MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science

Computation Structures Group Memo 181

Translation and Optimization of Data Flow Programs

by

J. Dean Brock
Lynn B. Montz

This paper will appear in the Proceedings of the 1979 International Conference on Parallel Processing.

July 1979

TRANSLATION AND OPTIMIZATION OF DATA FLOW PROGRAMS[a]

J. Dean Brock
Lynn B. Montz
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- We present ADFL, an Applicative Data Flow Language with an iterative control abstraction based on tail recursion and an error-handling scheme appropriate to the concurrency of data flow. An algorithm for translating ADFL programs into data flow graphs is described. These graphs may be executed without possibility of deadlock, but with potential loss of some concurrency, on packet communication systems with bounded buffering, such as the Dennis-Misunas data flow computer. Two techniques for optimizing graphs are given and their effect on performance and correctness is analyzed. One is the insertion of identity operators (buffers) into graphs to increase pipelining. The other is the elimination of unneeded acknowledge signals.

Introduction

In a data flow computer, an operation is performed as soon as its operands have been computed. The machine language is an explicit representation of the data dependencies of program operations. Its programs are directed *data flow graphs* whose nodes are called *operators*. The role of operators in a data flow machine is similar to the role of instructions in a von Neumann machine. The execution of an instruction corresponds to the *firing* of an operator. Each operator has several input and output ports. Whenever an operator fires, it absorbs tokens (values) at its input ports and produces tokens at its output ports. Operators have firing rules which determine when they are *enabled* for firing. These firing rules are based on the presence or absence of tokens on the operator's ports.

When operators are joined to form data flow graphs, the links of the graph are directed from operator output ports to operator input ports. A link transports the results produced at an operator output port to an operator input port. Thus, links form the pathways upon which data *flows* as tokens are absorbed and produced by the firing of operators during the execution of a graph.

(a) This research was supported in part by the Lawrence Livermore Laboratory of the University of California under contract 8545403, in part by the National Science Foundation under research grant MCS75-04060 A01, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661. Part of the research was conducted while Mr. Brock was supported by a National Science Foundation graduate fellowship.

The data flow graph of an elementary expression resembles its parse tree. The graph for computing the distance function:

$$\text{sqrt}((x1-x2)^2 + (y1-y2)^2)$$

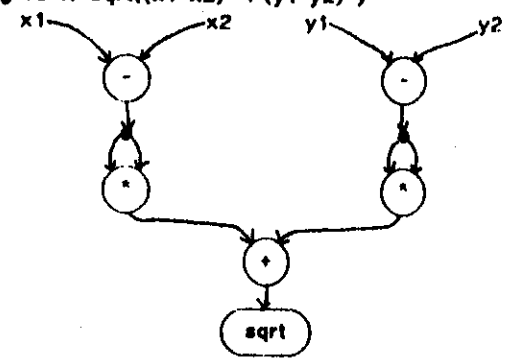
is illustrated in Figure 1. The solid black dot in the figure represents the copy operator which is used to distribute the results of one output port to several input ports. Note how this graph represents the operation dependencies and independencies of the distance function.

Preliminary data flow machine designs have been made by Arvind and Gostelow [2], Davis [5], and Dennis and Misunas [7]. Within these machines, a data flow graph is distributed over a network of processing elements. These elements operate concurrently, constrained only by the operational dependencies of the graph. Thus, a very efficient utilization of the machine's resources appears possible.

ADFL - An Applicative Data Flow Language

Data flow programming languages resemble conventional languages restricted to those features whose ease of translation does not depend on the state of a computation being a single, sequentially manipulated entity. Because the "state" of a data flow graph is distributed for concurrency, *goto*'s, expressions with side effects, and multiple assignments to the same variable are difficult to represent. ADFL, Applicative Data Flow Language, is a simplification of VAL, the value-oriented data flow language being developed by Ackerman and Dennis [1]. A BNF

Figure 1. $\text{sqrt}((x1-x2)^2 + (y1-y2)^2)$



$\text{sqrt}((x1-x2)^2 + (y1-y2)^2)$

specification of the syntax of ADFL follows:

exp ::= *id* | *const* | *exp* , *exp* | *oper*(*exp*) |
let *idlist* = *exp* in *exp* end |
if *exp* then *exp* else *exp* end |
for *idlist* = *exp* do *iterbody* end

iterbody ::= *exp* | iter *exp* |
let *idlist* = *exp* in *iterbody* end |
if *exp* then *iterbody* else *iterbody* end

id ::= "programming language identifiers"

idlist ::= *id* { , *id* }

const ::= "programming language constants"

oper ::= "programming language operators"

The most elementary expressions of ADFL are identifiers and constants. Tuples of expressions are also expressions: One such expression is "x, 5". The application of an operator to an expression is an expression. Although, the BNF specification only provides for operator applications in prefix form, such as "+(x, 5)"; applications in infix form, such as "x + 5", are considered acceptable equivalents (sugarings) and will be used in example ADFL programs. In sequential programming languages execution exceptions are generally handled by program interrupts (signals). This solution is inappropriate for data flow since there is no control flow to interrupt. Applied to "exceptional" inputs, data flow operators yield special error values, such as zero divide or pos over. The documentation of VAL [1] contains a detailed specification of this method of error-handling. For simplicity, only one error value undef is used throughout this paper.

