An Executable Representation of Distance and Direction

Richard Johnson
Wei Li
Keshav Pingali

Cornell University

Abstract

The dependence flow graph is a novel intermediate representation for optimizing and parallelizing compilers that can be viewed as an executable representation of program dependences. The execution model, called dependence-driven execution, is a generalization of the tagged-token dataflow model that permits imperative updates to memory. The dependence flow graph subsumes other representations such as continuation-passing style [12], data dependence graphs [13], and static single assignment form [8]. In this paper, we show how dependence distance and direction information can be represented in this model using *dependence operators*. From a functional perspective, these operators can be viewed as functions on streams [4].

1 Introduction

The growing complexity of optimizing and parallelizing compilers has led the compiler community to re-examine the design of intermediate program representations. Traditionally, compilers have used the control-flow graph augmented with dependence information such as def-use chains [1], data dependences [13] and control dependences [10]. The control-flow graph represents the execution semantics of the program (how the program can be executed), while dependences are viewed as precedence constraints between statements that

[1]This research was supported by an NSF Presidential Young Investigator award (NSF grant #CCR–8958543), NSF grant CCR-9008526, and grants from IBM and HP.

must be maintained for correct execution.

The separation of execution semantics from dependence information results in a number of problems.

When a program is transformed, dependence information may need to be modified, but it is usually difficult to do this incrementally; for example, it is hard to update def-use chains after eliminating unreachable code [15]. Therefore, the full benefit of program optimization must be obtained in one of two ways. One way is to perform repeated passes of program analysis and transformation. Alternatively, the problem can be circumvented through the use of complex algorithms such as the global constant propagation algorithm of Wegman and Zadeck, which combines constant propagation with unreachable code elimination. This algorithm requires simultaneous traversals of both the control-flow graph and def-use chains [17]. Neither approach is satisfactory; the first is expensive and the second does not solve the problem of out-of-date dependence information.

The separation of execution semantics and dependence information has also inhibited the development of an adequate semantic account of dependences. Such a semantic account is useful for two reasons. First, it would enable us to prove correctness of transformations. Second, like all semantic descriptions, it has prescriptive value in identifying and fixing weaknesses since constructs that are difficult to model semantically are usually difficult to use in practice — for example, consider the unstructured `goto` in high-level programming languages [9]. A first step towards a semantic account of dependences has been taken by Selke [16] and Cartwright and Felleison [5], who give a λ-calculus based execution model for the special case of program dependence graphs arising from a structured programming language without aliasing or arrays. Their approach is limited by the simple execution model that underlies it. For example, loops are represented as tail-recursive procedures and loop execution is modeled in terms of infinite unfoldings of tail-recursive procedures. It is unclear how one should view transformations such as loop interchange in such a model. In the absence of an updatable store, assignments to possibly aliased variables involve code that checks whether variables are indeed aliased, which results in significant code expansion[2]. Nevertheless, we note that difficulties in modeling conditional assignments in program dependence graphs has led to the proposal of the program representation graph which has turned out to be better suited than the program dependence graph for problems such as integration of program versions [11]; this illustrates the prescriptive power of giving semantics to dependence information.

In a previous paper [14], we showed that all these problems can be avoided by using an executable representation of dependences called the *dependence flow graph*. The

[2] A similar problem is encountered in using the static single assignment form [8].

dependence flow graph can be viewed either as a data structure incorporating dependence information or as a program that can be executed. Our execution model is called *dependence-driven execution* and is a generalization of the tagged-token dataflow model of computation; the generalization permits imperative updates to memory locations. Dependence flow graphs have the following advantages:

1. Since the representation is executable, we can use abstract interpretation to design optimization algorithms, facilitating systematic algorithm development and proof of correctness [7]. From a software engineering perspective, this is advantageous because algorithms based on abstract interpretation have the same structure, which permits code sharing.

2. Algorithms based on abstract interpretation of dependence flow graphs are efficient. Abstract interpretation fell out of favor when the compiler community realized that optimization algorithms based on dependences were more efficient than abstract interpretation algorithms for the same problem, since abstract interpretation had to be done on the control-flow graph. The dependence flow graph is an executable representation of dependence information, and abstract interpretation algorithms that use the dependence flow graph do not suffer from this problem. We have used this idea to design a simple global constant propagation algorithm that is as powerful as the complex one of Wegman and Zadeck that uses both the control-flow graph and def-use chains [14].

