From Control Flow To Dataflow

Micah Beck
Keshav Pingali
Department of Computer Science
Cornell University
Ithaca, NY 14853

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 dependencies, control dependencies, 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 Introduction

Religious wars between the declarative and imperative schools of parallel computation are fought on the battlegrounds of both language and architecture. Disciples of the declarative approach program in functional or logic languages and await the coming of dataflow or reduction machines on which their programs can run efficiently. Believers in the imperative approach keep connecting von Neumann processors in different ways and have faith that parallelizing compilers will let them program their machines in fortran. 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 in some natu-

*This research was supported by an NSF Presidential Young Investigator award (NSF grant #CCR–8958543), and grants from the General Electric Corp. the Math Sciences Institute of Cornell, and the Digital Equipment Corporation

†The authors can be reached at electronic mail addresses beck@cs.cornell.edu and pingali@cs.cornell.edu.

ral way on dataflow architectures?

At first sight, dataflow machines appear ill-suited to executing imperative language programs. Dataflow machines have no program counter to sequence operations; rather, instructions are scheduled dynamically for execution whenever they receive input data. All non-memory operations are purely functional in effect since a dataflow instruction executes by consuming data tokens at its inputs and producing a data token at its output. These notions are in stark contrast to the traditional operational semantics of imperative language programs, defined in terms of side-effects. Each statement in an imperative language program is a command whose execution causes a change in storage. 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 gotos's that modify the program counter. Thus, there would appear to be a vast gulf between the dataflow model of program execution and the operational semantics of imperative languages.

Given these differences, is it possible to execute imperative language programs on dataflow machines in some natural way? Some researchers have proposed to achieve this by extending dataflow graphs with imperative operators, and executing the entire graph sequentially using a "thread descriptor" to simulate a program counter. In our opinion, this is really a simulation of von Neumann instruction sequencing on a dataflow machine, which is one (bad) way of implementing imperative languages. Such a simulation overly restricts parallelism. In this paper, we exhibit a parallelizing translation of imperative language programs into dataflow graphs which can be executed on a dataflow machine like Monsoon [7]. Instruction level parallelism in imperative programming languages is very easily exploited using our approach. More importantly, we establish a close connection between our work and current compiler efforts in the imperative languages community. We establish ties with work on data dependencies [8], control dependencies, program dependence graphs [5, 3], and static single assignment form [4]. This result suggests that dataflow graphs can serve as an executable intermediate representation for imperative language programs.

The rest of the paper is organized as follows. In Section 2.1, we describe a simple imperative flowgraph language. In Section 2.2, we describe our dataflow model, and in Section 2.3 we show a simple translation of flowgraph 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 removing redundant control operations;

this refinement is related to the notions of control dependencies 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 aliasing into account. In Section 5, we describe a framework for exploiting parallelism in the presence of aliasing. Section 6 discusses standard parallelization methods including exploitation of array parallelism in loops. Our schemes for handling aliasing and loops demonstrate that we can exploit existing compiler techniques such as dependency analysis and loop detection. We summarize our results in Section 7.

2 A Framework for Translation

In this section, we present a simple imperative flowgraph language. We will initially ignore aliasing and data structures, but will consider these issues in Sections 5 and 6. We also present 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. [7]. Finally, we present a simple translation from flowgraph programs to dataflow program graphs.

2.1 A Simple Imperative Language

A program in our imperative flowgraph language consists of a set of labeled assignments connected by conditional or unconditional branches. The syntax of expressions and predicates is left unspecified, and we have ignored all issues of variable type, scope, and aliasing. We will however assume that each expression or predicate is a functional combination of the values of a fixed set of variables. In Section 5, we generalize our translation to handle aliasing, enabling programs in a language like fortran to be easily translated into our flowgraph language.

A flowgraph is a set of statements of the form

$$l:\ v := e \parallel \textbf{ if } p \textbf{ then } l_t \textbf{ else } l_f$$

where l is a unique label for that statement, v is a variable name, e is an expression, p is a predicate, and l_t and l_f are statement labels. Unconditional control transfer is implemented by using the constant predicate true. Execution begins at the statement labeled start and completes upon branching to the reserved label end.