Since ADFL is applicative, it provides for the binding, rather than the assignment, of identifiers. Evaluation of the binding expression:

let y, z = x + 5, 6 in y * z end

implies the evaluation of "y * z" with y equal to "x + 5" and z equal to 6. The result of binding is local: the values of y and z outside the binding expression are unchanged.

ADFL contains a conventional conditional expression, but has an unusual iteration expression. Evaluation of the iteration expression:

for *idlist* = *exp* do *iterbody* end

is accomplished by first binding the *iteration identifiers*, the elements of *idlist*, to the values of *exp*. Note from the BNF specification of *iterbody*, that the evaluation of the *iteration body* will ultimately result in either an expression or the "application" of a special operator iter to an expression. This application of iter is actually a tail recursive call of the iteration body with the iteration identifiers bound to the "arguments" of iter. The iteration is terminated when the evaluation of the iteration body results in an ordinary, non iter, expression. The value of this expression is returned as the value of the iteration expression. The following iteration expression computes the factorial of n:

```
for i, y = 1, 1 do
  if i < n then iter i + 1, y * i else y end
end
```

Syntactic restrictions which ensure that expressions are used only when appropriate in arity and type have been omitted from this discussion. Elsewhere in this volume, Dennis and Weng [8] define arity and type restrictions for a data flow language similar to VAL and, consequently, ADFL. Their language differs from ours in emphasis: They present an abstract interpreter with a dynamic allocation scheme for executing graphs and, accordingly, emphasize procedural control abstractions. We investigate the execution of statically allocated graphs (data flow machine language programs) and, accordingly, emphasize iterative control abstractions.

Translation of ADFL

The translation algorithm of ADFL consists of two functions $\mathcal{T}$, mapping ADFL expressions into their data flow graph implementations, and $\mathcal{T}_1$, mapping ADFL iteration bodies into their implementations. The graph implementing an expression or iteration body has an input port for each free variable of the expression or iteration body. For an expression *exp* which returns *n* values when evaluated, $\mathcal{T}[\![exp]\!]$ has *n* output ports. Recall that evaluation of an iteration body will yield either results to be re-iterated or results to be returned by the containing iteration expression. The graph $\mathcal{T}_1[\![iterbody]\!]$ has an output port iter? which signals which possibility has occurred and sets of output ports for each possibility: I output ports for values to be iterated and R output ports for values to be returned.

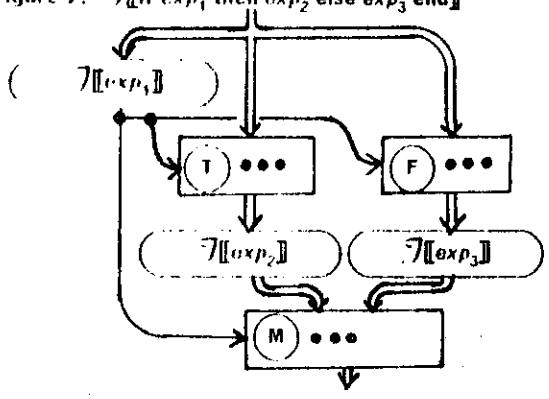
The translation algorithm for ADFL resembles previous translation schemes of Dennis [6] and Weng [11]. A detailed recursive definition of the algorithm over the eleven cases of the BNF specification of the syntax of ADFL has been given by Brock [3]. For brevity, only the cases of the conditional expression, the conditional iteration body, and the iteration expression will be examined in detail. It is assumed that most readers, informed that the graph of Figure 1 may be re-labeled:

$\mathcal{T}$[[let dx,dy = x1-x2,y1-y2 in sqrt(dx*dx+dy*dy) end]]

will discover the translation of the eight "trivial" cases.

