From Control Flow to Dataflow

Micah Beck
Richard Johnson
Keshav Pingali

TR 89-1050
October 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

This research was supported by an NSF Presidential Young Investigator award (NSF grant # CCR-8958543), NSF grant # CCR-90-08526, and grants from the Hewlett-Packard Corporation and IBM.

From Control Flow to Dataflow

Micah Beck
Richard Johnson
Keshav Pingali
Cornell University

Abstract

Are imperative languages tied inseparably to the von Neumann model or can they be implemented in some natural way on dataflow architectures? In this paper, we show how imperative language programs can be translated into dataflow graphs and executed on a dataflow machine like Monsoon. This translation can exploit both fine-grain and coarse-grain parallelism in imperative language programs. More importantly, we establish a close connection between our work and current research in the imperative languages community on data dependences, control dependences, program dependence graphs, and static single assignment form. These results suggest that dataflow graphs can serve as an *executable* intermediate representation in parallelizing compilers.

[1]This research was supported by an NSF Presidential Young Investigator award (NSF grant #CCR-8958543), NSF grant #CCR-90-08526, and grants from the Hewlett-Packard Corporation and IBM.

1 Introduction

Imperative and declarative schools of parallel computation offer competing approaches to exploiting parallelism. The imperative school takes a conservative position on both languages and architectures — it believes in using conventional, imperative languages like FORTRAN to program interconnections of von Neumann processors [14, 16]. The declarative school, on the other hand, is radical in its approach to both languages and architectures — it believes in using functional or logic programming languages [15, 20] to program dataflow and reduction machines [2, 10, 13]. Are these approaches contradictory and irreconcilable or can we find some middle ground? We are far from being able to answer this question, but to do so, it will be necessary to separate out the effects of language from those of architecture. In particular, we must answer the following question: Are imperative languages tied inseparably to the von Neumann model or can they be implemented efficiently on dataflow machines?

At first sight, dataflow machines appear ill-suited to executing imperative language programs. Traditionally, the dataflow model has been tied closely to functional languages; for example, dataflow operators in both the static and dynamic models of dataflow are functions from inputs to outputs [2, 10]. Moreover, dataflow machines have no program counter to sequence operations; rather, instructions are scheduled dynamically for execution whenever they receive input data. This is in stark contrast to the traditional operational semantics of imperative language programs. Each statement in an imperative language program is a command whose execution causes a change in a global, updatable store. Sequencing of command execution is achieved through a program counter which specifies the unique next instruction to be executed. The existence of a program counter in the underlying operational model is reflected in the programming language through commands such as GOTO's that modify the program counter. Thus, there is a wide gulf between the dataflow model of execution and the standard operational semantics of imperative languages.

Given these differences, is it possible to execute imperative language programs on dataflow machines? Some researchers have proposed to achieve this goal by extending dataflow graphs with imperative operators, and executing the entire graph sequentially using a "thread descriptor" [17] to simulate a program counter. In our opinion, this is really a simulation of von Neumann instruction sequencing on a dataflow machine, which exploits neither the parallelism that can be found in the source program by a parallelizing compiler, nor the potential for parallel execution in the dataflow hardware.

In this paper, we exhibit a translation of imperative language programs into dataflow graphs which can be executed on a dataflow machine like Monsoon [17]. This translation uses information about program dependencies to expose instruction-level parallelism in the dataflow graph. The benefits of this are manifold:

1. For dataflow researchers, our results mean that their machines can be programmed in conventional imperative languages like FORTRAN. This will vastly increase the potential user community of these machines.

2. For imperative language programmers and compiler writers, we offer a parallel model of execution for their programs, in which synchronization is cheap and in which details such as the number of processors, communication network topology, distribution of data structures, etc. are abstracted away. Such a model is ideally suited for measuring the extent to which parallelization techniques can expose parallelism in imperative language programs.

3. We establish a close connection between our work and current efforts in the imperative languages community to define a good intermediate representation for parallelizing compilers. We relate our work to data dependences [19], control dependences, program dependence graphs [11, 7], and static single assignment form [8]. We have used the results in this paper to define an executable intermediate program representation, for both imperative and declarative languages, that appears to be superior to existing ones [18].

The rest of the paper is organized as follows. In Section 2.1, we describe control-flow graphs for a simple imperative language. In Section 2.2, we describe our dataflow model, and in Section 2.3 we show a simple translation of imperative programs into dataflow graphs. This translation does not exploit any parallelism across the statements of the source program. In Section 3, we refine our translation by parallelizing independent memory operations. Section 4 discusses a further refinement that increases parallelism by avoiding redundant control operations; this refinement is related to the notions of control dependences and static single assignment form. The development in these sections ignores aliasing and data structures. However, any realistic scheme for implementing imperative languages on a parallel machine must take these factors into account. In Section 6, we present a translation that handles aliasing and sketch one way to use data dependence information to parallelize array operations. We conclude in Section 7 by contrasting our approach with related efforts.

2 A Framework for Translation

In this section, we show a simple translation of imperative language programs to dataflow graphs. Even though this scheme does not exploit parallelism across statements, it is a useful first step towards deriving the more sophisticated translation presented in Section 3. We also introduce a fairly standard dataflow execution model; with minor changes, the dataflow graphs in this paper can be executed on the Monsoon dataflow machine being built at M.I.T. [17].

2.1 Control-flow graphs

The translation begins with the statement level control-flow graph (CFG) for the source program. Nodes in this graph correspond to statements of three types:

1. *assignments* of the form $x := e$,
2. *forks* of the form **if** p **then goto** l_t **else goto** l_f [3], and
3. *labeled joins* which represent no computation, but are the only nodes in the graph which can be the target of **goto**'s.

An edge from node A to node B indicates that execution of node A may be followed immediately by execution of node B. There is a unique initial node labeled *start* which has no incoming edges, and a unique final node labeled *end*, which has no out-going edges. By convention, an edge is added between *start* and *end*, and thus *start* is a fork in the control-flow graph [11]. Every node

[3] Our development can be generalized to multi-way branches, but we restrict discussion to binary branches for simplicity.

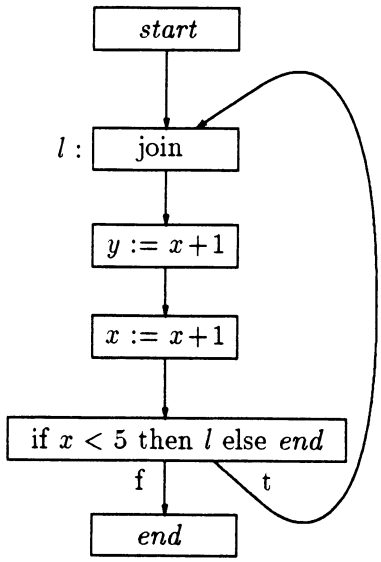


Figure 1: An Example Control-Flow Graph

lies on some path from *start* to *end*. Algorithms for translating high-level language programs into control-flow graphs are well-known [1].