3. The dependence flow graph is compact — it is asymptotically smaller than the data dependence graph.

Interestingly, dependence flow graphs incorporate all the advantages of recently proposed representations such as the program dependence graph and web [10, 3], program representation graph [5], static single assignment form [8] and continuation-passing style [12].

In our earlier presentation, we considered only scalars, and arrays were handled by treating them as scalars (that is, an assignment to an array element was treated as an assignment to the entire array). However, over the last fifteen years, a number of subscript analysis tests, such as the GCD test and Banerjee test [18], have been developed, which provide finer-grain dependence information. These tests not only yield information about the existence of dependences, but they also give semantic information such as dependence *directions* and *distances* needed to parallelize programs more effectively. The contribution of this paper is to show how this information can be incorporated into dependence flow graphs, and demonstrate the ability of our execution model to exploit the parallelism

that this exposes. As a bonus, we show that problems faced by conventional dependence representations in dealing with reduction operations, imperfectly nested loops and non-nested loops [18] pose no problem for us. For readers familiar with the concept of streams in functional languages, our results can be interpreted as a generalization of the ideas of Landin who first used streams to model functionally the execution of loops in imperative languages [4][3].

The rest of the paper is organized as follows. In Section 2, we introduce dependence flow graphs and their execution model. In Section 3, we include distance and direction vector information in our execution model. In Section 4, we show how reduction operators and producer-consumer parallelism fit naturally in our framework.

2 The Dependence Flow Graph

In this section, we give a brief introduction to the dependence flow graph and its execution model. The reader who is interested in a more detailed account is referred to our earlier paper [14]. We assume that the reader is familiar with the program dependence graph and with the tagged-token dataflow model. Hereafter, we abbreviate dependence flow graph as DFG and program dependence graph as PDG.

Figure 1 shows a small program for computing partial sums, along with its PDG and DFG representations. The PDG consists of control and data dependences. In Figure 1b, there are flow dependences from S_1 to S_3, from S_2 to S_3 (the value of `I`), from S_2 to itself, and from S_3 to S_4. The control dependence of S_3 on the `for` statement is represented by a dashed line. Note that control and data dependences are represented by distinct edges.

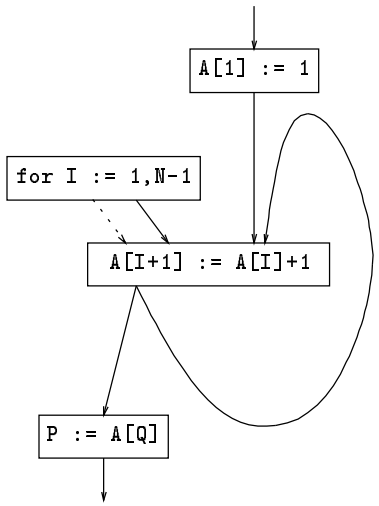
In the DFG, control information is threaded into data dependences through the use of dependence operators, and data dependences between statements having different control dependences pass through these operators. For example, the flow dependence from S_1 to S_3 in the PDG is from a statement outside a loop to one that is inside the loop; in the DFG, this dependence edge becomes a path punctuated by the $\texttt{loop}_1$ operator. The flow dependence from S_3 to itself also passes through this operator. Similarly, the dependence from S_3 inside the loop to S_4 outside the loop passes through a `sync` operator. Thus, traditional data dependence edges become dependence paths in our representation; perhaps surprisingly, this results in an asymptotically smaller representation, since these paths may share vertices [14].

The integration of control and data dependences enables us to give a parallel, composi-

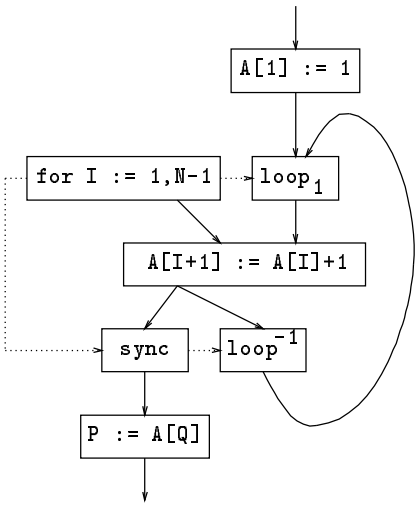
[3]No knowledge of streams is needed to read this paper, since we have tried to avoid such 'cultural' references.

