[Lee87b] E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow” *IEEE Proceedings*, September, 1987.

[Lee91] E. A. Lee, “Consistency in Dataflow Graphs”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 2, April 1991.

[LeG90] P. Le Guernic and T. Gautier, “Data-flow to Von Neumann: the SIGNAL approach”, in *Advanced Topics in Data-Flow Computing,* ed. J.-L. Gaudiot and L. Bic, Prentice-Hall, 1991.

[Mcg82] J. R. McGraw, “The VAL Language: Description and Analysis”, *ACM Trans. on Programming Languages and Systems,* ***4***(1), pp. 44-82, January 1982.

[Nik88] R. S. Nikhil, “ID Reference Manual”, Computation Structures Group Memo 284, August 29, 1988, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

[Pin85] K. Pingali and Arvind, “Efficient Demand-Driven Evaluation. Part I”, *ACM Trans. on Programming Languages and Systems*, Vol. 7, No. 2, April 1985, p. 311-333.

[Pri91] H. Printz, “Automatic Mapping of Large Signal Processing Systems to a Parallel Machine,” Memorandum CMU-CS-91-101, School of Computer Science, Carnegie Mellon University, Ph.D. Thesis, May 15, 1991.

[Sih91] Gilbert C. Sih, “Multiprocessor Scheduling to Account for Inter-processor Communication”, Ph.D. Thesis, Memorandum No. UCB/ERL M91/29, UC Berkeley, CA 94720, April 22, 1991.

[Tur81] D. A. Turner, “The Semantic Elegance of Applicative Languages”, *Proc. of the ACM Conf. on Functional Programming Languages and Computer Architecture*, (Portsmouth, N.H., 1981), p. 85-92.

[Wen75] K.-S. Weng, “Stream-Oriented Computation in Recursive Data Flow Schemas”, Laboratory for Computer Science (TM-68), MIT, Cambridge, MA 02139, Oct. 1975.

5. Conclusions

Dataflow graphs that are strongly consistent and have finite complete cycles can always be scheduled statically and executed in bounded memory. Some strongly consistent graphs without finite complete cycles can also be executed in bounded memory. We have described a clustering technique that can identify such graphs and construct their schedules. However, some perfectly correct graphs may not fit the models described. To execute these, we must resort to the much more costly methods of dynamic scheduling and memory allocation. Nonetheless, even within such uncooperative graphs, large subgraphs are likely to be consistent and have finite complete cycles. These subgraphs should be statically scheduled and memory should be statically allocated. The cost of dynamic scheduling and memory allocation should be incurred only where absolutely necessary.

6. References

[Arv82] Arvind and K. P. Gostelow, "The U-Interpreter", *Computer*, **15(2),** February 1982.

[Arv86] Arvind and D. E. Culler, "Managing resources in a parallel machine", in *Fifth Generation Computer Architectures*, pp. 103-121. Elsevier Science Publishers, 1986.

[Ben88] A. Benveniste, B. Le Goff, and P. Le Guernic, "Hybrid Dynamical Systems Theory and the Language SIGNAL", Research Report No. 838, April 1988, Institut National de Recherche en Informatique et en Automatique (INRIA), Domain de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France.

[Ben90a] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language", to appear in *IEEE Trans. on Automatic Control*, May, 1990.

[Ben90b] A. Benveniste and P. Le Guernic, "A Denotational Theory of Synchronous Reactive Systems", to appear in *Information and Computation*, 1990.

[Bha92] S. Bhattacharyya and E. A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," to appear in *J. of VLSI Signal Processing*, 1992.

[Bie90] J. Bier, E. Goei, W. Ho, P. Lapsley, M. O'Reilly, G. Sih and E.A. Lee, "Gabriel: A Design Environment for DSP," *IEEE Micro Magazine*, October 1990, Vol. 10, No. 5, pp. 28-45.