We denote the fact that node N is a predecessor of M in the control-flow graph by $N \rightarrow M$. A non-null control-flow path between N and M is denoted $N \stackrel{+}{\rightarrow} M$. The out-edges of a fork are indexed by a boolean; we refer to this as the out-direction of the edge.

We will use the following program as a running example.

start:
l: join
$y := x + 1$
$x := x + 1$
if $x < 5$ **then goto** *l* **else goto** *end*
end:

Figure 1 illustrates the CFG for this program.

2.2 Dataflow Model

We will use a conventional explicit token store dataflow machine as our model of execution. In this model, each invocation of a procedure and each loop iteration gets an activation context, which is analogous to a stack frame in conventional computers. Frame memory takes the place of the waiting-matching section in earlier dynamic dataflow models: tokens destined for an operator with more than one input will rendezvous at a fixed location in some frame.

One important aspect of our model is the behavior of memory. While every dataflow *machine* has updatable storage locations, the existence of such locations in the implementation is not directly

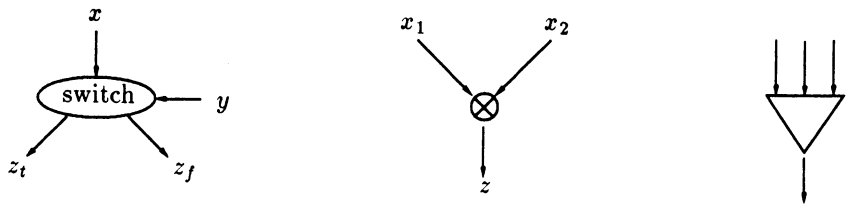


Figure 2: Key To Dataflow Schema Symbols

reflected in standard dataflow *models* [9, 2]. Instead, these models provide an abstraction of storage, such as I-structure storage, in which locations are written into at most once [3]. In our model of dataflow, memory locations can be written more than once. Thus, the result of a read can depend on the order of memory operations; in these cases correct ordering must be observed by the dataflow program graph. To facilitate such ordering, each load and store operation consumes a "dummy" token at its input and generates another at its output when the operation has completed. These tokens are used only to sequence load and store operations; the value they carry is irrelevant. As in all dataflow models, loads and stores are implemented as split-phase operations to avoid blocking the processor pipeline while a memory operation is underway.

In our opinion, this extension makes the dataflow model more useful while preserving the essential feature that execution of the program is accomplished by the concurrent execution of operators that test conditions at their inputs and outputs to determine when to execute.

There is no standard textual representation of dataflow programs. Instead they are represented as graphs, with operations specified at the nodes and the propagation of tokens between nodes shown as arcs. For clarity, we use dotted lines to represent arcs that carry dummy tokens used for coordinating memory operations. General subgraphs are represented by rectangles bearing an appropriate label; expression subgraphs are represented by labeled triangles.

Three important operators are shown in Figure 2. A *switch* is a special operation that has two inputs x and y, and two outputs z_t and z_f. When tokens are present on both x and y, the token on x is output to either z_t or z_f according to the boolean value carried by the token on y. A *merge* operator has inputs $x_1, x_2, \ldots$ and output z. A token arriving on any input is output on z. In the translation, we will use *switch* to model forking of control at forks, and *merge* to model joins. A *synch tree* has n inputs and a single output; once a token has arrived on each of the n inputs, a token is output.

2.3 A Simple Translation

To translate control-flow graphs into dataflow graphs, we apply a compositional translation scheme that translates each assignment and conditional statement in the control-flow graph according to the rules shown in Figure 3. The graphs for the expressions e and p are not shown. A detailed view of the read block in Figure 3 is shown in Figure 4. Joins are simply translated to a dataflow merge. The nodes labeled *start* and *end* in the control-flow graph are translated to nodes with the same labels in the dataflow graph. Figure 5 illustrates the translation for our example

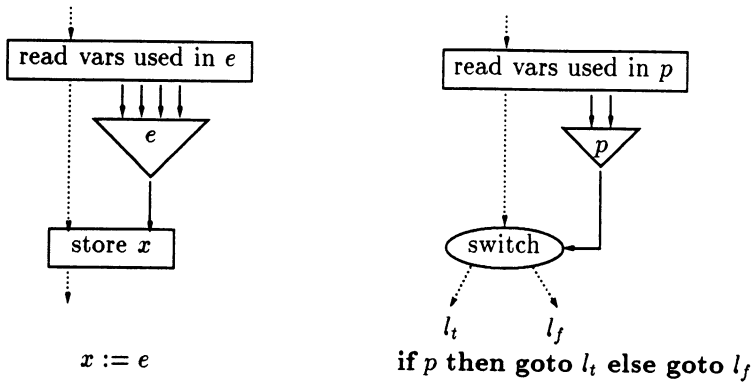


Figure 3: Schema 1 — Implementing Sequential Semantics

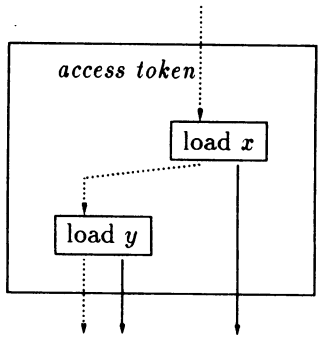


Figure 4: Detail of Schema 1 Read Block

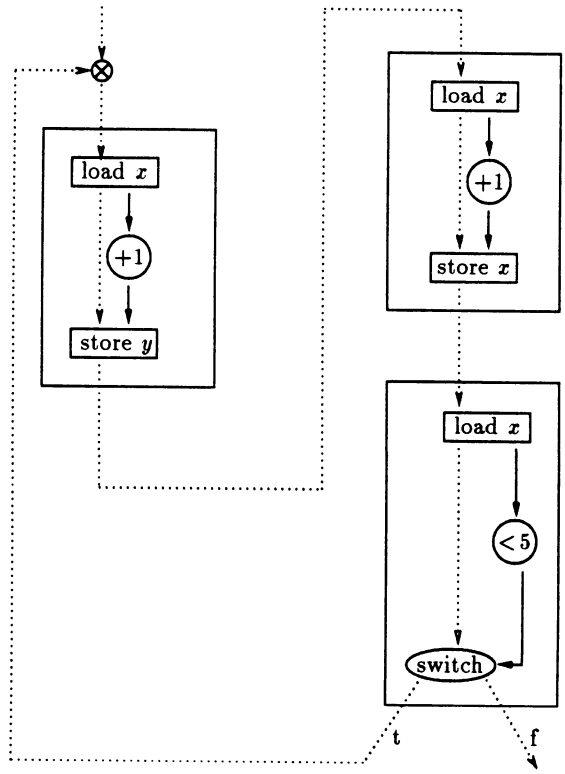
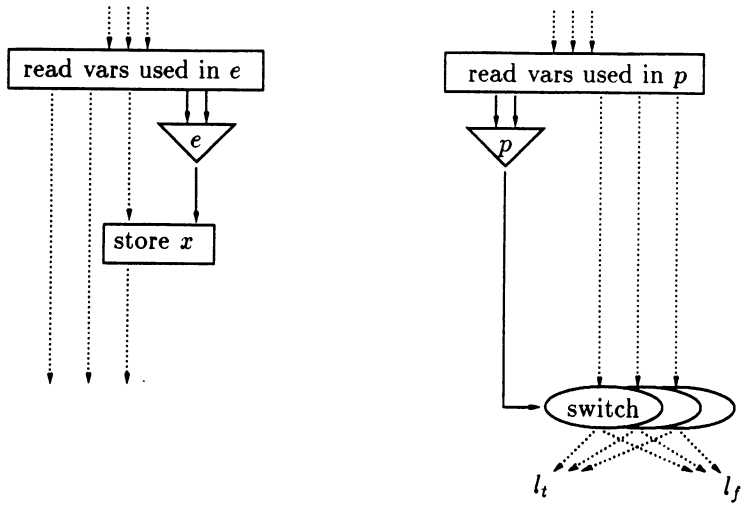