The graph $\mathcal{T}[\![\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3 \text{ end}]\!]$ is shown in Figure 2. The graph contains three subgraphs, $\mathcal{T}[\![exp_1]\!]$, $\mathcal{T}[\![exp_2]\!]$, and $\mathcal{T}[\![exp_3]\!]$, and several gates. The T gate has a control input port (entering its left side), a data input port, and an output port. When the T gate fires, it absorbs a token from each input port. If the control token is true, the data token is passed to the output port. If the

Figure 2. $\mathcal{F}$[[if exp_1 then exp_2 else exp_3 end]]

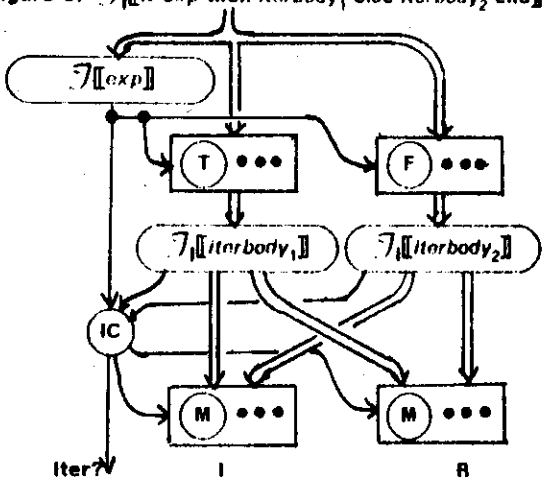


control token is false, the data token is simply absorbed. No output token is produced. The F gate is defined analogously. It passes its data token only if its control token is **false**. By passing inputs to $\mathcal{F}$[[exp_2]], respectively $\mathcal{F}$[[exp_3]], through T gates, respectively F gates, controlled by the output of $\mathcal{F}$[[exp_1]], the proper subexpression is "enabled" during data flow evaluation of the conditional expression. The results of $\mathcal{F}$[[exp_2]] and $\mathcal{F}$[[exp_3]] are merged by M gates. The M gate has one control input port, two data input ports, and one output port. Its control token selects the data token to be passed. If the control value is the error value **undef**; each T or F gate absorbs a data token and produces no output tokens, and each M gate produces **undef** and absorbs no input tokens. Thus, data flow evaluation of a conditional expression yields a tuple of **undef**'s if the condition is **undef**.

$\mathcal{F}_i$[[if *exp* then $iterbody_1$ else $iterbody_2$ end]], the conditional iteration body graph illustrated in Figure 3, is similar to the conditional expression graph. With T and F gates, the output of the expression subgraph, $\mathcal{F}$[[*exp*]], enables one of the iteration body subgraphs, $\mathcal{F}_i$[[$iterbody_1$]] and $\mathcal{F}_i$[[$iterbody_2$]]. The selected subgraph will produce output at either its I or R output ports, according to its **iter?** output: **true**, for I outputs to be iterated; **false**, for R outputs to be returned. Using the output of the expression subgraph and the **iter?** outputs of the iteration body subgraphs, the IC gate calculates three iteration control outputs: the graph **iter?** output and the control tokens for the M gates producing the graph I and R outputs. The table at the bottom of Figure 3 gives the firing rules of the IC gate. Note that, if the output of the expression subgraph is **undef**, the conditional iteration body graph will produce **false** at its **iter?** port, thus announcing termination of iteration, and will produce **undef** at its R output ports.

The graph $\mathcal{F}$[[for *idlist* = *exp* do *iterbody* end]] is shown in Figure 4. This cyclic graph is formed by using M gates to merge the outputs of $\mathcal{F}$[[*exp*]] and the I outputs of $\mathcal{F}_i$[[*iterbody*]] and by routing the merged outputs into the input ports of $\mathcal{F}_i$[[*iterbody*]] labeled by identifiers of *idlist*. The control input port of each M gate is connected to the

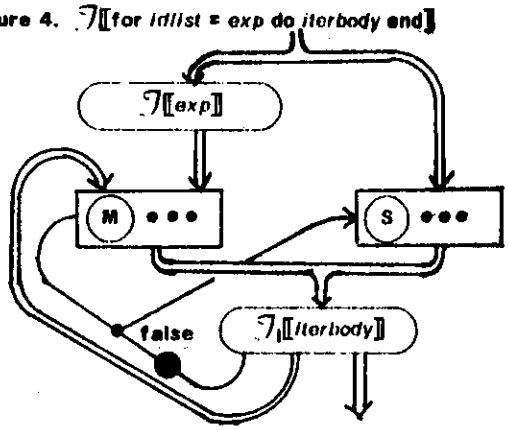
Figure 3. $\mathcal{F}_i$[[if *exp* then $iterbody_1$ else $iterbody_2$ end]]



IC gate firing rules

inputs			*outputs*		
true	**true**	-	**true**	**true**	-
true	**false**	-	**false**	-	**true**
false	-	**true**	**true**	**false**	-
false	-	**false**	**false**	-	**false**
undef	-	-	**false**	-	**undef**

Figure 4. $\mathcal{F}$[[for *idlist* = *exp* do *iterbody* end]]



iter? output of $\mathcal{F}_i$[[*iterbody*]]. The connecting arc contains an initial **false** token to ensure that the first data value is selected from $\mathcal{F}$[[*exp*]]. Thereafter, data tokens are selected according to **iter?**. A **true** **iter?** token, signalling continued iteration, selects the data tokens of the I output ports. A **false** **iter?** token, signalling termination, re-initializes the M gates for subsequent iteration expression evaluations. Identifiers which are free in *iterbody* but are not contained in *idlist* are routed through S gates. For **false** control tokens, the S gate absorbs, stores, and outputs its data tokens. For **true** control tokens, it produces its stored value and absorbs no data tokens. Thus, the S gate stores new values when

evaluation of the iteration expression begins and produces them at each subsequent iteration step. Like the M gate, it is initialized with a false control value.