[Buc91] J. Buck, S. Ha, E. A. Lee, and D.G. Messerschmitt, "Ptolemy: A Platform for Heterogeneous Simulation and Prototyping, to appear in *Proc. 1991 European Simulation Conference*, Copenhagen, Denmark, June 17-19, 1991.

[Dav82] A. L. Davis and R. M. Keller, "Data Flow Program Graphs", *Computer*, **15(2)**, February, 1982.

[Den75] J.B. Dennis, "First Version Data Flow Procedure Language", Technical Memo MAC TM61, May, 1975, MIT Laboratory for Computer Science.

[Hil89] P. N. Hilfinger, "Silage Reference Manual, DRAFT Release 2.0", Computer Science Division, EECS Dept., University of California, Berkeley, CA 94720, July 8, 1989.

[Kav86] K. Kavi, B. P. Buckles, U. N. Bhat, "A Formal Definition of Data Flow Graph Models," *IEEE Trans. on Computers,* C-35(11), November 1986.

[Kos78] P. R. Kosinski, "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs", *Conf. Record of the 5th Ann. ACM Symp. on Principles of Programming Languages*, Tuscon, AZ, 1978.

[Lee87a] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Trans. on Computers,* January 1987.

```
do {
        fire $S_1$;
} while ($b_1$ = TRUE);
```

Treated as a unit, the cluster always consumes and produces exactly one token, so the resulting schedule is

$$S = \{1, S_2, 5\} . \tag{35}$$

For this purpose, the "state" of a dataflow graph has two components, the number of tokens on each arc, and the value of any Boolean tokens on Boolean arcs.

An alternative approach is to construct a schedule with the repetition vector given by (34) with $N = 1$, and observe that two possible states can result from executing this schedule. One of these states is the original state, denoted Σ_1. The second is a new state, Σ_2, shown in figure 17. If a schedule with repetition vector (34) is applied to the starting state Σ_2, then exactly two ending states are possible, Σ_1 and Σ_2. Consequently, a schedule with repetition vector (34) can be repeated indefinitely with no difficulty.

Yet a third alternative approach is to construct a *preamble*, or initial schedule that transforms the graph into one that has no delays on Boolean arcs. For the graph in figure 16, firing actors 1 and 2 constitute such a preamble. After the preamble has executed, then ordinary scheduling methods can be applied.

4.7. Scheduling Graphs That Require Unbounded Memory

All three methods fail, however, for the graph in figure 13. The number of states that would need to be considered is unbounded. For this graph, we offer no alternative to dynamic scheduling with dynamic memory allocation. However, the clustering algorithm still finds well-behaved substructures in the graph; by clustering the graph as much as possible, we reduce the amount of dynamic memory allocation and scheduling that is required.

Some complex and irregular graphs will not be successfully clustered by our algorithm, and state enumeration requires a heuristic to avoid exploring the (possibly infinite) state space forever. Because of this, some graphs that have bounded-memory schedules may not be handled successfully by these techniques. If so, some dynamic memory allocation will be used although it is not actually required. However, graphs composed only of the "well-behaved" structures appearing in the dataflow literature are handled successfully (e.g. [Den75],[Gao92]).

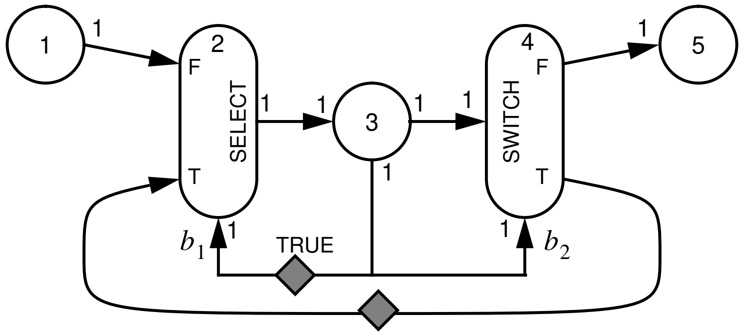

Figure 17: One of two possible ending states for the graph in figure 16 after executing a schedule with repetition vector given by (34).