Figure 5: Example of Schema 1 Translation



$x := e$ **if** p **then goto** l_t **else goto** l_f

Figure 6: Schema 2 — Refining Access Control

control-flow graph.

Execution of the dataflow graph is achieved by circulating a token along the dotted arcs of the dataflow graph. This token plays a role similar to that of the program counter in executing a control-flow graph — it originates at *start*, visits statements in sequence, and visits every memory operation within a statement in sequence. Conditional branching in the control-flow graph is implemented through a *switch* operator that directs this token to one of two possible destinations. Notice that this token does not carry any value since it represents permission to access the stored state of the program variables. Therefore, we will call it the *access token.*

This schema correctly implements the sequential semantics of control-flow graphs. Expression parallelism is allowed within a single statement, but statements are executed one at a time. This ordering of operations is more restrictive than it needs to be. In the next section, we refine our schema to allow parallelism across the statements of a program.

3 Parallelizing Memory Operations

Since operations on different memory locations need not be sequentialized, we can enhance parallelism in the dataflow graph by circulating a *set* of access tokens, one for each variable. To simplify the presentation of this schema, we will assume initially that there is no aliasing — therefore, every variable denotes a unique memory location. We will denote the token that represents access to the variable named x by $\mathbf{access}_x$. Its arrival at a statement means that all previous memory operations on location x have completed. Therefore, a memory operation on x begins when it receives the token $\mathbf{access}_x$ at its input, and it propagates the token to its output when the operation is complete.

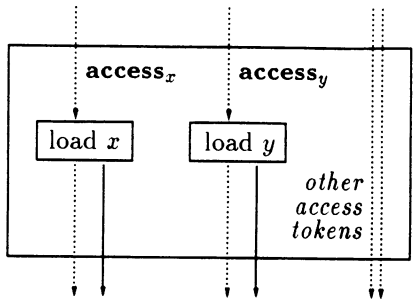


Figure 7: Detail of Schema 2 Read Block

In translation Schema 2, illustrated in Figures 6 and 7, the circulation of a single access token has been replaced by a set of access tokens corresponding to variable names. Tokens representing variables not used by an assignment statement flow directly to the next statement. By allowing independent memory operations to proceed in parallel, we are exploiting fine-grain parallelism across statements.

Readers familiar with the dataflow model of execution may find it odd that all communication of values between statements is through memory. The assignment schema begins by reading the values it will reference, and ends by storing a result. The ability of dataflow tokens to carry useful values is used only in the calculation of expressions and predicates. Access tokens that flow between nodes are dummy tokens which carry no useful value, but are used only for synchronization.

It would be possible to replace every $\mathbf{access}_x$ token in Schema 2 with a token carrying the value of x. This is true because we have disallowed array references and aliasing. We retain the memory-based description of Schema 2 in order to allow us to generalize it in Section 6 to account for these complications.

Unfortunately, naively translating each statement using Schema 2 does not work correctly if there is a cycle. Consider the dataflow graph corresponding to our running example, illustrated in Figure 8. For now, let us ignore the boxes labeled *loop entry* and *loop exit*. Operations on location x can proceed independently of operations on location y. When the load operator labeled $\mathcal{L}$ fires, it produces a token at the input of the increment operator labeled $\mathcal{I}$. The token $\mathbf{access}_x$ is passed on to the second statement where operations on x are allowed to proceed. When these operators complete, the token $\mathbf{access}_x$ can start on a new iteration of the loop, returning to the first statement, and the load operator labeled $\mathcal{L}$ can fire again.

It is easy to verify that the load operator labeled $\mathcal{L}$ can fire an unbounded number of times before the increment operator labeled $\mathcal{I}$ fires. In explicit token store machines like Monsoon, as in static dataflow architectures, each arc can hold at most one token. In tagged-token dataflow machines, tokens belonging to different iterations must have different tags, and we have not introduced any operators yet for generating such tags. Therefore, if we ignore the loop control boxes, the graph shown in Figure 8 does not specify a meaningful dataflow computation[4].

This problem arises whenever the control-flow graph has cycles in it. To identify cycles in

[4]There was no problem with cycles under Schema 1 since statements were not executed in parallel in that translation.