S_1 : A[1] := 1
S_2 : for I := 1,N-1
S_3 : A[I+1] := A[I]+1
S_4 : P := A[Q]

(a) Source Program



(b) PDG Representation



(c) DFG Representation

Figure 1: Summation Example

tional execution semantics to dependence flow graphs. To introduce this parallel semantics, we first consider the sequential execution of statements in a control-flow graph modeled in a dataflow style as follows. The program counter is represented by a token whose arrival at the input of a statement signifies that the statement may execute. Execution begins by passing the token to the first statement in the control-flow graph. On receiving the token, a statement executes by reading its operands from a global store, performing the specified computation, writing the result into the store, and then passing the token along the control-flow edge to its successor. Conditional statements pass the token down the control-flow edge indicated by the computed predicate.

In this simple sequential model, the arrival of the token at the input of a statement signifies that all prior statements have executed; therefore, all dependences of the statement are satisfied and the statement may execute. Since a statement does not usually depend on all statements preceding it, we can introduce parallelism into the execution by notifying a statement as soon as all its dependences are satisfied, even if not all prior statements have executed. In the dependence flow graph, tokens flow down dependence paths and represent

satisfied dependences; a statement executes once tokens have arrived on all incoming edges, and tokens are produced on its outedges and enable subsequent statements to execute. To distinguish between statement instances in different iterations, we tag each token with an iteration number, and require that a statement in iteration i must receive a token with tag i on each input before executing in that iteration. Once the statement has executed, it produces tokens with tag i at its outputs.

Consider the graph in Figure 1c. The instance of statement S_3 in the first iteration depends on a flow dependence from S_1 and the value `I` = 1 from the `for` statement. When S_1 executes it produces a token δ which flows to the $\texttt{loop}_1$ operator. The $\texttt{loop}_1$ operator tags this token, producing $\delta.1$, effectively notifying S_3 that the flow dependence in the first iteration is satisfied. Tokens may also carry values. The `for` statement produces tokens carrying the appropriate value of `I` for each iteration; in the first iteration, S_3 receives the token $\langle 1 \rangle.1$ representing the value `I` = 1 with tag 1. When both tokens, $\delta.1$ and $\langle 1 \rangle.1$ have arrived at S_3, the statement executes and produces a token $\delta.1$ on its outedge. The $\texttt{loop}_1$ operator increments the tag on this token by 1 and sends it to S_3; this token satisfies the flow dependence for the instance of S_3 in the second iteration.

In general, an instance of S_3 in iteration i+1 depends on an instance of itself in iteration i. When S_3 in iteration i executes, it produces a token $\delta.i$ which flows to the $\texttt{loop}_1$ operator[4]. The $\texttt{loop}_1$ operator increments the token's tag by 1, producing the token $\delta.i+1$ which flows to S_3 in iteration $i+1$. (The reader familiar with the tagged-token dataflow model will recognize that the $\texttt{loop}_1$ operator implements the function of the D operator in that model [2].)

Since we do not know which element of array `A` is read by statement S_4, we conservatively say S_4 depends on all instances of S_3. The `sync` operator implements this 'barrier' synchronization for this dependence from all iterations of the loop by waiting until a token from every iteration has arrived on its input edge before producing a token δ on its output.

A detailed account of the semantics of dependence operators is given in Section 3. For the reader familiar with the static single assignment form (SSA), we point out some important differences between that representation and ours. In SSA, variables are single assignment as in functional languages: if X and Y are two possibly aliased variables, an assignment to either variable must be followed by statements that test whether the other variable is actually an alias, and if so, perform a redefinition of that variable. These tests and redefinitions can significantly increase code size [6]. Note that in our model, we retain an imperative, updatable store using dependences to disambiguate which 'version' of the variable is needed by a statement; therefore, modeling possibly aliased variables does not

[4]Ignore the $\texttt{loop}^{-1}$ operator for now; its function is described in Section 3.

result in code explosion. Another major difference is illustrated by the $\texttt{loop}_1$ operator. At join points in the control-flow graph, such as at the bottom of conditionals and at the top of loops, the SSA representation introduces ϕ-functions that serve to combine dependences on the same variable; thus, the SSA would have a ϕ-function where we have introduced the $\texttt{loop}_1$ operator. In DFG's, we have a variety of operators that combine dependences together in many ways. In Section 3, we use this flexibility to incorporate distance and direction information into DFG's.