3. when the augmented cluster is treated as an atomic actor, the graph does not deadlock.

The final test requires a conceptually straightforward extension of the reachability matrix test in [Bha92] to perform symbolic manipulations, rather than just numeric. When a cluster can no longer be augmented, it is treated as an atomic unit and invoked however many times are required to match the "sample rates" of its neighbors. In this repeated invocation of the cluster, we allow an unbounded number of invocations, implemented by embedding the schedule for the cluster inside a while loop. We can still guarantee bounded memory requirements because the cluster internally repeats a complete cycle, and hence has bounded memory requirements. Repetition for a fixed number of cycles and conditional execution are also permitted.

Once a cluster is repeated so that it matches its neighbors, more merging is possible; the process of merging actors into clusters, and of looping the clusters, is repeated until no more changes are possible. The resulting hierarchical structure of clusters reflects the loop structure of the dataflow graph, with loops, *if-then,* and *do-while* constructs.

This procedure is precisely the reverse of the "well-behaved dataflow graph" approach of [Gao92]; instead of building up the dataflow graph out of bounded subunits, the clustering algorithm finds the well-behaved subunits. Consequently, it also permits constructs that are not "well-behaved" but nevertheless correspond to useful and correct programs.

4.6. State Enumeration

A more difficult example is shown in figure 16. This graph is strongly consistent, with

$$\vec{r}(\vec{p}) = \left[(1-p_1)\ 1\ 1\ 1\ (p_2)\right]^T \tag{33}$$

satisfying the balance equations, assuming $p_1 = p_2$. The difficulty arises when we try to construct a finite complete cycle, and we discover that we cannot assure that $t_1 = t_2$. Hence, a schedule with repetition vector

$$\vec{r}(\vec{p}(S)) = \left[(N-t_1)\ N\ N\ N\ (t_2)\right]^T \tag{34}$$

may not satisfy the balance equations. The solution is two-fold. First, we cluster the graph as shown in figure 16. The cluster is internally SDF and can be cyclically scheduled using the firing sequence $S_1 = \{2, 3, 4\}$. Then we construct a schedule S_2 that returns the cluster to its original state:

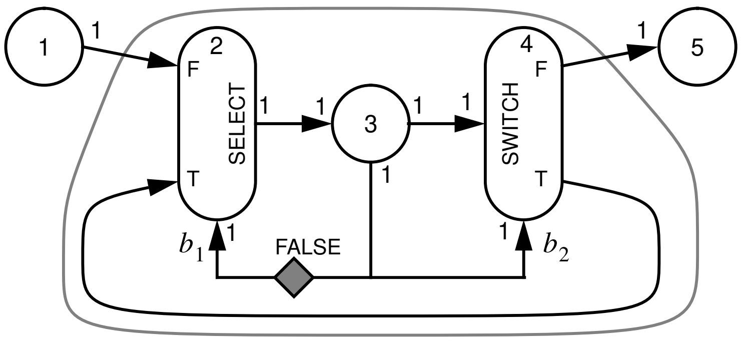

Figure 16: A do-while construct. This graph is strongly consistent, and can execute in bounded memory, but it has no finite complete cycle. Constructing a static schedule requires both clustering and state enumeration.

offering one solution to the balance equations. However, this graph has no finite complete cycle. To see this, let N be the number of repetitions in any S of actors 1, 2, and 3. Any complete cycle S must have an equal number of firings of these three actors, as can be seen from (31). If t_1 is the number of TRUE tokens consumed by actor 2 in S, then

$$\vec{r}(\vec{p}(S)) = \begin{bmatrix} N & N & N & t_1 & \frac{N-t_1}{2} \end{bmatrix}^T . \tag{32}$$

For any N, the last entry may not be an integer.