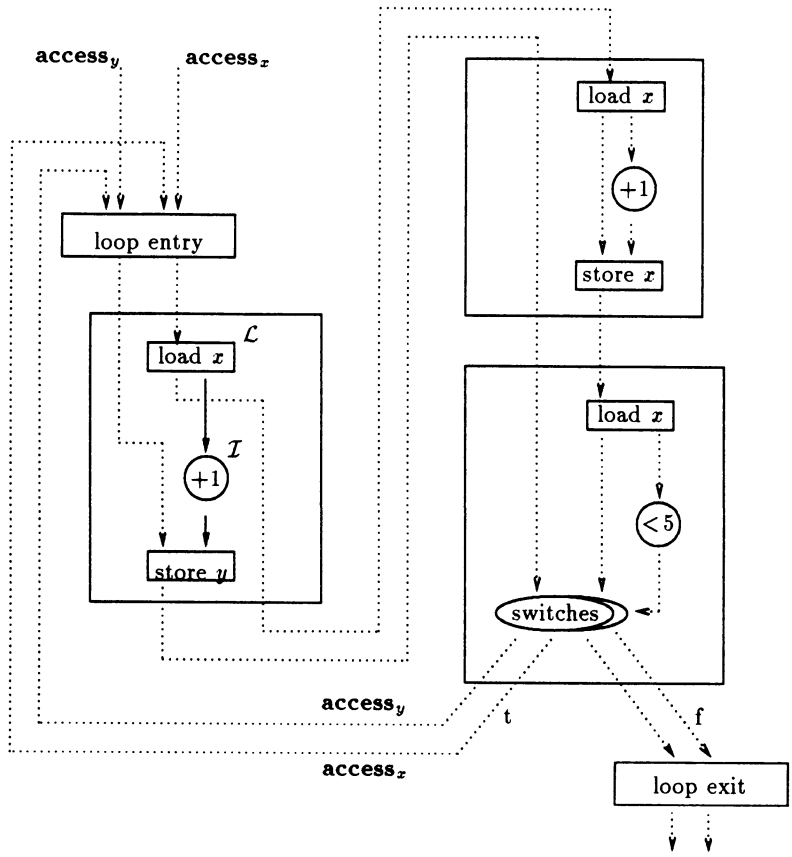


Figure 8: Example of Schema 2 Translation

general control-flow graphs that may arise from unstructured source programs, we perform an *interval decomposition* of the control-flow graph [1]. An interval is a generalization of a loop and is a maximal, single entry subgraph having a unique node called the header which is the only entry node and in which all cyclic paths contain the header. Notice that any subgraph that strictly contains an interval cannot itself be an interval. On the other hand, if the inner intervals are collapsed to single nodes (and self-loops are eliminated), the outer subgraph may become an interval in the new graph. It can be shown that most control-flow graphs arising from programs can be decomposed hierarchically into nested intervals this way — if we allow code copying, then any control-flow graph can be decomposed into such nested intervals[5].

To translate control-flow graphs into dataflow code, we first decompose the control-flow graph into nested intervals and introduce two new statements called *loop control* statements at the entry and exit of each cyclic interval. Arcs leading to the header from outside the interval are changed to lead to a single *loop entry* statement, which then leads to the header. All arcs from within the interval back to the header are changed to lead back to the *loop entry* node. A *loop exit* statement is placed on any edge that exits the cyclic part of the interval — that is, on any edge A $\rightarrow$ B in which there a path in the interval from A to the header, but not from B to the header. This transformed control-flow graph can then be translated as in Schema 2.

How should we translate the new statements into dataflow operators? This depends intimately on the model of dataflow we assume. On Monsoon, the *loop entry* statement could be translated into code that allocated a frame for the next iteration, while the *loop exit* operator could be translated into code that returned tokens from the last iteration of the loop. There are many other possible approaches to dataflow loop control, and we will not specify the implementation of the loop control statements any further. The interested reader is referred to the dataflow literature [2, 17]. In the rest of the paper, we leave the loop control operators as black boxes, and in translation Schema 2, we will simply require that each of them takes the complete set of access tokens as input and produces this set again as output. In Section 4, we will relax this requirement to allow some access tokens to bypass loops in which they are not needed, thereby increasing parallelism.

Notice that corresponding to every edge in the control-flow graph there is one edge in the dataflow graph for each variable in the program. Therefore, if E is the number of edges in the control-flow graph and V is the number of variables, then the size of the dataflow graph is $O(E \cdot V)$.

4 Optimizing the Dataflow Graph

Schema 2 exploits parallelism between independent memory operations by using access tokens that circulate independently. However, these access tokens still flow along the path of sequential execution — that is, they flow through every node that is executed even if they play no role in the execution of that node. This can result in access tokens flowing through switch operators needlessly, which introduces unnecessary order constraints. An example of this is shown in Figure 9. Figure 9(a) shows a control-flow graph in which x is not used within the if-then-else construct. Figure 9(b) is the corresponding dataflow graph produced by Schema 2. Clearly, the switch operator for $\mathbf{access}_x$ is unnecessary. Eliminating this switch and sending $\mathbf{access}_x$ directly from the statement $x := x+1$ to the statement $x := 0$ results in a more parallel program with no order imposed between the calculation of the predicate $w = 0$ and the execution of the second assignment to x. Notice

[5]For structured programs, this is the same as identifying all loops.

that in the optimized program, the $\mathbf{access}_x$ token does not flow along a control-flow path since it bypasses the conditional construct altogether.

This suggests that one way to optimize the dataflow graph produced by Schema 2 is to eliminate switches whose outputs are immediately merged together in the dataflow graph. The elimination of such redundant switches may make other switches redundant. As an example, consider a program in which one if-then-else construct is nested within another if-then-else, neither of which use x. Once the inner switch for $\mathbf{access}_x$ is recognized as redundant and removed, the outer switch for $\mathbf{access}_x$ becomes redundant. These newly redundant switches may be eliminated in turn. A generalization of this idea, which also allows access tokens to bypass loops in which they are not needed, leads to an iterative algorithm for redundant switch elimination and was discussed at length in an earlier version of this paper [6].

Rather than optimize iteratively the dataflow graph produced by Schema 2, we develop a direct construction that avoids introducing unnecessary switches. Notice that the problem is trivial if the source language has only structured conditionals and loops, since the program can be decomposed hierarchically, and syntactic analysis is sufficient to determine which access tokens are needed in each portion of the program.

With unstructured control-flow the problem is harder, but it can be viewed as a generalization of the structured case. Rather than look for uses of variables inside loops and conditionals, consider the portion of the control-flow graph between a node N and its *immediate postdominator*[6] P. Every control-flow path starting at N ultimately ends up at P. Suppose that there is no reference to a variable x in any node on any path between N and P. It is clear that an access token for x that enters N may bypass this region of the graph altogether and go directly to P.

This intuitive idea is formalized in Section 4.1. For each variable x, we identify fork nodes in the control-flow graph where switches must be introduced for $\mathbf{access}_x$. We show that switch placement can be efficiently computed using the postdominator tree of the control-flow graph. In Section 4.2, we present an algorithm which uses the switch placement information to construct directly a dataflow graph without any redundant switches.

4.1 Switch Placement

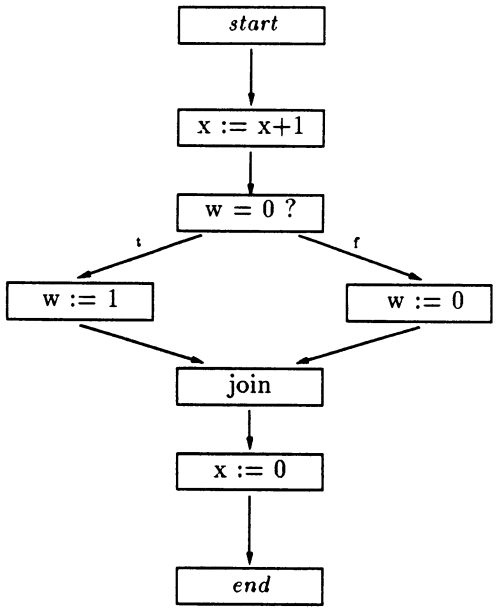
We first give a simple characterization of where switches are needed and then show how this information can be efficiently computed. This characterization is closely tied to the notion of control dependence [11].