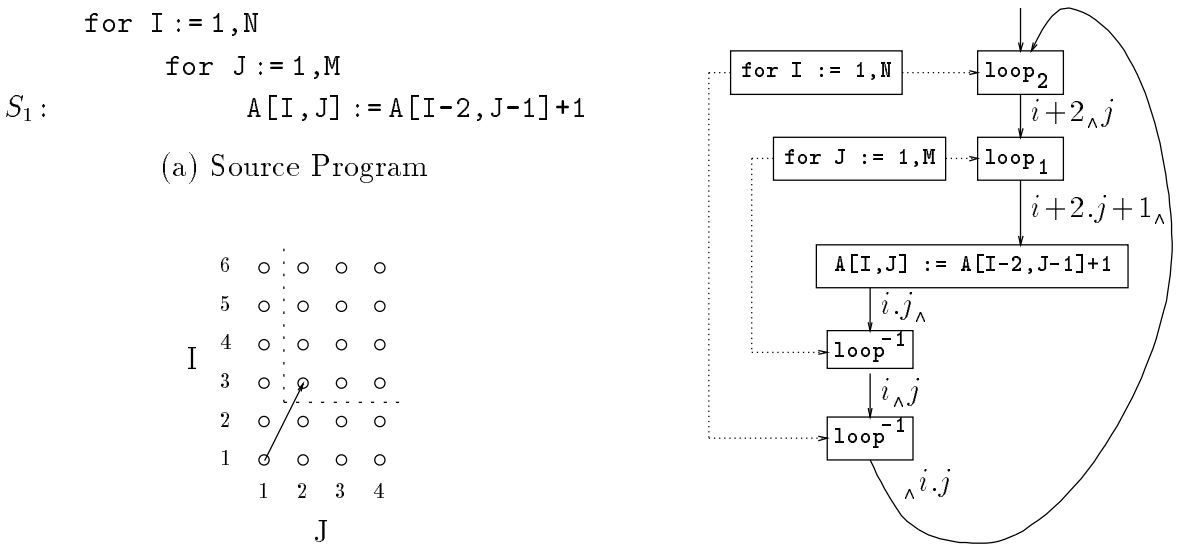
3 Representing Distance and Direction

In the previous section, we showed how a simple loop-carried dependence could be represented within the dependence-driven execution model by augmenting tokens with iteration tags that are manipulated by the $\texttt{loop}_1$ operator. In this section we present the full method for representing distance and direction information operationally. Since this information is relative to an n-dimensional iteration space, one challenge is to retain compositionality; that is, the representation of a loop nested at level k should depend only on the k^{th} elements of distance or direction vectors of dependences within the nested loops, and not on surrounding loops or even the particular value of k. A main difficulty in accomplishing this is correctly generating dependences for initially satisfied iterations. We first show the normal behavior of operators representing distance and direction information and then discuss their initialization behavior.

3.1 Distance Operators

In Figure 1, the loop-carried dependence has a distance of 1 and this fact is reflected operationally in the $\texttt{loop}_1$ node: when a statement instance in iteration i completes, a token $\delta.i$ is generated and passed to the statement instance in iteration $i+1$ by the $\texttt{loop}_1$ node. In general, a dependence between some statements S and S' may have an associated n-dimensional distance vector, $(d_1, \ldots, d_n)$. When statement S executes in iteration $(i_1, \ldots, i_n)$, it generates a token $\delta.i_1 \ldots i_n$ which must be passed to statement S' in iteration $(i_1+d_1, \ldots, i_n+d_n)$. To do this, we must transform $\delta.i_1 \ldots i_n$ into $\delta.i_1+d_1 \ldots i_n+d_n$.

Our approach is to represent the dependence $S\ \delta_{(d_1,\ldots,d_n)}\ S'$ with a path from S to S' passing through special operators that manipulate iteration tags, effectively passing tokens between points in iteration space. The first special operator, $\texttt{loop}_k$, is a generalization of the $\texttt{loop}_1$ operator; it increments a tag element by the constant integer k. We insert a special pointer symbol ($\wedge$) in the tag to indicate which tag element should be incremented. After incrementing an element, $\texttt{loop}_k$ moves the pointer right one position, so a subsequent



(a) Source Program

(b) Iteration Space

(c) DFG Representation

Figure 2: Distance Vector Example

$\texttt{loop}_k$ will modify the next element; when all elements have been modified, the pointer is to the right of all tag elements and the token is ready to be consumed. The second operator, $\texttt{loop}^{-1}$, simply moves the pointer left one position.