Consider clustering the graph as shown in figure 15. Actors 1, 2, 3, and 4 are collected into a cluster that internally has a complete cycle and an easily derived annotated schedule. Moreover, a bounded memory schedule can be constructed as follows:

```
repeat_forever {
        n=0;
        do {
                fire 1;
                fire 3;
                fire 2;
                if (t) fire 4;
                n += t;
        } while (n < 2);
}
```

In this pseudo-code notation, we use “t” to denote the Boolean value of the token produced by actor 3. It takes on value 0 or 1.

We have developed systematic methods for constructing clusters like that in figure 15. The method is related to the loop-scheduling of Bhattacharyya and Lee [Bha92], but generalized beyond SDF graphs. Briefly, these methods begin a cluster with a single actor, and augment the cluster with a new actor when

1. the new actor has the same “sample rate” as the cluster (on the connecting arc, the source actor always writes the same number of tokens that the destination actor reads);
2. the augmented cluster has a finite complete cycle, and

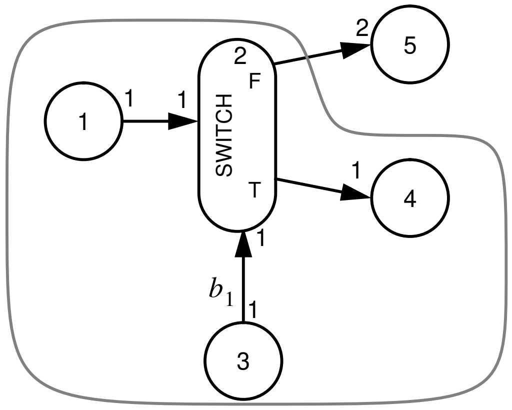

Figure 15: A graph that can be executed in bounded memory, but has no finite complete cycle. Actors 1 to 4 considered alone, however, do have a complete cycle; we can treat them as a cluster and execute the cluster until actor 5’s requirements are met.

requirements. Buffers will have to be dynamically allocated, greatly increasing the cost of implementation compared to static allocation.

4.3. Constructing Annotated Schedules when a Finite Firing Sequence Exists

Assume we have a dataflow graph with no delays on Boolean arcs and with an integer solution to the balance equations. A procedure similar to that of the SDF case can be applied to this dynamic dataflow case to construct annotated schedules. In this paper, we can only briefly describe this procedure, and only for a single processor schedule. For multiple processors, the procedure must be modified to construct an annotated precedence graph, which in turn can be used (in principle) to construct multiprocessor schedules. The annotated precedence graph can also be used to check for deadlock conditions, although in this paper we simply assume deadlock does not occur.

Constructing a single processor annotated schedule requires successively adding actors to a schedule list S, together with the annotations that enable them, until the number of invocations of the actors $\vec{r}(S)$ forms a vector in the nullspace of the appropriate matrix $\Gamma(\vec{p}(S))$, where $\vec{p}(S)$ is a vector of Boolean proportions for the Boolean tokens consumed in S. Note that $\vec{p}(S)$ is not well-defined for streams that have no Booleans consumed yet in S, but these undefined entries also cannot affect the system state, and hence can be ignored.

Define the state of the system to be the number of tokens on each arc (there are no delays on Boolean arcs, recall). The initial state reflects the initial tokens on arcs with delays. We must also uniquely label every Boolean token that is consumed in S. In general, the number of tokens on an arc may be a symbolic function of $\vec{p}(S)$. With a given system state, some actors are conditionally enabled. An actor is conditionally enabled if for some Boolean condition B the number of tokens on its input arcs is greater than or equal to the number of tokens it requires to fire. If B is a function that maps Boolean outcomes onto $\{0, 1\}$, then the number of tokens the actor puts on its output arcs is B times the number of tokens it produces on each output when it fires. This symbolic function is used to update the state of the system after conditionally scheduling the enabled actor. The number of firings of the actor is incremented by the symbolic function B. Scheduling is completed when the vector $\vec{r}(S)$ is in the nullspace of $\Gamma(\vec{p}(S))$. Except for the increased complication of having to manipulate symbolic expressions, the scheduling procedure is not much more complicated than for the SDF case.