The following definition captures the intuitive idea of a node being between another node and its immediate postdominator.

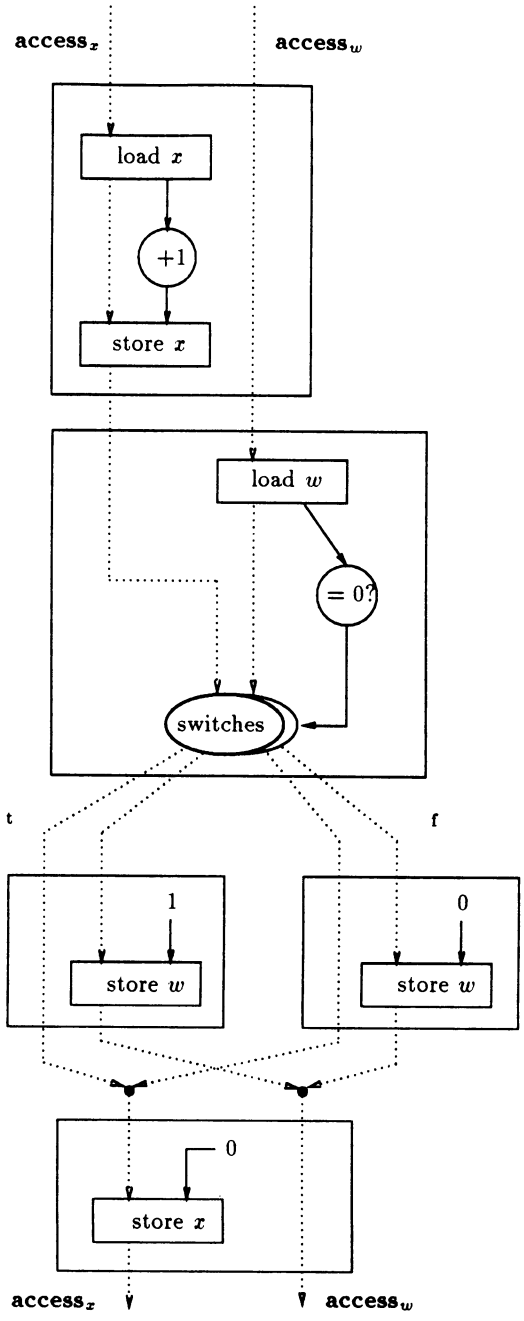
Definition 1 *If F is a node in the control-flow graph, we say that node N is* between F and its immediate postdominator P *iff there exists a non-null path $F \xrightarrow{+} N$ that does not pass through P.*

Since the path $F \xrightarrow{+} N$ can be extended to pass through P, it is natural to think of N as being between F and P. If N lies between F and P, our intuitive characterization says that we must

[6] A CFG node M *postdominates* node N iff every path from N to *end* passes through M [21]. Postdomination is reflexive; every node postdominates itself. If M postdominates N, and M and N are distinct nodes, M is said to be a *strict* postdominator of N. Every node has a unique *immediate postdominator* which is its closest strict postdominator on any path to *end*. The immediate postdominator relation is tree structured, and can be computed in $O(E)$ time [12].



(a) control-flow graph



(b) dataflow graph

Figure 9: An Example of Restrictive Sequential Ordering

introduce switches at F for all the access tokens needed by N. The following definition formalizes this intuitive description.

Definition 2 *Let F and N be nodes in the control-flow graph.* F needs a switch for N *iff N is between F and its immediate postdominator.*

We apply this general notion to define needing a switch for $\mathbf{access}_x$.

Definition 3 F needs a switch for $\mathbf{access}_x$ *if F needs a switch for some N that contains a reference to variable x.*

Having characterized where switches are needed for the access token of each variable, we show how this information may be computed efficiently. This is done by relating this problem to that of computing control dependences [11].

Definition 4 *A node N is* control dependent *on node F iff*

1. *there is a non-null path $p : F \xrightarrow{+} N$ such that N postdominates every node after F on the path p, and*
2. *N does not strictly postdominate the node F.*

Let $CD(N)$ denote the set of nodes on which N is control dependent; for S, a set of nodes, we let $CD(S) = \bigcup_{N \in S} CD(N)$.

As we will prove in Corollary 1, the set of forks that need a switch for N is exactly the *iterated control dependence set* of N. This set contains all nodes that N is control dependent on, together with all nodes that these nodes are themselves control dependent on, etc. More formally, we have the following definition.

Definition 5 *The* iterated control dependence set *of node N, denoted $CD^+(N)$, is the limit of the following sequence of sets:*

$$\begin{aligned} CD^1(N) &= CD(N) \\ CD^{i+1}(N) &= CD^i(N) \cup CD(CD^i(N)) \end{aligned}$$

Theorem 1 *If F and N are nodes in the control-flow graph, then N is between F and its immediate postdominator iff $F \in CD^+(N)$.*

Proof: Let P be the immediate postdominator of F.
($\Rightarrow$) Given that N is between F and P.

Let p be a path $F \xrightarrow{+} N$ that does not pass through P. Suppose that N postdominates all nodes on p between F and N after F. Notice that N cannot strictly postdominate F since P does not appear on path p. Therefore, N is control dependent on F. Otherwise, there is at least one node on path p other than F that is not postdominated by N. If F_1 is the last such node on path p_1, then clearly N is control dependent on F_1, so $F_1 \in CD(N)$. Now, consider the portion of path p between F and F_1. If F_1 postdominates all the nodes on p_1 between F and F_1 other than F, then $F \in CD(F_1)$, which implies that $F \in CD^+(N)$ and we are done. Otherwise, we can apply the

```
worklist ← ∅
for each variable x do
    for each node N do
        WL(N) ← false   /* flag set to true when N place on worklist */

    for each node N that references x do
        worklist ← worklist ∪ {N}
        WL(N) ← true

    while worklist not empty do
        remove N from worklist
        for all F on which N is control dependent do
            mark F as needing a switch for access_x
            if WL(F) = false then
                worklist ← worklist ∪ F
                WL(F) ← true
```

Figure 10: Algorithm for Switch Placement

construction again to the path between F and F_1 to find a node F_2 between F and F_1 such that $F_2 \in CD(F_1)$ etc. Since the length of the path under consideration decreases each time, it follows that $F \in CD^+(N)$.

($\Leftarrow$) Given $F \in CD^+(N)$. The proof is an induction on the definition of $CD^+(N)$.

Suppose that $F \in CD(N)$. Then N postdominates all nodes on a path p from F to N other than F, but does not strictly postdominate F. Therefore, P cannot occur on path p. Hence, there is path from F to N that does not pass through P.