Brock [4] has verified this translation algorithm by proving it to be consistent with a denotational [10] specification of ADFL. In the proof, data flow arcs are assumed to be implemented by infinite (unbounded) queues. The transformations described subsequently will relax this requirement without affecting the correctness of translation.

Transformations of Data Flow Graphs

In proposed data flow machines of the Dennis-Misunas [7] design, operations are held in instruction cells which contain a register for each input arc. These registers are effectively an implementation of data flow arcs as queues of capacity one. The implication of the bounded arcs is that operators must be prevented from producing new tokens until their output arcs are empty. This behavior is ensured by modifying the firing rules so that no operator is enabled if a token is present on *any* of its output arcs.

By performing a transformation, illustrated in Figure 5, which replaces each arc of the graph by an appropriate data/acknowledge arc pair (d/a arc pair), the effect of the modified firing rule can be explicitly built into the graph: The presence of a token indicates that the corresponding data arc is empty. As a consequence, operator firing rules revert to the original format of depending only on the presence of tokens on input (including acknowledge) arcs, where the previous enabling requirement that output arcs be empty has been replaced with the requirement that acknowledge inputs be present.

Montz [9] and Dennis and Misunas [7] have shown that graphs of data flow programs may be executed without deadlock when arcs are implemented as data/acknowledge pairs. Consequently, the correctness of the translation algorithm is not affected by this transformation. However, the implementation is not without cost. Aside from the obvious overhead involved in incorporating acknowledge arcs and tokens, the constraints which they impose on the token flow through the graphs may cause bottlenecks. In response to these issues, Montz [9] has developed optimization techniques specifically aimed at either increasing the throughput by balancing the token flow or decreasing the overhead by removal of unnecessary acknowledge arcs.

Balancing Token Flow

The goal of the optimization to balance token flow through the graph is to increase throughput by modifying the graph to display maximum pipelining. The bottleneck problem, and therefore application of the optimization, arises in acyclic segments of a data flow graph. A clear illustration of the problem and solution is shown in the Figure 6 graph, the implementation of the ADFL expression: ⟦if f=1 then *f1* else *f2* end⟧. Although successive sets of inputs should be processed simultaneously, the control structure of the graph dictates that the overlap be very minimal. In order for a second set of values to enter the branches of the conditional, both α and β (Figure 6) must fire a second time presenting the sets of T and F gates with new control inputs. However, α cannot fire a second time until the M gate to which it also sends a control input has fired, to produce an acknowledge. Thus the d/a arc pair connecting α and the M gate (shown with slashes in Figure 6) creates a bottleneck whose severity depends on the depth of the computations performed within the branches of the conditional.

Eliminating this behavior so that successive sets of values may pipeline through the graph can be accomplished by inserting identity operators (buffers) along the slashed arc, breaking it into d/a arc segments which consequently

---

Figure 5. Replacement of one-place buffers with d/a arc pairs

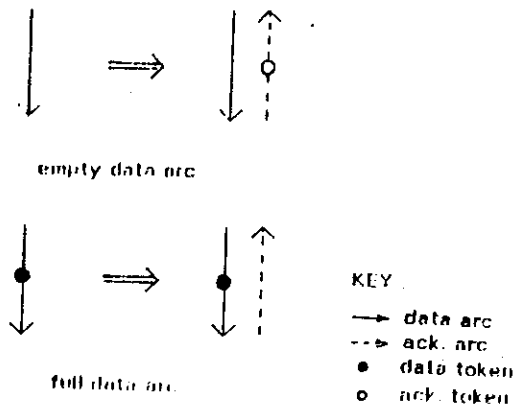
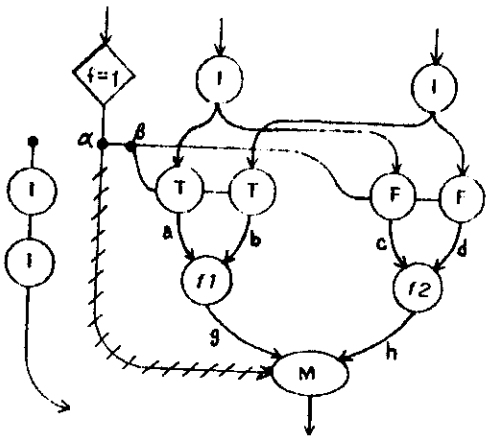


Figure 6. Insertion of buffers for a conditional expression



allow α to fire several times before forcing the M gate to fire. For the Figure 6 graph, this is accomplished by replacing the slashed arc with the arc segment shown to its immediate left. To generalize this optimization technique, a determination of the ideal number and location of inserted buffers must be made. This requires an analysis of data flow graph execution.