The operational semantics of statement execution are as follows:

1. expression e and predicate p are evaluated.

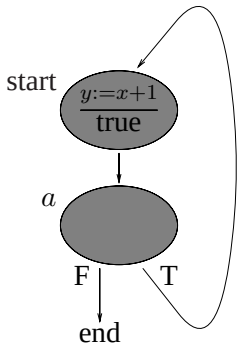


Figure 1: An example flowgraph

2. the variable v is assigned the value of e.[‡]

3. control then passes to statement l_t if p is true, and to statement l_f otherwise.

We will use the following simple flowgraph as a running example:

start*:** $y := x + 1 \parallel$ **if *true* then *a
a*:** $x := x + 1 \parallel$ **if** $x < 5$ **then *start* else *end

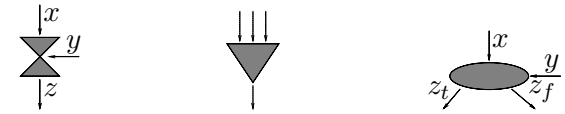
Figure 1 illustrates this flowgraph.

2.2 Dataflow Model

We will use a conventional explicit token store dataflow machine, such as MIT's Monsoon [7] 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 a two-input operator will rendezvous at a fixed location in some frame.

One important aspect of our model is the behavior of memory. Data structures in earlier models of dataflow were implemented in I-structure memory in which locations could be allocated under program control and could be written into at most once. A garbage collector was used to recycle I-structure locations no longer in use by the program. This memory also supported synchronization between reads and writes at the memory — a read request that arrived at a location before the corresponding write was held by the memory controller until the write occurred. In our model, as in Monsoon, 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 block-

[‡]Note that p is evaluated before the assignment to v. An occurrence of v in p refers to the old value of v.



Throttle Synch Tree Switch

Figure 2: Key To Dataflow Schema Symbols

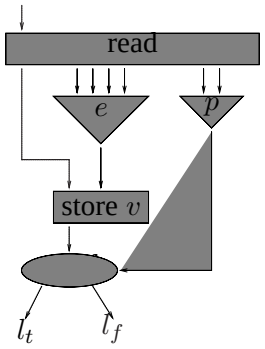


Figure 3: Schema 1 — Implementing Sequential Semantics

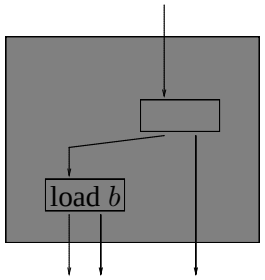


Figure 4: Detail of Schema 1 read block

ing the processor pipeline while a memory operation is underway.

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.

Some important operations are indicated by special symbols, as shown in Figure 2. The throttle is a two-input identity operation used for synchronization. It has two inputs x and y; when both are present the throttle copies x to the output z; the input y is treated as a signal — its value is ignored. Throttles can be combined into synchronization trees, which generate a single output token when all of their inputs are present. Since a synchronization tree is a form of expression, it is indicated by an unlabeled triangle. A switch is a special operation that has two inputs x and y, and two possible outputs z_t and z_f. When both x and y are present, x is copied to either z_t or z_f according to the boolean value carried by y.

2.3 A Simple Translation

We will first consider implementing the fully sequential semantics of flowgraph execution directly in dataflow graphs. The

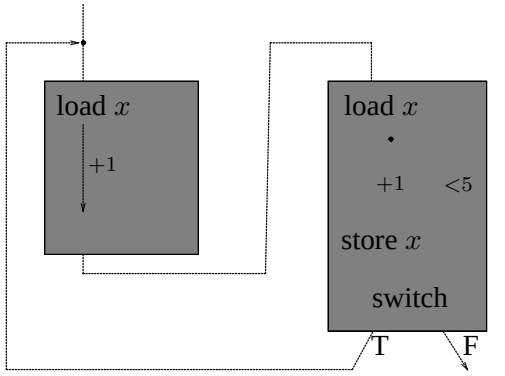


Figure 5: Example flowgraph schema 1

translation of a statement is illustrated in Figure 3, with a detailed view of the read block shown in Figure 4. The translation schema for a statement consists of graphs to evaluate e and p, a store operation to assign the value to v, and a switch.