Inductively, suppose that $F_1 \in CD^i(N)$ and $F \in CD(F_1)$. Let P_1 be the immediate postdominator of F_1. By the induction hypothesis, there is a path p_1 from F_1 to$N$ that does not pass through P_1. From the base case of the induction, there is a path p_2 from F to F_1 that does not pass through P. Now consider the path obtained by concatenating p_1 and p_2. We will show that P cannot occur on this path, thereby completing the inductive step. Now, P does occur on p_1. If P occurred on p_2, there is a path from F_1 to P that does not pass through P_1, a contradiction. Therefore, we have constructed a path $F \xrightarrow{+} N$ that does not pass through P. □

Corollary 1 *If F and N are nodes in the control-flow graph, then F needs a switch for N iff $F \in CD^+(N)$.*

Control dependences can be computed efficiently using a bottom-up walk of the postdominator tree [8]. The algorithm of Figure 10 uses this information to place switches.

Like switch placement, merge placement can also be computed directly from the control-flow graph. We will not discuss this algorithm since the algorithm in Section 4.2 for constructing the dataflow graph requires only that switch placement information be available, and it computes merge placement information as it wires up the graph.

4.2 Constructing the Dataflow Graph

Given a control-flow graph, we can now construct a dataflow graph having no redundant switches. At each node N we construct a *source vector* indexed by program variable, which we denote $SV_N(x)$. For each node N, $SV_N(x)$ specifies a set of pairs $\langle M, s\rangle$ where M is a dataflow node and s is an out-direction from M. Intuitively, $SV_N(x)$ represents the sources of $\mathbf{access}_x$ for node N. If pair $\langle M, s\rangle$ is in $SV_N(x)$, then $\mathbf{access}_x$ will flow from M to N along a dataflow arc corresponding to out-direction s (*true or false*). If N does not need $\mathbf{access}_x$, then $SV_N(x) = \{\}$. For example, if $\mathbf{access}_x$ flows from conditional F along the false out-direction directly to node N, then $SV_N(x) = \langle F, \mathit{false}\rangle$. If the source node has only a single out-direction then we simply use *true* as the out-direction.

If N is a switch which needs $\mathbf{access}_x$ or a statement which refers to x, then each set $SV_N(x)$ will have a single element. In the case where N is a join, and $SV_N(x)$ has a single element, no dataflow merge for $\mathbf{access}_x$ is needed at that point. If $SV_N(x)$ has more than one element, a dataflow merge is needed for $\mathbf{access}_x$ at N.

We construct a dataflow graph from a CFG as follows:

1. Insert *loop-entry* and *loop-exit* nodes into the CFG as discussed in Section 3.
2. Calculate where switches are needed for $\mathbf{access}_x$ using the algorithm in Figure 10.
3. Calculate SV_N using the algorithm in Figure 11. This calculation is analogous to building def-use chains when every reference to a variable x, every fork which needs a switch for $access_x$, and every join of $\mathbf{access}_x$ is treated as both a definition and a use of x. It differs from a simple data flow algorithm in two ways:
   (a) a non-local step is introduced to propagate information from a node which does not need a switch for $\mathbf{access}_x$ to its immediate postdominator, and
   (b) the basic chaining information is augmented with the out-direction of each arc.
4. Build the dataflow graph from the control-flow graph and source vector information by translating each node N according to Schema 2 and wiring its input arcs according to SV_N. Conditionals are translated by creating a switch for every $\mathbf{access}_x$ which is needed at N. Joins are translated by creating a dataflow merge for every $\mathbf{access}_x$ with more than one source. A join with a single source is equivalent to no operator.

The dataflow graph so constructed exhibits all of the data parallelism of Schema 2, and gains additional parallelism through the suppresion of redundant switches.

5 Aliasing, Parallelism, and Synchronization

The possibility of aliasing is significant because it means that a single memory location may be accessed by more than one name. The set of variables that may be aliased to a given variable x is called the *alias class* of x, and is denoted $[x]$. We formalize this notion by defining a structure that specifies the aliasing of variable names.

Definition 6 *If V is a set of variable names, an* alias structure *over V is a pair $\langle V, \sim\rangle$ where V is a set of variable names and the* alias relation *$\sim$ is a reflexive, symmetric binary relation on V.*

for all N **do**
    $SV_N(\star) \leftarrow \{\ \}$
    mark N as unvisited

worklist $\leftarrow \{start\}$
mark *start* as visited

while worklist not empty **do**
    remove N from worklist

    case typeof N
        • N is *start*:
            let S be the successor of N.
            for each program variable x **do**
                $SV_S(x) \leftarrow \{\ \langle N, true \rangle\ \}$

        • N is an assignment statement:
            let S be the successor of N.
            for each program variable x **do**
                if x referenced in N **then**
                    $SV_S(x) \leftarrow SV_S(x)\ \cup\ \{\ \langle N, true \rangle\ \}$
                else
                    $SV_S(x) \leftarrow SV_S(x)\ \cup\ SV_N(x)$

        • N is a fork:
            for each $\mathbf{access}_x$ needing a switch at N **do**
                for each successor S of N along out-direction d **do**
                    $SV_S(x) \leftarrow SV_S(x)\ \cup\ \{\ \langle N, d \rangle\ \}$

            for each $\mathbf{access}_z$ not needing a switch at N **do**
                $SV_P(z) \leftarrow SV_P(z)\ \cup\ \{\ SV_N(z)\ \}$
                    where P is the immediate postdominator of N

        • N is a join:
            let S be the successor of N.
            for each variable x **do**
                $SV_S(x) \leftarrow SV_S(x)\ \cup\ \{\ \langle N, true \rangle\ \}$

    for each unmarked successor S of N **do**
        if all predecessors (ignoring backedges) of S have been visited **then**
            worklist $\leftarrow$ worklist $\cup\ \{S\}$
            mark S as visited

Figure 11: Algorithm for Computing Source Vectors

The interpretation of the alias relation is that $x \sim y$ iff $x \in [y]$. Thus the alias relation determines the alias class of every variable in V.

Consider the following simple example. A FORTRAN subroutine is declared with three formal parameters `X`, `Y`, and `Z`:

```
SUBROUTINE F(X, Y, Z)
```

and is called in two places:

```
CALL F(A, B, A)
CALL F(C, D, D)
```

Since all parameter passing in FORTRAN is by reference, the formal parameters `X` and `Y` are each aliased to `Z`. However, since neither call identifies `X` and `Y` with the same location, they are not aliased to one another. Thus the alias structure for subroutine `F` is given by:

$[\texttt{X}] = \{\texttt{X}, \texttt{Z}\}$
$[\texttt{Y}] = \{\texttt{Y}, \texttt{Z}\}$
$[\texttt{Z}] = \{\texttt{X}, \texttt{Y}, \texttt{Z}\}$

Schema 2 can be generalized to account for aliasing as follows. Before a memory operation is performed on variable x, it is necessary to ensure that all memory operations on that location have completed. This condition can be ensured by requiring that all memory operations on any variable in the alias class of x have completed. We implement this requirement by specifying that access tokens for *all* variable names in the alias class of x must be collected before a memory operation on x is initiated.