Though the data flow computer is asynchronous, it can be made to model a synchronous machine by assuming that during any given unit of time all enabled operators must fire and produce a result. This approximates optimal program execution by preventing an enabled operator from remaining enabled and thereby slowing up processing for any length of time.

Referring to Figure 6, we note that each input set to the graph will result in the production of a token on the control (slashed) arc and tokens that will be processed by either *f1* or *f2*. While under the "synchronous machine" assumption the tokens being processed by the functional operators can move one step through the graph during every time unit, the control token on the slashed arc cannot, restricting throughput to an output every fifth time unit. Adding identity operators to equalize buffer capacities achieves maximum pipelining, or equivalently, the optimal throughput of an output every *second* time unit. The algorithm presented below equalizes buffering.

Algorithm to Maximize Pipelining

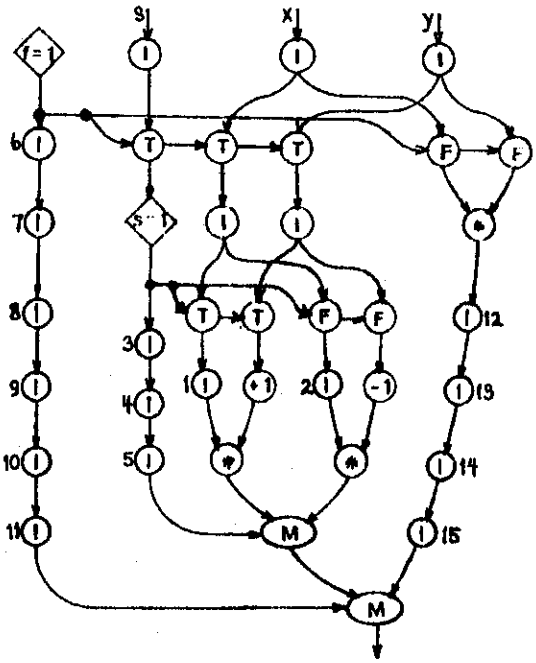
> Starting from each graph input, descend through the graph assigning consecutive numbers to the arcs joining successive sets of operators until a multi-input operator is encountered. Compare the arc numbers on the input arcs of the operator and:
>
> (a) If equal, continue the arc numbering process
> (b) If not equal, balance the arcs by inserting identity operators into the lower numbered arcs. Renumber the modified arcs and continue the arc numbering process.

Note that if the operator is an M gate, the comparison and balancing process described above must involve all *three* input arcs, using the highest numbered arc as the goal. Figure 7 shows the result of applying this algorithm to the graph translation of the following program segment:

> **if f=1 then if s=1 then x*(y+1) else x*(y-1) end else x*y end**

For reference purposes, the added identities have been numbered. Identities I1 and I2 have been added in response to the imbalances which occur when comparing arc numbers on the input arcs to the *multiplication* operators. I3 through I5 are added in response to the comparison of input arcs to the inner M gate. Note that as specified in the algorithm, arc number comparisons involve all three M gate input arcs. Finally, operators I6 through I15 are introduced as a result of comparing input arcs to the outer M gate.

Figure 7. Example of maximal pipelining



In applying the algorithm to this example, there are several interesting observations to make. Recall from the algorithm, that M gate comparisons must involve the two data arcs and the control arc. The algorithm modifies the graph to achieve maximum pipelining by making buffering capacities of the paths through the graph to the control arc and two data arcs the same. However, while each branch of the conditional operates in conjunction with the control arc, the branches themselves are *independent*. Thus, while each branch must pipeline with the control path, they need not necessarily pipeline with each other. If the two conditional paths are of different lengths, the pipelining choices available are to equalize the control path with either the shorter or the longer conditional branch, or to equalize all three. The latter of these, implemented by the algorithm above, achieves best throughput, but has the disadvantage of causing the insertion of additional identity operators in the shorter conditional branch. The other two choices recognize the independence of the two conditional paths and avoid excess buffering, but possibly at the cost of reduced throughput.