Sequencing of statements is achieved by circulating a token that represents access to the stored state of the program variables. This access token visits every memory operation in a statement in sequence, and visits statements in sequence. Because the access token carries no useful value, it is considered to be a signal and is indicated by dotted lines. Conditional sequencing is implemented by a token switch that directs the access token to one of two possible destinations. For unconditional sequencing the switch is omitted. Figure 5 illustrates the translation for our example flowgraph.

This schema correctly implements the sequential semantics of flowgraphs. Expression parallelism is allowed within a single statement, but all memory operations are sequentialized. The ordering of operations enforced by our translation schema is even more restrictive than is specified by the operational semantics given in Section 2.1.

In the next section we refine our schema to allow parallelism across the statements of a program.

3 Parallelizing Memory Operations

The translation of flowgraphs to dataflow program graphs given in Section 2.3 uses a single token to represent access to any part of the stored program state. Since every statement accesses memory, this token enforces sequential execution of statements. While this schema does ensure a correct sequence of memory accesses, it restricts parallelism unnecessarily.

Our first refinement of this translation implements access control with a set of access tokens, one for each program variable name. In order to simplify the presentation of this schema, we will assume initially that there is no aliasing of variables. We will denote the token that represents access to the variable named x by **access**$_x$. Its arrival at a statement means that all previous memory operations on the variable name x

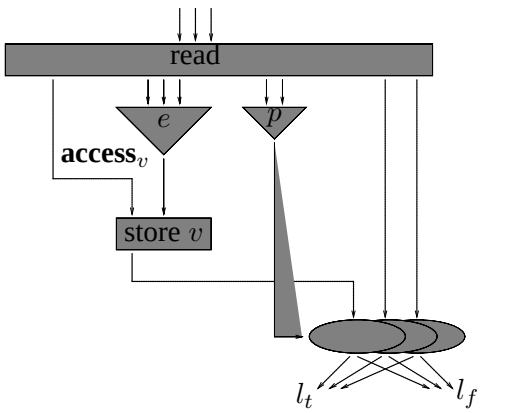


Figure 6: Schema 2 — Refining Access Control

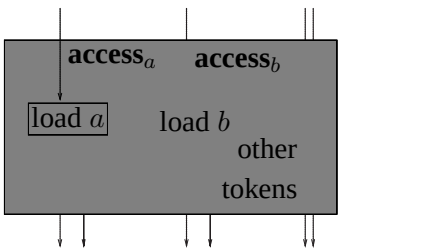


Figure 7: Detail of Schema 2 read block

have completed.

Because we are not considering aliasing of variables, every variable name denotes a unique memory location. To achieve the sequential semantics of flowgraphs, we will preserve the order in which load and store operations are applied to each memory location. This is implemented by synchronizing each memory operation of a variable a with the arrival of the token **access**$_a$. The **access**$_a$ token is propagated when the memory operation is complete.

Statement Schema 2, illustrated in Figures 6 and 7, shows that 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 the statement flow directly to the switch and on to the next statement. By allowing independent memory operations to proceed in parallel, we are exploiting fine-grain parallelism across statements.

3.1 Cyclic Flow Graphs

Unfortunately our translation does not work correctly if there is a cycle in the flowgraph. Consider the dataflow graph corresponding to our running example, illustrated in Figure 8. For now, let us ignore the boxes labeled loop entry, loop exit, and loop-back. Since x and y denote different memory locations, 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 **access**$_x$ token is passed on to the second statement where operations on x are allowed to proceed. When these operators complete, the **access**$_x$ token 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

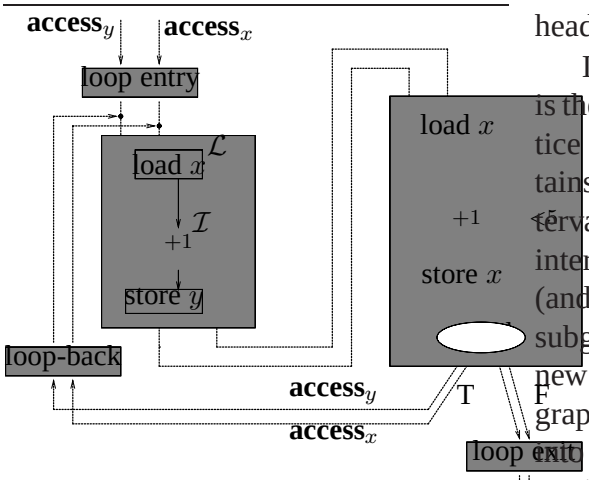


Figure 8: Example flowgraph schema 2

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. Therefore, the graph shown in Figure 8 does not specify a meaningful computation.