We construct a path from S to S' which passes out through one $\texttt{loop}^{-1}$ node for each of the n nested loops surrounding both S and S', and then passes in through $\texttt{loop}_k$ nodes, where $k = d_i$ for the i^{th}-nested loop. For example, consider the program in Figure 2a. The dependence from S_1 to itself has distance $(2,1)$. When S_1 executes in iteration (i,j), it produces a token $\delta.i.j_\wedge$ intended to satisfy the dependence in iteration $(i{+}2, j{+}1)$. After passing through the first $\texttt{loop}^{-1}$ (associated with the J loop), the token become $\delta.i_\wedge j$; after passing through the second $\texttt{loop}^{-1}$, it is $\delta_\wedge i.j$. Note that the pointer is immediately left of the tag element associated with the outer loop. $\texttt{loop}_2$ consumes the token and produces $\delta.i{+}2_\wedge j$; $\texttt{loop}_1$ consumes this token and produces $\delta.i{+}2.j{+}1_\wedge$, allowing S_1 to execute in iteration $(i{+}2, j{+}1)$ as required.

Using the $_\wedge$-pointer to encode nesting level allows the operators to be compositional: the function of $\texttt{loop}_k$ nodes does not depend on their position in the graph. We will show in section 4 how our choice of operators also leads to natural extensions of distance and direction vectors such as representing reductions and producer-consumer parallelism.

Several additional points should be noted. First, the distance vector for any dependence

can be read directly from the $\texttt{loop}_k$ nodes along the path representing the dependence. Also, dependences can be accessed by dimension; transformations such as loop interchange can easily access a particular distance vector element of all loop-carried dependences.

Although we have described the behavior of dependence operators using the $_\wedge$-pointer, readers familiar with the concept of streams in non-strict functional languages such as Id [2] should note the correspondence with streams and stream operators. The tokens flowing down a dependence arc can viewed as elements of a non-strict data structure called a stream whose elements can be used before the entire data structure is produced. On a token, the integer(s) to the right of the $_\wedge$-pointer constitute the position of that token in the stream. The dependence operators we have discussed can be interpreted as stream operators. A full discussion of this connection is beyond the scope of this paper.

3.2 Initially Satisfied Dependences

In the partial sums example (Figure 1), the first iteration received a token representing its satisfied flow dependence from a statement outside the loop, while subsequent iterations received tokens from the previous iteration. In the second example (Figure 2), multiple iterations depend on statements outside the loops; we say these statements are in *initial iterations*. In Figure 2b, the initial iterations are below and left of the dotted line. Statements in iterations above and right of the dotted line receive tokens representing satisfied flow dependence from previous iterations. In this section we describe how a single token representing access to the entire array is expanded into tokens satisfying dependences in the initial iterations. In section 4 we take a different approach to dependences in initial iterations based on producer-consumer parallelism.

Let $I_1, \ldots, I_n$ be the loop indices and $N_1, \ldots, N_n$ be the number of iterations for n nested loops. Let $(d_1, \ldots, d_n)$ be the distance vector associated with some dependence between statements within the nested loops. To simplify the discussion, assume all distances are non-negative. For any dimension l, all iterations $(i_1, \ldots, i_l, \ldots, i_n)$ are initial provided $1 \leq i_l \leq d_l$. In other words, the set of initial iterations may be decomposed by dimension into the union of subregions:

$$\bigcup_{l=1,\ldots,n} \{(i_1, \ldots, i_n) \mid 1 \leq i_l \leq d_l\}$$

In Figure 2b, this L-shaped region is decomposed by dimension as:

$$\begin{array}{lll} I = 1: & L = & \{(1,1),(1,2),(1,3),\ldots,(1,M)\} \ \bigcup \\ I = 2: & & \{(2,1),(2,2),(2,3),\ldots,(2,M)\} \ \bigcup \\ J = 1: & & \{(1,1),(1,2),(1,3),\ldots,(1,N)\} \end{array}$$

Since each $\texttt{loop}_{d_i}$ node knows d_i, it should be able to generate tokens for the appropriate subregion of the initial iterations. The main difficulty is suppressing the generation of multiple tokens for iterations in the intersection of the above subregions[5]. We accomplish this by adding a new pointer symbol ($_+$) which may take the place of $_\wedge$ in the right-most position of a partially expanded tag. When a single token, δ_+, is consumed by $\texttt{loop}_{d_1}$ at the outermost loop, the node generates $\delta.1_\wedge, \ldots, \delta.(d_1)_\wedge$ and $\delta.(d_1+1)_+, \ldots, \delta.(N_1)_+$. The $_\wedge$-pointer with nothing to its right indicates that all innermost $\texttt{loop}_k$ nodes should expand their dimension's range completely; the $_+$-pointer with nothing to its right indicates that the next nested loop should expand its range just as this one did, generating tokens with added suffix $.i_\wedge$ for $1 \leq i \leq d_2$, and added suffix $.i_+$ for $d_2 < i \leq N_2$. At the inner loop, the tokens produced ending with $_\wedge$ exactly cover the initial iterations, and the tokens ending with $_+$ cover the remaining iterations and are simply ignored (consumed without action) by statements.