A factor not yet considered which interacts with this pipelining choice is the frequency with which graph paths are taken. In Figure 7 each input set can take any of three paths corresponding to the three possible states of f and s. If, for example, the pattern of input sets is such that no one of the three paths is taken twice in a row, identity operators I1 and I2 would be unnecessary and could be removed without decreasing the throughput. Illustrations of this point can be found in Montz [9].

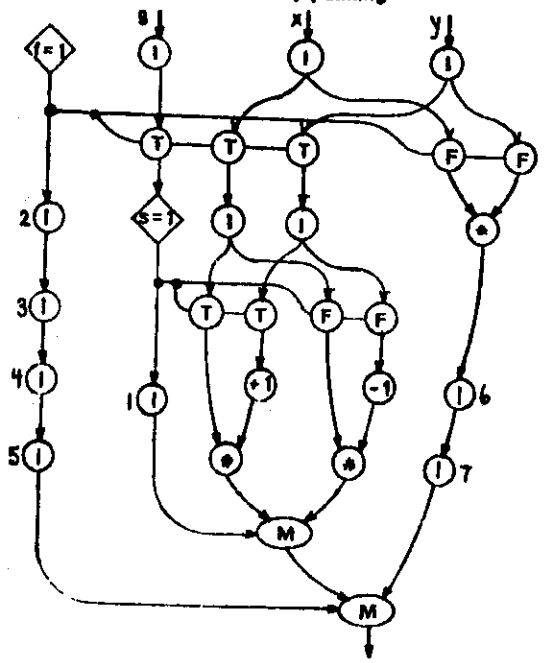
The discussion of trade-offs and options to consider in maximally pipelining data flow graphs, indicates that the advantage of smaller size resulting from a less than maximally pipelined graph may be worth a decrease in throughput. Some key issues influencing the choice might include cost of identity operations, processor utilization, token flow patterns, and width and depth of program. By modifying the pipelining algorithm, we can produce data flow graphs which display *limited pipelining*, meaning that the delay between an operator's firing and receiving appropriate acknowledge signals may be several time units. For example, it is possible to specify that the delay in sending acknowledge signals be no greater than two time units. The change to the algorithm, which involves balancing arcs to within a specified bound, allows a graph to be easily reconfigured to display different degrees of pipelining, and thereby provides a feasible and practical control method of studying varying levels of pipelining in a graph. Though the details of the modified algorithm will not be given, we proceed by briefly comparing the Figure 7 graph with that of Figure 8 which can be produced using a limited pipelining algorithm.

The most striking contrast between the fully pipelined graph and this partially pipelined version is the large reduction in inserted identity operators, from 15 to 7. The question which arises is whether the cost of this reduction is a decrease in performance, where the Figure 7 graph displays the optimum performance by producing an output every *second* time unit. An analysis of several token flow patterns using different successions of input sets shows that the limited pipelining scheme does *not* necessarily degrade the throughput. This can be seen by pipelining three sets of inputs through the Figure 8 graph assuming that they respectively follow the paths indicated by the f-s values: true-true, false, and true-false.

Once an actual data flow machine is available, a study of the number of inserted identity operators vs. throughput trade-off should provide insight into the direction to take concerning optimization. This information in combination with a particular application should indicate other optimization possibilities; for instance, concentrating on only the main source of bottleneck within a graph. For the conditional construct this point appears to be the control arc to the M gate. Modifications of the pipelining algorithm could also be weighed more realistically as alternative approaches.

A final point to note in the consideration of this pipelining optimization strategy is that conditional constructs and general compositions of operators turn out to be fairly representative of the type of graphs for which this optimization is applicable. In fact, this optimization approach is basically inappropriate for an iterative process whose function is to modify and recycle a single set of inputs at a time (although subgraphs within an iteration may be pipelined). Thus an alternative optimization which aims to minimize the number of acknowledges in a graph by eliminating those which are unnecessary has been developed.

Figure 8. Example of limited pipelining



Eliminating Acknowledges

This optimization technique aims at decreasing overhead by removing acknowledge arcs which are not necessary to maintaining safe operation. This *safety* requirement is equivalent to guaranteeing that at most one token will reside on any arc of a data flow graph at any time. An examination of various ADFL constructs leads to the identification of arc pairs which are candidates for *acknowledge arc removal*. The strategy will be to develop a rule specifying the requirements for acknowledge arc removal for each candidate arc pair identified in the construct. By recursively applying the resulting set of rules to the data flow graph translation of an ADFL program, acknowledge arc removal for all candidate arc pairs can be determined.