The problem with circular dataflow graphs is not a consequence of the unstructured nature of our flowgraph language — the same problem arises when compiling the structured looping construct of a functional language like Id [6]. We can generalize the solutions used in translating Id programs to dataflow graphs by considering intervals rather than loops [1]. An interval is a maximal, single entry subgraph which has a unique node called the header which is the only entry node and in which all cyclic paths contain the header.

If loops generalize to intervals, what is the generalization of nested loops? 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 any graph can be decomposed hierarchically into "nested intervals" this way — every "nested interval" is either an interval in its own right, or it becomes an interval if inner intervals are collapsed into single nodes and self-loops are eliminated [1].

To execute programs with cycles correctly, we decompose the flowgraph into nested intervals and introduce three new statements called loop control statements. Arcs leading to the header from outside the interval are changed to lead to a single loop entry statment, which then leads to the header. A loop exit statement is placed before any node which:

1. is not in any cycle, and

2. all of whose immediate predecessors are in some cycle.

Arcs leading to such a node are changed to lead to a loop exit statment. All arcs from within the interval back to the header are changed to lead instead to a single loop-back statement, which then leads to the header.

This flowgraph can then be translated as in Schema 2. How should we translate the new statements into dataflow graphs? Since there are many possible approaches to dataflow loop control, we will not specify the implementation of the loop control statements. We will instead introduce three "black box" subgraphs corresponding to these statements and in our translation 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 increase parallelism.

A full discuss of the implementation of interval control subgraphs and a full must take several factors, including details of the specific model of dataflow execution, into account [2]. No single solution exists; instead there is a tradeoff of parallelism against resources which admits many possible solutions. Because of space limitations, we offer the following summary.

In Monsoon, each loop iteration is given an activation frame for holding its tokens. The loop entry operator arranges for the allocation of such a frame and starts the execution of the first iteration. The loop-back operator handles frame allocation for subsequent iterations. If a new frame is required, it arranges for its allocation; otherwise, a frame allocated for a previous iteration is reused, provided that that iteration has terminated. The loop exit operator sends the outputs of the loop back to the parent context. Thus, these operators are concerned with context manipulation.

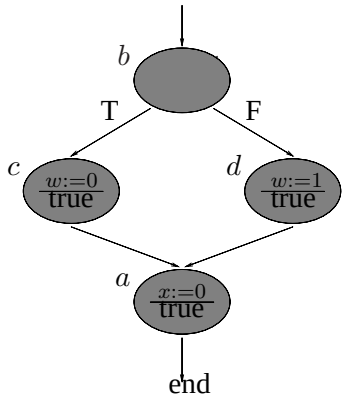


Figure 9: An example of restrictive sequential ordering

4 Optimizing Control Flow

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 statement that is executed, even if they play no role in that statement. This can result in access tokens flowing through switch operators unnecessarily, which introduces unnecessary dependencies. An example of this is shown in Figure 9, and the dataflow graph for this program in shown in Figure 10.

In this example, the token **access**$_x$ will flow from statement b to statement

a regardless of the value of variable w. However Schema 2 enforces sequentiality between the calculation of the predicate in statement b and the execution of statement a. If **access**$_x$ were passed directly to statement a, then no such dependence would be introduced, and parallelism would be enhanced.

In representations such as the program dependence graph (PDG), this situation can be recognized by the fact that there is no control dependence between the predicate in statement b and statement a [5, 3]. That is, statement a is executed regardless of the value of the predicate in statement b. However, this does not necessarily mean that the access token for x can be passed directly from statement b to statement a. For example if statement c contained an assignment to x, then there would be a data dependence from that statement to a, and the access token for x would have to flow through c. In PDGs, control and data dependencies are represented by separate sets of arcs. Interactions between dependency types introduce complications into the use of PDGs [3].