4.4. Conditions for Bounded Cycle Length

Elsewhere in this paper we assume that graphs do not deadlock; however, if deadlock actually occurs, this fact will be detected during the process of building the annotated precedence graph (if will be impossible to complete its construction). The existence of a bounded integer solution to the balance equations, together with the successful construction of the annotated precedence graph, are necessary and sufficient conditions for bounded cycle length. In addition to guaranteeing bounded memory requirements, this condition can be important in scheduling the graph with a hard real-time constraint, as it may permit a proof that the constraint is always met regardless of the Boolean outcomes.

4.5. Clustering

Some graphs do not have bounded cycle length, but nonetheless can be scheduled with bounded memory. Consider the example in figure 15. This graph is strongly consistent, with

$$\vec{r}(\vec{p}) = \begin{bmatrix} 1 & 1 & 1 & p_1 & \frac{(1-p_1)}{2} \end{bmatrix}^T \qquad (31)$$

$$\vec{b}(S) = \vec{b}(0) + \Gamma(\vec{p}(S))\vec{r}(S)\ , \tag{28}$$

analogous to (8).

The problem of scheduling a dynamic dataflow graph at compile time is one of devising a complete cycle. We wish to find a complete cycle that can be expressed as a finite annotated firing sequence. Since such a sequence returns the number of tokens on each arc to its original value, it can be repeated indefinitely. Furthermore, since the annotated sequence consists of a bounded number of firings of each actor, then the total memory requirements for indefinite execution of the graph can be bounded.

It is immediate that any graph possessing a finite complete cycle can be executed in bounded memory. Since the state of the graph after the cycle is the same as before, then memory requirements on arcs cannot accumulate over multiple cycles. Within a cycle, every actor fires a finite number of times with an easily derived upper bound, so there is no possibility of unbounded storage requirements.

Unfortunately, a finite complete cycle does not exist for some strongly consistent graphs. Consider again the example in figure 13. This graph is consistent under the condition $p_1 = p_2$, in which case a solution to the balance equations is given by (12). This condition is satisfied under the probabilistic formulation of the token-flow model, but will not be satisfied for any finite firing sequence in general. From (12) we can see that if a complete sequence S exists for this graph, it must contain the same number of firings of actors 1, 2, 5, 6, and 7. Let that number be N. Hence $n_1(S) = n_2(S) = N$. Then the repetition vector for S is given by

$$\vec{r}(\vec{p}(S)) = \begin{bmatrix} N \ N \ [N - t_1(S)] \ \ [t_2(S)] \ N \ N \ N \end{bmatrix}^T . \tag{29}$$

But this repetition vector will not satisfy the balance equations unless $t_1(S) = t_2(S)$. Furthermore, this latter condition cannot be satisfied unless the last Boolean produced by actor 7 in the sequence S has value TRUE, returning the Boolean arc to its original state.

The example in figure 12 has a different problem with the same net effect. Let S be any firing sequence that contains N firings of actors 1, 2, 5, 6, and 7. From (21), it is clear that any complete cycle S must fire these actors the same number of times. N, therefore, gives the number of Booleans consumed in S on both b_1 and b_2. Hence the repetition vector is

$$\vec{r}(\vec{p}(S)) = \begin{bmatrix} N \ N \ \dfrac{N - t_1(S)}{2} \ \dfrac{t_1(S)}{2} \ N \ N \ N \end{bmatrix}^T . \tag{30}$$

Since $t_1(S)$ can take on any integer value between zero and N, there is no assurance for any N that the third and fourth entry will be integers. Since no firing sequence can fire any actor a non-integer number of times, there is no finite complete cycle for this graph.