To illustrate the analysis and formulate the desired rules, we begin by considering the data flow graph translation of the general conditional construct shown in Figure 6. As in the preceding section, the discussion centers on the arc pair connecting α and the M gate. However, while overcoming the restricting behavior of this arc pair was the focus of that optimization aimed at increasing pipelining, the restriction is an advantage to the process of eliminating acknowledges. Specifically, α, which *cannot* fire a second time until it receives an acknowledge from the M gate, guarantees that a second input set will not be within the branches of the conditional until processing of the preceding set has completed. Each input set (which will be processed by either *f1* or *f2*) places a token on the controlling arc of the M gate and a data token on each of the arcs labeled either a and b, or c and d, depending

respectively on whether the control token was *true* or *false*. Assuming that *f1* and *f2* are well-formed, an output should appear on arc g (assuming the control token was *true*) within finite time, with the impossibility of a second token appearing on arc g, or of *any* token appearing on arc h until the M gate has fired. This firing simultaneously processes the token on arc g and sends an acknowledge token to α, consequent to which a successive input set may enter a branch of the conditional. This behavior guarantees that the acknowledge arc of the arc pair denoted by g can be safely removed. By an analogous argument we can remove the acknowledge arc of the arc pair labeled h.

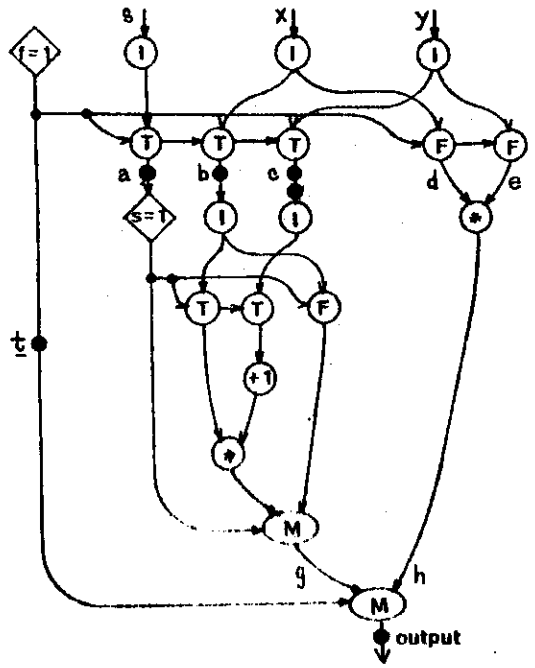
Using similar reasoning one might be tempted to remove the acknowledge arcs from arc pairs a, b, c, and d under the assumption that once a set of tokens has entered a branch of the conditional, the tokens must be used by the appropriate function to produce the corresponding output. However, a consideration of the Figure 9 data flow graph will show that removal of acknowledge arcs for these arc pairs is dependent on the subgraphs represented by *f1* and *f2*.

The Figure 9 data flow graph is a translation of the following ADFL program segment:

```
if f=1 then if s=1 then x*(y+1) else x end
       else x*y end
```

Assume that the outer decision operator evaluates to *true* and that of the inner conditional construct previously represented by *f1*, evaluates to *false*. The important point to note is that an output can be produced using only the tokens on arcs *a* and *b*. The token on arc c need not propagate through the graph, and may in fact still be on the arc when a successive set of values arrives. Removal of c's acknowledge arc would make it possible to reach the *unsafe* token configuration shown in Figure 9. This example shows that the necessity of acknowledge arcs for arc pairs *a* through *e* is dependent on whether or not their values are guaranteed to be used in producing the outputs of their appropriate subgraph (*f1* or *f2*). An analysis of the subgraphs in Figure 9 reveals that tokens arriving on arcs *a*, *b*, *d*, and *e* *must* be used to produce their corresponding output, while the need of a token arriving on arc *c* is dependent on the outcome of the inner decision operator. Therefore, we must leave c's acknowledge arc, but can remove those of arc pairs *a*, *b*, *d*, and *e*.

Figure 9. Unsafe token configuration resulting from removal of c's acknowledge arc



This analysis, specific to the conditional construct, results in designating all input arc pairs to the *f1* or *f2* subgraphs subject to rule C1 with regard to acknowledge arc removal:

> C1: The acknowledge arc of an input arc pair to a subgraph may be removed if any token arriving on the arc *must* be used in producing the output of the subgraph.

This form of analysis must be recursively applied to subgraphs in determining acknowledge arc removal for both inner constructs and outer arc pairs. It is interesting to note that this rule could be applied at the source level by taking the intersection of variables appearing in the then and else clauses. Variables found in the intersection would be guaranteed to be used in producing the output, and in graph form would not require acknowledge arcs.