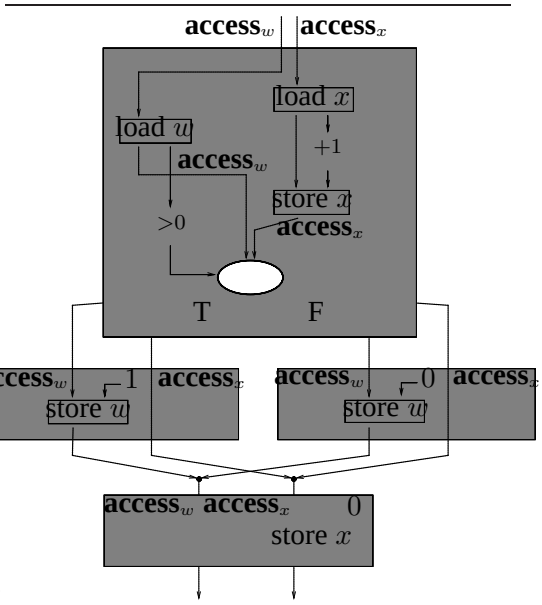


Figure 10: Dataflow graph with redundant switch

In contrast, our use of dataflow switches to represent the effect of control flow on data dependencies makes the identification of unnecessary switches easy. Consider the switch operator for **access**$_x$ in Figure 10, which corresponds to the conditional branch in statement b of our example. Both arms of this switch go to statement a and there are no uses of x on either of these arms. It is clear that this

switch operator is redundant and can be deleted from the graph if **access**$_x$ is rerouted to flow directly from the store operator in statement b to the store operator in statement a. In this example there is only one redundant switch, but in general the removal of one switch may cause other switches to become eligible for deletion. Section 4.2 shows how redundant switches can be removed from the dataflow graph.

4.1 Detecting Redundant Switches

We now formalize the notion of a switch in a dataflow graph being redundant, and indicate an efficient global algorithm for determining which switches are redundant. We will then show how the flow of access tokens in the dataflow graph can be optimized on the basis of this information.

Definition 1 A use of variable x in a dataflow graph is any memory operation on variable x.

Definition 2 A node P postdominates a node S in a dataflow graph iff all paths from S to exit pass through P. If P is a postdominator of S, and every postdominator of S postdominates P, then P is the immediate postdominator of S.

Definition 3 Let P be the immediate postdominator of a switch S in a dataflow graph. Switch S is redundant iff there is no use of x on any path from S to P, other than P itself.

Theorem 1 establishes an important connection between the problem of finding redundant switches in a dataflow graph and the notion of static single assignment form [4]. This leads us to an efficient algorithm for calculating the set of redundant switches.

Theorem 1 A switch S for an access token **access**$_x$ in the dataflow graph is redundant iff it is not possible to find two paths $S \xrightarrow{+} S_1$ and $S \xrightarrow{+} S_2$ such that

1. S_1 and S_2 are uses of x
2. the two paths are disjoint except for S.

The proof of Theorem 1, which appears in [2] has been omitted for reasons of space.

Standard compiler techniques such as def-use chaining can be used to calculate the set of redundant switches. A faster algorithm can be obtained by reducing the problem to that of computing static single assignment (SSA) form [4]. Computing SSA form requires the introduction of so-called ϕ-functions for each variable x in the program. A ϕ-function for variable x is introduced at a node S in the graph if there exist nodes S_1 and S_2 such that there are non-null paths $S_1 \xrightarrow{+} S$ and $S_2 \xrightarrow{+} S$ in which S is the only common node and S_1 and S_2 both contain assignments to x, or themselves need ϕ-functions for x. In other words, we can determine which switches

for a variable x are redundant by reversing the direction of arcs in the graph, treating every use of x as a definition and solving the problem of finding where to introduce ϕ-functions for variable x in this graph. An $O(E \times V)$ algorithm for this problem has been devised by Cytron et al [4].