Thus, in Figure 2(c), $\texttt{loop}_2$ consumes δ_+ and generates $\delta.1_\wedge, \delta.2_\wedge, \delta.3_+, \delta.4_+, \ldots, \delta.N_+$. Each of these tokens is consumed by $\texttt{loop}_1$ and generates:

$$
\begin{array}{lcccccc}
\delta.1_\wedge & \rightarrow & \delta.1.1_\wedge, & \delta.1.2_\wedge, & \delta.1.3_\wedge, & \ldots, & \delta.1.M_\wedge \\
\delta.2_\wedge & \rightarrow & \delta.2.1_\wedge, & \delta.2.2_\wedge, & \delta.2.3_\wedge, & \ldots, & \delta.2.M_\wedge \\
\delta.3_+ & \rightarrow & \delta.3.1_\wedge, & \delta.3.2_+, & \delta.3.3_+, & \ldots, & \delta.3.M_+ \\
\vdots & & \vdots & & & & \\
\delta.N_+ & \rightarrow & \delta.N.1_\wedge, & \delta.N.2_+, & \delta.N.3_+, & \ldots, & \delta.N.M_+
\end{array}
$$

Tokens ending with $_\wedge$ satisfy the initial iterations, whereas tokens ending with $_+$ are ignored (consumed without action) by statement S_1. As instances of S_1 execute in initial iterations, tokens are produced which flow through the loop operators and satisfy instances of S_1 in subsequent iterations.

3.3 Direction Operators

Direction information is a conservative approximation to distance information. Thus, it is not surprising that the operators for encoding direction information are closely related to the `loop` operators. Rather than consuming a single token and then generating a token k iterations away, the $\texttt{loop}_<$ operator waits until it has consumed all tokens of the form $\delta.I_\wedge j.J$ for $1 \leq j < i$ before generating a token for iteration $\delta.I.i_\wedge J$. Here, I, J are any constant index strings. Similarly, the other direction operators wait until consuming tokens

[5]It is possible to define an operational semantics based on environment bindings rather than token passing in which satisfying a dependence multiple times is allowed, but we want to show that this can be done within the dataflow context as well.

Node	Input	Output	
`loopC`	$\delta.I_+$ or $\delta.I_\wedge$	$\delta.I.i_\wedge$	for $1 \le i \le N$
$\texttt{loop}_k$	$\delta.I_\wedge$	$\delta.I.i_\wedge$	for $1 \le i \le N$
if $k \ge 0$:	$\delta.I_+$	$\delta.I.i_\wedge$	for $1 \le i \le k$
		$\delta.I.i_+$	for $k < i \le N$
	$\delta.I_\wedge i.J$	$\delta.I.i{+}k_\wedge J$	for $1 \le i \le N-k$
if $k < 0$:	$\delta.I_+$	$\delta.I.i_\wedge$	for $N-\lvert k\rvert \le i \le N$
		$\delta.I.i_+$	for $1 \le i < N-\lvert k\rvert$
	$\delta.I_\wedge i.J$	$\delta.I.i{+}k_\wedge J$	for $\lvert k\rvert < i \le N$
$\texttt{loop}^{-1}$	$\delta.I.i_\wedge J$	$\delta.I_\wedge i.J$	
`sync`	$\delta.I.i_\wedge$ or $\delta.I.i_+$	$\delta.I_+$	when inputs consumed for $1 \le i \le N$