In our example there would be three access tokens representing variables `X`, `Y`, and `Z`. Memory operations on `X` or `Y` would collect two access tokens, corresponding to the variable operated on and `Z`. Memory operations on `Z` would collect all three access tokens.

The drawback with this simple approach is that if there is a lot of aliasing then there will be considerable synchronization devoted to collecting access tokens. We will thus consider a refinement to this basic scheme which can reduce synchronization. Instead of having each access token denote a single variable name, we allow an access token to denote an arbitrary subset of the set of variables V.

Definition 7 *A* cover *C of an alias structure over V is a collection of subsets of V whose union is equal to V. A member of this collection is called a* cover element.

Schema 3 is parameterized by the choice of a cover C. Each access token $\mathbf{access}_c$ specifies a cover element $c \in C$. As is illustrated in Figures 12 and 13, a memory operation on variable x must collect all access tokens $\mathbf{access}_c$ such that $c \cap [a] \neq \{\}$. This collection of cover elements is the *access set* of x, denoted $C[x]$. We denote the set of access tokens corresponding to the access set of a variable x by $\mathbf{access}_{[x]}$, leaving C implicit. The entry and exit points of the dataflow graph are considered to be a use of every variable, and thus every access token must pass through them.

In choosing a cover for a given alias structure there are two concerns: maximizing parallelism and minimizing synchronization. It is possible to find a cover that maximizes parallelism and one that minimizes synchronization. However, in general there will be no one cover that achieves both. Choices of cover can provide a tradeoff between parallelism and synchronization, depending on the

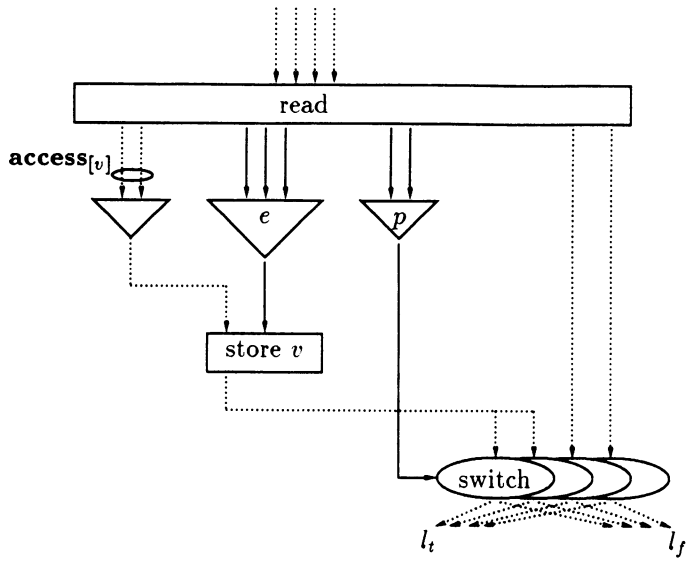


Figure 12: Schema 3 — Accounting for Aliasing

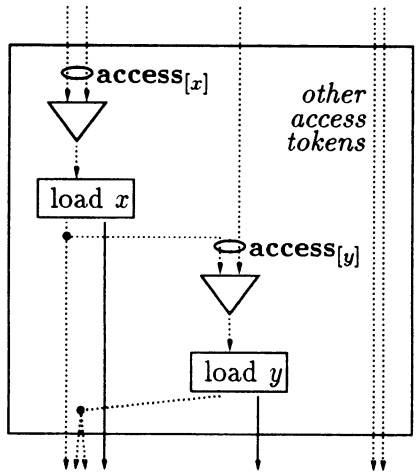


Figure 13: Detail of Schema 3 read block

particular flowgraph being considered. Since we do not as yet have a useful analysis of this tradeoff, we will not consider such choices here, and refer the reader to an associated technical report in which some of these issues are discussed [5].

6 Parallelizing Transformations

Up to this point, our concern has been the translation of programs into dataflow graphs without inserting unnecessary dependencies. In many programs, parallelism can be enhanced by transformations that remove dependencies. Many of these transformations make the program more "functional" in the sense that they are used to remove dependencies that arise because of multiple writes into a single memory location. In fact, in the absence of aliasing, memory operations on scalars can be eliminated completely and all values can be carried on tokens, as is usual in implementations of functional languages on dataflow machines. We also discuss briefly the use of dependence analysis techniques to improve the code generated by the schemas presented so far.

6.1 Elimination of Memory Operations

In our dataflow schema, all communication of values between statements is through memory. Every statement begins by reading the values it will reference, and ends by storing a result. The ability of dataflow tokens to carry useful values is used only in the calculation of expressions and predicates. Access tokens that flow across statements are dummy tokens which carry no useful value, but are used only to sequence memory operations. An important optimization is to eliminate memory operations wherever possible by passing *values* directly on tokens, rather than through memory locations. For variables that are not aliased, this is very easy. Load and store operations are deleted from the graph, and values are passed on tokens from definitions to uses.

If we restrict our attention to unaliased scalar variables only, this transformation has the effect of converting the program into a single assignment, functional program. It is similar in effect to classical transformations like renaming, live range splitting and conversion to static single assignment form. Why is this transformation so simple in our representation? Consider two definitions of a variable that both reach some use. Most conventional transformations will not rename the lefthand side variables of the two definitions to different variables since there is no easy way of "joining" these variables together at the use; the exception is static single assignment form which uses ϕ-functions for this purpose. In our representation, the joining of values to produce a single value is implicit in the model, which simplifies the transformation considerably.

6.2 Parallel Operations and Aliasing

Eliminating storage operations for potentially aliased variables is more difficult. However, some parallelism can be exploited even for these variables. If a store to a variable x is followed sequentially by a read from x, with no intervening stores to any variable that could be aliased to x, then the value stored can be passed directly to the output of the load.

Another important category of parallelization is parallelism between memory reads. Our access tokens enforce sequential access to memory, which is necessary only when writes are involved. Parallel access to memory can be allowed among any set of reads, even to potentially aliased variables.