Since not all graphs of interest have finite complete cycles, we will develop below two methods, called *clustering* and *state enumeration*, that can identify infinite complete cycles that can be scheduled in bounded memory for some types of graphs. However, even with these methods, some perfectly correct graphs will elude our analysis. Our strategy will be to identify subgraphs for which we can find a complete cycle, construct a static schedule, and if possible statically allocate memory for the arcs. Graphs for which we cannot find a complete cycle, but which are consistent, will require dynamic memory allocation and probably dynamic scheduling. In many cases, this extra overhead is fundamentally required by the program. Although it is certainly possible to schedule the graph in figure 13, no scheduling policy can guarantee finite memory comes of the Boolean proportions $t_i(S)/n_i(S)$. Furthermore, we require that all arcs in the graph have the same number of tokens after executing S as before, or equivalently, that

$$\Gamma(\vec{p}(S))\vec{r}(S) = \vec{o}. \tag{23}$$

We elaborate these requirements below, first showing how they are all satisfied for an appropriately chosen S for the if-then-else example.

4.1. Annotated Firing Sequence

In the if-then-else example in figure 8, consider the following *annotated firing sequence*:

$$S = \{1, 7, 2, (1 - t_1(S))\,3, (t_1(S))\,4, 5, 6\}. \tag{24}$$

The parenthesized expressions give the conditions under which the corresponding actor should fire. In this case, actor 3 will fire if $t_1(S)$ is unity, and otherwise 4 will fire. Exactly one token is consumed in S from each Boolean stream, so $n_1(S) = n_2(S) = 1$. Furthermore, $t_1(S) = t_2(S)$.

The firing sequence in (24) can be repeated indefinitely, and hence constitutes an admissible periodic annotated schedule. For this example, the number of firings of each actor in S is given by

$$\vec{r}(S) = \begin{bmatrix} 1 \\ 1 \\ t_1(S) \\ 1 - t_1(S) \\ 1 \\ 1 \\ 1 \end{bmatrix}. \tag{25}$$

Since $t_1(S)$ can only take on values zero or one, $\vec{r}(S)$ is a vector of integers for any realization of S. Further, defining

$$\vec{p}(S) = \begin{bmatrix} t_1(S) \\ t_1(S) \end{bmatrix} \tag{26}$$

$C(\vec{p}(S))$ is always true, and using the topology matrix in (9), we get,

$$\Gamma(\vec{p}(S))\vec{r}(S) = \vec{o}, \tag{27}$$

where again $\vec{o}$ is the vector of zeros.

4.2. Complete Cycles

A *complete cycle* is a sequence of (possibly annotated) firings where the number of tokens on each arc after the sequence is equal to the number before. Fractional firings are not allowed, of course. It is similar to the "firing cycle" of Kavi, et. al. [Kav86], but applies to more general dataflow graphs. Any S satisfying (27) for all possible outcomes of all Boolean proportions $t_i(S)/n_i(S)$ is a complete cycle. Mathematically, for any annotated sequence S, the state of the FIFO buffers after the sequence fires is given by effect on p_2, so $p_1 = p_2$, and the graph is strongly consistent. Suppose, however, that the value of the initial token is TRUE, and b_1 consists of an infinite sequence of FALSE tokens. Then the SELECT can never fire.

Note that the examples in figures 12 and 13 have bounded memory requirements under a tagged-token schema, which does not require that actors process tokens in order. A variant of figure 12, shown in figure 14, has unbounded memory requirements under any dataflow schema. The feedback loop effectively enforces serial execution of actors 6 and 7, so that tokens into 6 must be processed in order.

We recognize, however, that perfectly correct programs can have unbounded memory requirements (consider a program that recognizes a context-free grammar, for example). Such programs fundamentally require dynamic memory allocation and at least some measure of dynamic scheduling. However, we wish to isolate the particular dataflow arcs and actors that require this, and incur the additional runtime cost *only where needed*. The vast majority of arcs and actors in a typical program have no such requirement, and can be implemented with much lower run-time cost on routine hardware.