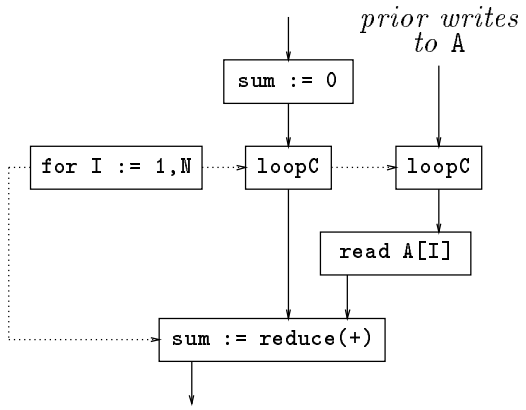
Figure 3: Loop Nodes

Node	Input	Output	
$\texttt{loop}_<$	$\delta.I_\wedge j.J$	$\delta.I.i_\wedge J$	when $\delta.I_\wedge j.J$ defined for $1 \le j < i$
	$\delta.I_\wedge$	$\delta.I.i_\wedge$	for $1 \le i \le N$
	$\delta.I_+$	$\delta.I.1_\wedge$	and
		$\delta.I.i_+$	for $1 < i \le N$
$\texttt{loop}_>$	$\delta.I_\wedge j.J$	$\delta.I.i_\wedge J$	when $\delta.I_\wedge j.J$ defined for $i < j \le N$
	$\delta.I_\wedge$	$\delta.I.i_\wedge$	for $1 \le i \le N$
	$\delta.I_+$	$\delta.I.N_\wedge$	and
		$\delta.I.i_+$	for $1 \le i < N$
$\texttt{loop}_=$	$\delta.I_\wedge j.J$	$\delta.I.i_\wedge J$	when $\delta.I_\wedge j.J$ defined for $j = i$
	$\delta.I_\wedge$	$\delta.I.i_\wedge$	for $1 \le i \le N$
	$\delta.I_+$	$\delta.I.i_+$	for $1 \le i \le N$
$\texttt{loop}_\neq$	$\delta.I_\wedge j.J$	$\delta.I.i_\wedge J$	when $\delta.I_\wedge j.J$ defined for all $j \ne i$
	$\delta.I_\wedge$	$\delta.I.i_\wedge$	for $1 \le i \le N$
	$\delta.I_+$	$\delta.I.i_+$	for $1 \le i \le N$
$\texttt{loop}_*$	$\delta.I_\wedge j.J$	$\delta.I.i_\wedge J$	for $1 \le i \le N$
			when $\delta.I_\wedge j.J$ defined for $1 \le j \le N$
	$\delta.I_\wedge$	$\delta.I.i_\wedge$	for $1 \le i \le N$
	$\delta.I_+$	$\delta.I.i_+$	for $1 \le i \le N$

Figure 4: Direction-Vector Operators

from the appropriate subrange before generating the i^{th} token. Of interest is the $\texttt{loop}_*$ operator which waits for an entire index range.

```
      sum := 0
      for I := 1,N
S1 :        sum := sum + A[I]
```



(a) Source Program (b) DFG Representation

Figure 5: Reduction Example

4 Extensions to Distance and Direction Vectors

Although distance and direction vectors provide dependence information needed in many loop transformation, not all dependence information needed by transformations is expressible within this framework. In this section we show how important extensions to distance and direction vector information fit naturally into our framework.

4.1 Representing Reductions

A reduction is a statement which accumulates into a single variable the result of applying an associative operator to values from each iteration of a loop. For example, statement S_1 in Figure 5a is a reduction statement; there are output and flow dependences from S_1 to itself, each having distance 1. Recognizing reductions is essential to generating good vector and concurrent code [18]. Many vector machines support vectorization of reductions; without special recognition, reductions appear unvectorizable. Recognizing commutative reduction operators allows more flexible scheduling of iterations on multiprocessors and transformations such as loop interchange which appear illegal otherwise. Thus, reductions may often be handled specially, allowing their loop-carried dependences to be ignored. For this reason, we represent reductions as an operator having no loop-carried dependences.

Recall the `sync` operator described in Section 3.1. This operator accumulates a dependence from each iteration and produces a single token representing the boolean `AND` of these dependences; we could relabel this operator as a `reduce(AND)` to indicate that it is a reduction. We generalize this operator to allow any associative operation in place of `AND`.

In Figure 5b, statement S_1 is represented as a reduction operator. Reduction operators are located at the bottom of loops since they take tokens from every iteration whereas statements are associated with individual iterations.

4.2 Representing Producer-Consumer Parallelism

Another important extension to dependence vectors is the representation of producer-consumer parallelism. For example, information about dependences between non-nested loops is used in generating pipelined code for multiprocessors.

Consider the example in Figure 6. Values written in iteration i of the `I` loop are used in iteration $i+1$ of the `J` loop. Figure 6b shows a conservative representation of this flow dependence from S_1 to S_2; all iterations of the `I` loop must terminate before any iteration of the `J` loop begins. In Figure 6c, we represent more detailed information about the flow dependence; in particular, the producer-consumer relationship between iteration i of the `I` loop and iteration $i+1$ of the `J` loop is indicated explicitly. Note that S_2 executing in the first iteration (`J` = 2) uses `A[1]` which is not written by any instance of S_1, and thus depends on some prior write to `A`. Thus, some prior barrier synchronization provides token δ_+ to the $\texttt{loop}_1$ node, thereby initializing S_2 in iteration 1. Instances of S_2 in subsequent iterations receive tokens from instances of S_1 in a producer-consumer manner.