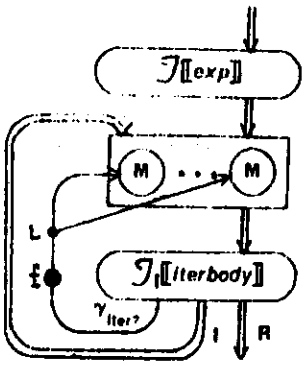
Referring to Figure 6, the arc pairs presenting inputs to the T and F gates have not yet been discussed with regard to acknowledge arc removal. Since the only way to guarantee the absence of a token on any of these data arcs is via the presence of a token on the corresponding acknowledge arc, these acknowledge arcs must remain. A final point concerns the initially discussed control arc connecting α with the M gate which may not need an acknowledge arc. The control arc of the inner conditional construct of Figure 9 is an example of such an occurrence which can be characterized by rule C2:

> C2: The acknowledge arc of the control arc connecting α and the M gate of a conditional construct can be removed if the acknowledge arc of the output arc pair of the M gate has been removed.

Developing a complete recursive algorithm to determine acknowledge arc removal in data flow graphs requires this type of analysis for each ADFL construct.

As a second example, we briefly examine the iteration construct shown in Figure 10 to identify candidate arc pairs for acknowledge arc removal. The arc labeled $T_{iter?}$, the control output of the iteration, provides the controlling value for the sequence of M gates handling the presentation of

Figure 10. $\mathcal{T}$[[for *idlist* = *exp* do *iterbody* end]]



successive sets of inputs to the iteration body. Since the $\gamma_{iter?}$ value is dependent on at least some of the M gate inputs, a number of them must fire before a second $\gamma_{iter?}$ value is produced. This necessarily implies the firing of the copy operator, "L", to present the M gates with new control inputs needed to re-enable them, ensuring that the $\gamma_{iter?}$ output arc from the iteration body to L must be empty for a successive $\gamma_{iter?}$ value to be produced. Consequently, the $\gamma_{iter?}$ arc needs no acknowledge. No such guarantee can be made for the arcs between the copy operator and M gates, acknowledges for which can be conditionally removed subject to rule T1:

> T1: The acknowledge arc for an arc pair between operator L and the sequence of M gates can be removed if its data value *must* be used in producing the $\gamma_{iter?}$ value.

The output arc of the iteration body labeled I represents the arc pairs for the iteration variables. The analysis for these arcs is more complex and is governed by the following rule:

> T2: The acknowledge arc of an I (iteration) arc pair can be removed if *either*
>
> (1) The iteration body cannot emit a value on that output arc until it has absorbed the corresponding input value on the corresponding input arc.
>
> (2) The $\gamma_{iter?}$ value depends on the corresponding input arc.

Examples involving the iteration construct, as well as an expanded discussion of these rules and an analysis of the remaining arcs can be found in Montz [9].

Conclusions

We have described a data flow language, an algorithm for translating its programs into data flow graphs, and two techniques for optimizing these graphs for execution with data flow machines of the Dennis-Misunas [7] design. While the two optimization methods have been presented as isolated techniques, they must be integrated into a single procedure for application to a given program.

We have not compared the costs of operation of the Dennis-Misunas [7] computer design with that of the Arvind-Gostelow [2] design, which avoids conflicts through the use of tagged values rather than acknowledge tokens.

Acknowledgments

We wish to thank Jack Dennis for supervising the theses [3, 9] on which this paper is based and Bill Ackerman for reading and criticizing drafts of this paper.

References

[1] W. B. Ackerman, and J. B. Dennis, *VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual*, Laboratory for Computer Science, Massachusetts Institute of Technology, TR-218, (July, 1979), 80 pp.

[2] Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time", *Information Processing 77: Proceedings of IFIP Congress 77* (August, 1977), pp. 849-853.

[3] J. D. Brock, *Operational Semantics of a Data Flow Language*, Laboratory for Computer Science, Massachusetts Institute of Technology, TM-120, (December, 1978), 55 pp.

[4] J. D. Brock, *Consistent Semantics for a Data Flow Language*, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Memo 172, (January, 1979), 30 pp.

[5] A. L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," *Proceedings of the Fifth Annual Symposium on Computer Architecture, Computer Architecture News* 6 (April, 1978), pp. 210-215.

[6] J. B. Dennis, "First Version of a Data Flow Procedure Language," *Programming Symposium: Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science 19* (October, 1976), pp. 362-376.

[7] J. B. Dennis, and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *The Second Annual Symposium on Computer Architecture: Conference Proceedings* (January, 1975), pp. 126-132.

[8] J. B. Dennis, and K.-S. Weng, "An Abstract Implementation for Concurrent Computation with Streams," *Proceedings of the 1979 International Conference on Parallel Processing*, (August, 1979).

[9] I. B. Montz, *Safety and Optimization Transformations for Data Flow Programs*, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, S. M. Thesis in preparation.

[10] R. D. Tennent, "The Denotational Semantics of Programming Languages," *Communications of the ACM 19* (August, 1976), pp. 437-453.

[11] K.-S. Weng, *Stream-Oriented Computation in Recursive Data Flow Schemas*, Laboratory for Computer Science, Massachusetts Institute of Technology, TM-68, (October, 1975), 93 pp.