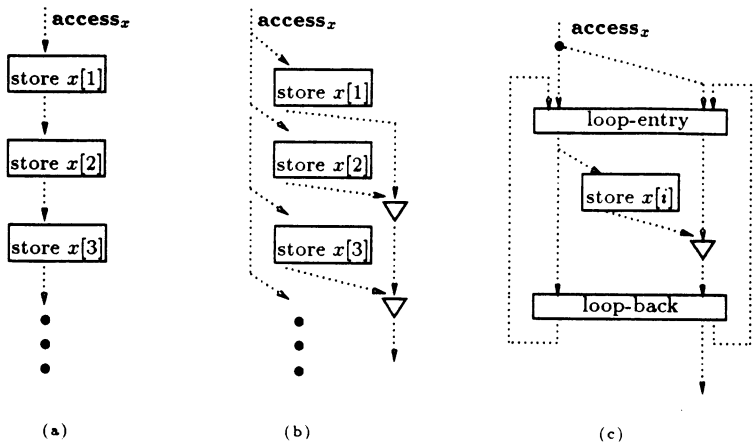


Figure 14: Parallelizing array operations

Consider a sequence of load operations, each of which receives the $\mathbf{access}_c$ from its predecessor and passes it directly to its successor. The predecessor of the first load can safely replicate $\mathbf{access}_c$ and pass it to every operation in the sequence. The replicas must be collected and passed to the successor of the last operation in the sequence. By parallelizing maximal sequences of load operations, read parallelism is maximized.

6.3 Parallelization of Array Operations

Arrays can be incorporated into the the translation schemas discussed thus far by treating an assignment to any array location as an assignment to the entire array — in a sense, arrays and scalar variables are treated identically. In many programs, however, standard disambiguation techniques such as subscript analysis [16] can be applied to programs to permit more parallelism in the resulting code.

Consider the following loop:

start: *join*
$i := i + 1;$
$x[i] := 1;$
if $i < 10$ **then goto** *start* **else goto** *end*
end:

It is clear that stores to successive elements of the array x are independent, and can be executed in parallel. However, analysis based on variable names would sequentialize them, since all require the access token for x.

One approach to parallelizing these operations, illustrated in Figure 14 is a generalization of the method for parallelizing reads. Part (a) shows the sequential stores to $x[i]$ in each iteration of the loop unfolded into a single thread of execution. Part (b) shows how the the access token for x can duplicated and passed to the next iteration. After storing to $x[i]$, the access token must

then be synchronized with the completion of the store in the next iteration. The duplication of the token ensures that there is no dependence between stores in successive iterations, and the synchronization ensures that the token is not generated at the end of the loop until all stores have completed. Finally, part (c) shows how this schema can be implemented in a dataflow loop.

A further enhancement of this transformation is to detect when an array is "write-once". If the dataflow machine has I-structure memory [3], array reads and writes can be done concurrently, since I-structure memory takes care of delaying premature read requests until the corresponding writes have occurred.

This is a simple example of how array dependence information can be used to enhance parallelism. A more complete discussion of this issue is under preparation and will be presented elsewhere.

7 Conclusions

In this paper, we have shown that imperative languages are not wedded to von Neumann architectures or to the von Neumann execution model by presenting a number of schemes for executing such programs on a dataflow machine. The final scheme can generate dataflow graphs from programs in an imperative language with unstructured flow of control, aliased variables and arrays, making use of sophisticated dependence information to enhance parallelism.

Veen and van den Born [22, 23] use a method similar to our Schema 2 to compile a restricted subset of the C programming language to a static dataflow architecture. They consider only structured programs with single-exit loops. For such programs, it is easy to build dataflow graphs without redundant switches, by constructing the graph starting with the innermost control structure. In addition, their dataflow model does not allow imperative updates to memory locations, which complicates the handling of aliased variables and arrays. Our approach does not suffer from these problems since our memory model permits imperative updates. We believe that the essence of the dataflow model lies in the local firing rules that determine when an operator should execute, not in the functional treatment of memory.

Ballance, Maccabe and Ottenstein [4] take a different approach to this problem. They start with the program dependence graph (PDG), in which control and data dependencies are separated out, and attempt to deduce information, such as switch and merge placement, that deals with the interaction of control and data dependencies. The translation schema described in their paper [4] is very complex, although it is unclear to us whether this is an inevitable consequence of starting from PDG's or whether it is an artifact of their particular approach. The simplicity of our approach arises from the fact that we base our translation on the control-flow graph, using dependency information to optimize this process.

The importance of our work is not limited to the implementation of imperative languages on dataflow architectures. Currently, compilers for imperative languages use abstract syntax trees, control flow graphs, and a combination of data dependence graphs and control dependence graphs to provide complete information about the execution semantics of programs and dependences between operations. On the other hand, many declarative language compilers use continuation passing style (CPS) as an intermediate form. Dataflow graphs synthesize these compiling technologies since the arcs in a dataflow graph can be seen both as encodings of dependence information as well as continuations in a parallel model of execution. Based on these ideas, we have developed the *dependence flow graph* (DFG), a program representation that has a *parallel, executable* semantics

and can also be viewed as a data structure incorporating data dependence information. The Typhoon project at Cornell is developing a compiler based on this representation to show its usefulness for conventional optimizations and for parallelization.

Acknowledgments

We would like to thank Richard Huff, Wei Li, and Anne Rogers for their helpful comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Arvind and K. Gostelow. The U-interpreter. *Computer*, 15(2), February 1982.

[3] Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11, October 1989.

[4] 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. *Proceedings of the 1990 SIGPLAN Conference on Programming Language Design and Implementation*, 25(6):257–271, June 1990.

[5] M. Beck and K. Pingali. From control flow to dataflow. Technical Report TR89-1050, Cornell University, October 1989.

[6] Micah Beck and Keshav Pingali. From control flow to dataflow. In *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[7] R. Cartwright and M. Felleisen. The semantics of program dependence. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 25(6), June 1989.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 25–35, January 1989.

[9] J. B. Dennis. First version of a data flow procedure language. In *Proceedings of the Colloque sur la Programmation, Vol. 19, Lecture Notes in Computer Science*, pages 362–376, 1974.

[10] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, November 1980.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependency graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, June 1987.

[12] D. Harel. A linear time algorithm for finding dominators in flowgraphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 185–194, May 1985.

[13] R. Keller. Rediflow multiprocessing. In *Proceedings of Compcon 84*, 1984.

[14] D. Kuck, D. Lawrie, R. Cytron, A. Sameh, and D. Gajski. The architecture and programming of the cedar system. Technical report, University of Illinois, Urbana-Champaign, 1983.

[15] R. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical Report CSG Memo 265, M.I.T. Laboratory for Computer Science, 1986.

[16] D. Padua and M. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, pages 1184–1201, December 1986.

[17] G. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Massachusetts Institute of Technology, 1988.

[18] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence Flow Graphs: An algebraic approach to program dependencies. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, January 1991.

[19] C. Polychronopoulos. The Parafrase-2 restructurer. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[20] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1986.

[21] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal of Computing*, 3(1):62–89, 1974.

[22] A. H. Veen. The misconstrued semicolon: Reconciling imperative languages and dataflow machines. Technical Report CWI Tract 26, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1986.

[23] A. H. Veen and R. van den Born. The RC Compiler for the DTN Dataflow Computer. *Journal of Parallel and Distributed Computing*, 10:319–332, 1990.

List of Figures