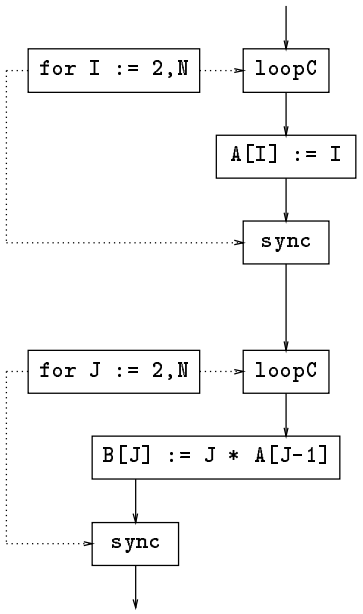
5 Conclusions

Traditionally, dataflow has been viewed as a way of organizing hardware to exploit fine-grain parallelism in functional language programs. We have chosen to use it for a radically different purpose — to organize information in (imperative language) compilers! At a fundamental level, both these uses rely on the ability of dataflow graphs to generate names for all the concurrent activities in the program through mechanisms like tagging. We have described the dependence flow graph which is an executable representation of program dependences; the underlying execution model, called dependence-driven execution, is a generalization of the tagged-token dataflow model that permits imperative memory operations. In this paper, we have shown that dependence distance and direction information can be incorporated into this model through the use of dependence operators, and that the execution model is capable of exploiting the additional parallelism that is exposed through the use of such information. A fundamental question that is preoccupying the field is the determination of how good parallelizing compilers are in exposing parallelism in programs. We believe that the dependence flow graph is the right representation on which to perform such studies. The results of these experiments will be discussed elsewhere.

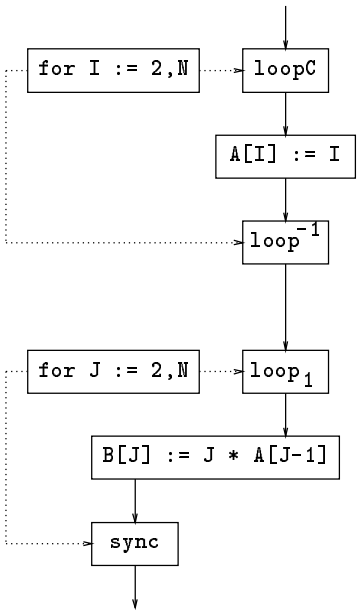
```
        for I := 2,N
S1 :        A[I] := I

        for J := 2,N
S2 :        B[J] := J * A[J-1]
```

(a) Source Program



(b) non-Pipelining



(c) Pipelining

Figure 6: Producer-Consumer Example

References

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

[2] Arvind, K. P. Gostelow, and W. Plouffe. An asynchronous programming language and computing machine. Technical Report 114a, Univ. of Calif., Irvine, Dec. 1978.

[3] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. of the 1990 SIGPLAN Conference on Programming Language Design and Implementation*, pages 257–271, June 1990.

[4] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley, Reading, MA, 1975.

[5] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proc. of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–27, June 1989.

[6] J.-D. Choi. personal communication, 1991.

[7] P. Cousout and R. Cousout. Abstract Interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages*, pages 25–35, Jan. 1989.

[9] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[10] 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.

[11] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proc. of the 1990 SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[12] G. L. S. Jr. and G. J. Sussman. Scheme: An interpreter for extended lambda calculus. Technical Report Memo 349, M.I.T. Artificial Intellegence Laboratory, 1975.

[13] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, New York, 1978.

[14] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: An algebraic approach to program dependencies. In *Proc. of the 18th ACM Symposium on Principles of Programming Languages*, pages 67–78, Jan. 1991.

[15] B. Rosen. Linear time is sometimes quadratic. In *Proc. of the 8th ACM Symposium on Principles of Programming Languages*, Jan. 1981.

[16] R. P. Selke. A rewriting semantics for program dependence graphs. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages*, pages 12–24, 1989.

[17] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *Proc. of the 11th ACM Symposium on Principles of Programming Languages*, pages 291–299, 1984.

[18] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London, 1989.