at least one token has arrived on either of the input arcs labelled `a` or `b`; if there is already at least one token on one or both of the arcs, then the node does not wait.

`consume(a,b)` removes one token from arc `a` *and* one token from arc `b`.

`output(a)` generates on its output arc the value of the last token consumed on arc `a`.

$\texttt{output}_{\texttt{fh}}\texttt{(v)}$ generates on its output arc labelled `fh` the value of the last token consumed on arc `v`.

`output(a*b)` generates on its output arc the value of the last token consumed on arc `a` multiplied by the value of the last token consumed on arc `b`.

`demand(a,b)` generates a demand for one token on arc `a` and a demand for one token on arc `b`, then waits at this statement until at least one token arrives on each arc; if there is already at least one token on both arcs, then the node does not wait.

`if (token on a)` takes the *true* branch if there is at least one token waiting on the input arc `a`.

`if (a = true)` takes the *true* branch if the value of the last token consumed off of arc `a` is *true*.

10.1 Operator

Control-driven Interpretation of Strict Operator (Multiply):

```
consume(a,b);
output (a*b);
```

Data-driven Interpretation of Strict Operator (Multiply):

```
wait-for-token-on (a,b);
consume(a,b);
output (a*b);
```

Demand-driven Interpretation of Strict Operator (Multiply):

```
demand (a,b);
consume(a,b);
output (a*b);
```

Control-driven Interpretation of Non-Strict Operator (Or):

```
consume (a);
if (a = true)
          output (a);
```

```
else {
          consume(b);
          output (b);
}
```

Data-driven Interpretation of Non-Strict Operator (Or):

```
wait-for-token-on (a,b);
if (token on a) {
     consume (a);
     if (a = true)
          output (a);
     else {
          wait-for-token-on (b);
          consume(b);
          output (b);
     }
}
else {
     consume (b);
     if (b = true)
          output (b);
     else {
          wait-for-token-on (a);
          consume(a);
          output (a);
     }
}
```

Demand-driven Interpretation of Non-Strict Operator (Or):

```
if (token on a) {
     consume (a);
     if (a = true)
          output (a);
     else {
          demand (b);
          consume(b);
          output (b);
     }
}
else if (token on b){
     consume (b);
     if (b = true)
          output (b);
     else {
          demand (a);
          consume(a);
          output (a);
     }
}
else {
     demand (a);
     consume(a);
     if (a = true)
```

```
            output (a);
        else {
            demand (b);
            consume(b);
            output (b);
    }
```

10.2 Predicate

Control-driven interpretation (using <):

```
consume(a,b);
output (a < b);
```

Data-driven interpretation (using <):

```
wait-for-token-on (a,b);
consume(a,b);
output (a < b);
```

Demand-driven interpretation (using <):

```
demand (a,b);
consume(a,b);
output (a < b);
```

10.3 Read

Control-driven interpretation:

```
consume (fh);
read (fh); /* perform the read */
output_fh(fh);
output_v (v); /* where v is the value read in */
```

Data-driven interpretation:

```
wait-for-token-on (fh);
consume(fh);
read (fh);
output_fh(fh);
output_v (v);
```

Demand-driven interpretation:

```
demand (fh);
read (fh);
output_fh(fh);
output_v (v);
```

10.4 Write

Control-driven interpretation:

```
consume (fh,v);
write (fh,v); /* perform write */
output(fh);
```

Data-driven interpretation:

```
wait-for-token-on (fh,v);
consume (fh,v);
write (fh,v);
output(fh);
```

Demand-driven interpretation:

```
demand (fh,v);
consume(fh,v);
write (fh,v);
output(fh);
```

10.5 PassThrough-IgnoreP

Control-driven Interpretation of Switch, η^F, γ and β nodes:

```
/* Find the one arc with the one token on it, consume and output
   that token. Do not consider arc P. Under control-driven
   interpretation there is guaranteed to be exactly one token on
   exactly one arc (ignoring P). Let “a,” “b,” “c,”... represent the
   labels of the arcs.
*/
if (token on a) {
     consume(a);
     output (a);
}
else if (token on b) {
     consume(b);
     output (b);
}
else if (token on c) {
...
```

10.6 Switch

Data-driven Interpretation of Switch^F node:

```
wait-for-token-on (P,v);
consume (P,v);
if (P = false) /* for switchT: if (P = true)... */
     output (v);
```

Demand-driven Interpretation of Switch^F Node (same as Switch$_{demand}$):

```
demand (P);
consume(P);
while (P = true) {
     /* for SwitchT: while (P = false)... */
     /* discard unwanted values; */
     if (token on v) {
     consume(v);
     }
     demand (P); /* a token must be present on P */
     consume(P);
}
demand (v);
consume(v);
output (v);
```

10.7 η^F

Data-driven Interpretation of η^F node (same as Switch^F):

```
wait-for-token-on (P,v);
consume (P,v);
if (P = false)
     output (v);
```

Demand-driven Interpretation:

```
do {
     demand (P,v);
     consume(P,v);
} while (P = true);
output(v);
```

10.8 γ

Data-driven Interpretation of γ node (γ_{data}):

```
wait-for-token-on (P);
consume (P);
if (P = true) {
     wait-for-token-on (v^true);
     consume(v^true);
     output (v^true);
}
else {
     wait-for-token-on (v^false);
     consume(v^false);
     output (v^false);
}
```

Demand-driven Interpretation:

```
demand (P);
```

```
consume(P);
if (P = true) {
     demand (v^true);
     consume(v^true);
     output (v^true);
}
else {
     demand (v^false);
     consume(v^false);
     output (v^false);
}
```

10.9 β

Data-driven Interpretation:

```
if (first = true) {
     /* state 1 (see Table 2) */
     wait-for-token-on (v^init);
     consume(v^init);
     first = false;
     output (v^init);
}
else {
     wait-for-token-on (P);
     consume(P);
     if (P = true) {
          /* state 2 */
          wait-for-token-on (v^next);
          consume(v^next);
          output (v^next);
     }
     else {
          /* state 3 */
          first = true;
          /* no output */
     }
}
```

Demand-driven Interpretation of β node (β_{demand}):

```
/* see explanation in text where β node is defined;
note no use of variable “first” */

if (not (token on P)) {
     /* first time ever */
     demand (v^init);
     consume(v^init);
     output (v^init);
else {
     consume (P);
```

```
        if (P = false) {
              demand (v^init);
              consume(v^init);
              output (v^init);
        }
        else {
              demand (v^next);
              consume(v^next);
              output (v^next);
}
```