The correspondence between optimal placement of switches and reverse SSA form is not coincidental. The arcs along which access tokens flow in the dataflow graph can be viewed as a representation of def-use chains in which this information has been knitted in with control flow information (the switches implement forking of these chains due to conditional branching). By way of analogy, we can view the SSA form as a representation of use-def chains in which the ϕ-functions implement joining of these chains due to the confluence of control flow paths. Thus while different graphs are considered and the dependencies are reversed, the structure is the same.

4.2 Eliminating Redundant Switches

Having calculated which switches are needed in a dataflow graph, we can now eliminate those that are redundant. The postdominator of each redundant switch can easily be calculated as part of the algorithm for determining which switches are needed. Any redundant switch for a variable x can then be eliminated by simply rerouting its input(s) to its immediate postdomintator. By definition there is no use of x between the switch and the postdominator.

This said, some attention must be paid to the order in which redundant switches are eliminated. A redundant switch S_1 may itself be an immediate postdominator of another redundant switch S_2. If switch S_1 is eliminated before switch S_2, then the postdominator of S_2 must be recalculated. We want to eliminate the switches in an order which avoids recalculating immediate postdominators in such cases.

To find such an order, we use the fact that the immediate postdominator relation organizes the nodes of the dataflow graph into a tree. In this tree the leaves are nodes which postdominate nothing, and the root is exit which postdominates everything. If we order the nodes in the graph by a postorder traversal of this tree, then the restriction of this order to redundant switches has the desired property. A switch will always be eliminated before its immediate postdominator.

This optimization has the property that an access token which is not used in the cyclic portion of an interval will be eliminated from the cycles, and will not be made to flow through the loop-back subgraphs. However, such a token will still flow needlessly through the loop entry and exit subgraphs. This case can easily be detected, and such tokens can be removed from the loop control subgraphs altogether.

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. Thus in Schema 2 it will not suffice to synchronize a memory operation on variable x with the arrival of **access**$_x$.

We call the set of variables which might be aliased to variable x the alias class of x. 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 y 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. The completion signal from the memory operation would similarly be split into a separate signal to propagate each **access**$_y$.

This simple approach has one drawback: it will perform unnecessary synchronization in many cases. Consider two variables x and y which are equivalent; they always represent the same location. Assume further that neither x nor y is aliased to any other variable. Clearly, we can represent x and y by a single access token with no loss of parallelism. The cost of synchronizing **access**$_x$ and **access**$_y$ will then be eliminated.

Equivalent variables are an extreme case, and are easily accounted for. However, when alias classes are not transitive, the assignment of sets of variables to access tokens can reduce parallelism. A formal framework can be developed for analyzing the decomposition of the set of all variables into a collection of sets of variables called a cover [2]. It is possible to prove that, given particular aliasing information, two natural covers can be defined: one which maximizes parallelism, and one which minimizes synchronization. In general, there will not exist a single cover which achieves both.

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 anti-dependencies (that is, 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 some coarse-grain parallelization transformations similar to vectorization.

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 values is used only in the calculation of expressions and predicates. Access tokens that flow across statements are dummy tokens which carry no value, but are used only for synchronization. An important optimization is to eliminate memory locations wherever possible by passing values directly on tokens, rather than via access tokens. For variables that are not aliased, this is very easy. Recall that a store operation takes an access token and a value as input, performs the store and passes on the address at its output. To delete the store, we simply remove the store operator and the access token line from the graph, rerouting the value line to the output. The load operator can be deleted by replacing it with a fork operation that receives a value token at its input and duplicates it to both of its outputs. Some of these outputs may be removed from the graph if the access token from the load operator went to a store operator in the original graph.

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 of the 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 inter-

vening 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.

Consider a sequence of load operations, each of which receives the **access**$_c$ from its predecessor and passes it directly to its successor. The predecessor of the first load can safely replicate **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.

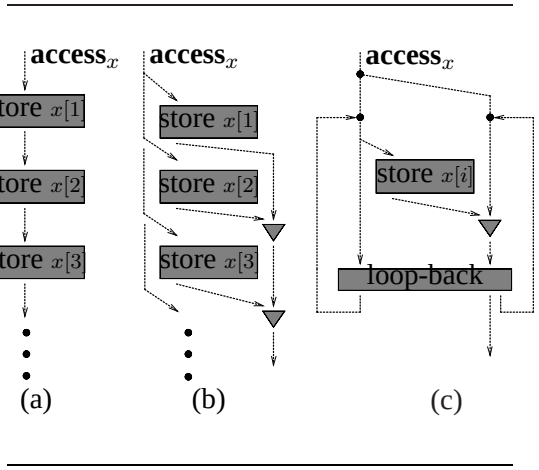


Figure 11: Vectorizing loop operations

6.3 Loop Vectorization

In Schema 3, we have shown how an analysis of aliasing can be applied to the parallelization of independent memory operations. The above section addresses the parallelization of scalar reads. A more subtle issue is the parallelization of independent memory operations in loops. Our simple method of determining independence on the basis of variable names ignores the fact that many accesses to a single array may be independent.

Consider the following looping flowgraph:

***start*:**	$i := i + 1$	$\parallel$	**if *true* then *a***
***a*:**	$x[i] := 1$	$\parallel$	**if** $i < 10$ **then *start* else *end***

It is clear that stores to successive elements of the array x are independent,

and can be executed in parallel. However our analysis based on variable names would sequentialize them, since all require the access token for x.

Our approach to parallelizing these operations, illustrated in Figure 11 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.

This schema for vectorizing loops is a departure from our uniform treatment of arbitrary control flow. Loops are a special case because their regularity can allow an analysis of subscripts to determine the independence of successive iterations. Because our simple flowgraph language does not include looping constructs, such loops must be discovered. Parallel versions of fortran not only specify loops, but also allow parallelism to be declared by the user in a doall construct. A further enhancement of this transformation is to detect when an array is "write-once". In that case, array reads and writes can be done concurrently, since the I-structure memory takes care of delaying premature read requests until the corresponding writes have occurred.

7 Conclusions

We have shown in this paper that imperative languages are not wedded to von Neumann architectures nor to the von Neumann execution model. The constraints imposed by the possibility of multiple writes to a single memory location define a partial order on the execution of operations. We can implement this partial order on a dataflow machine by circulating a set of access tokens. Starting with a fairly sequential schema, we introduced parallelism on the basis of independent memory operations. We then further optimized the sequential control flow of the original program to eliminate synchronization at decision points not corresponding to control dependencies. Finally, we considered a number of parallelizing transformations and showed that they could be implemented easily in our framework.

We do not see the importance of this work as being limited to the development of parallelizing compilers for imperative languages running on dataflow architectures. The arcs that connect operations in a dataflow program graph can be viewed as a representation of depen-

dencies. By abstracting away the details of context manipulation instructions in dataflow graphs, we get a parallel, executable representation of imperative programs that incorporates all the dependencies between operations. We believe that this representation is good for parallelizing compilers, even if the target architecture is not dataflow.

Currently, parallelizing compilers for imperative languages use a combination of abstract syntax trees control flow graphs, data dependence graphs and control dependence graphs (linked together in some way) to provide complete information about the execution semantics of programs and dependencies between operations. On the other hand, many declarative language compilers use continuation passing style (CPS) as an intermediate form [9]. Dataflow graphs synthesize these two representations of dependencies since the arcs in a dataflow graph can be viewed both as encodings of dependency information as well as continuations in a parallel model of execution. We believe that some kind of dataflow representation is better suited for this purpose, and that the real impact of dataflow ideas may well be at the level of compiling, rather than at the level of machine architecture. We leave that subject for another paper.

Acknowledgments

We would like to thank Richard Huff, Richard Johnson, and Anne Rogers for their helpful comments. The proof of Theorem 1 was contributed by Richard Johnson and Wei Li. Discussions with Karl Ottenstein helped us to understand Program Dependence Graphs. We thank Bob Rau the the Advanced Architecture group at Hewlett Packard for useful discussions on the subject of compiler intermediate forms.

References

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

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

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] G. Steele. RABBIT: A compiler for SCHEME. Technical Report AI memo 474, M.I.T. Laboratory for Artificial Intelligence, May 1978.