4. Scheduling

To construct a static schedule, we need to consider not probabilistic behavior of the dynamic dataflow actors, but rather the behavior over short, well-defined firing sequences. To do this, we modify the token flow model to consider not the probability p_i, but rather the proportion of TRUE tokens in a finite firing sequence S. For a particular firing sequence S, let $t_i(S)$ denote the number of TRUE tokens consumed from stream b_i within the sequence. Let $n_i(S)$ denote the total number of tokens consumed from b_i in S. This is an unknown that we will only manipulate symbolically. Define the vector

$$\vec{p}(S) = \begin{bmatrix} t_1(S)/n_1(S) \\ t_2(S)/n_2(S) \\ \dots \end{bmatrix}, \tag{22}$$

where the size of the vector equals the number of Boolean streams in the graph. We wish to find a finite S such that $C(\vec{p}(S))$ is true for any possible outcomes of the Boolean proportions $t_i(S)/n_i(S)$. Let $\vec{r}(S)$ denote the number of firings of each actor in S. Note that $\vec{r}(S)$ must evaluate to an integer vector (actors can't fire a fractional number of times) regardless of the out-

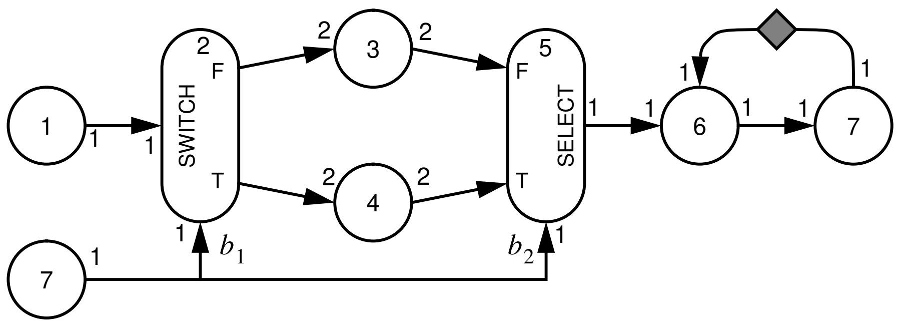

Figure 14: A variant of figure 12 that cannot execute in bounded memory even under a tagged-token schema.

2. Can we find an upper bound on the memory required to execute the graph?

We wish to answer both questions in finite time for an infinite execution of the graph. Not surprisingly, we will find that we cannot always answer these questions definitively. However, for a large class of dataflow graphs, we can answer quickly in the affirmative and synthesize a lean implementation. Dataflow graphs for which the answer to both questions is yes have most, if not all the desirable properties of the clean dataflow graphs of Davis [Dav82], the well-behaved dataflow graphs of Arvind [Arv82] and Gao [Gao92], and k-bounded loops of Arvind and Culler [Arv86].

Strong consistency is necessary, but not sufficient, to ensure that a graph will execute in bounded memory. Perfectly correct (and strongly consistent) programs exist that cannot be executed indefinitely in bounded memory. Consider the example in figure 12, given by Gao, et. al. [Gao92]. A solution to the balance equations for this system is

$$\vec{r}(\vec{p}) = \left[2\ 2\ (1-p)\ p\ 2\ 2\ 2\right]^T, \tag{21}$$

where $p = p_1 = p_2$. If a FIFO schema on the arcs is enforced, the memory required to execute this graph is not bounded. Suppose for example the Boolean streams b_1 and b_2 consist of a single TRUE token followed by an unbounded number of FALSE tokens. The SELECT actor cannot fire until actor 4 has fired, and actor 4 cannot fire until a second TRUE token arrives. Hence, the arc between actor 3 and the SELECT will require unbounded memory. A related example, devised independently of [Gao92], is shown in figure 13. As in figure 8, this graph is consistent if $p_1 = p_2$. In the probabilistic interpretation of p_i, the initial token implied by the delay has no

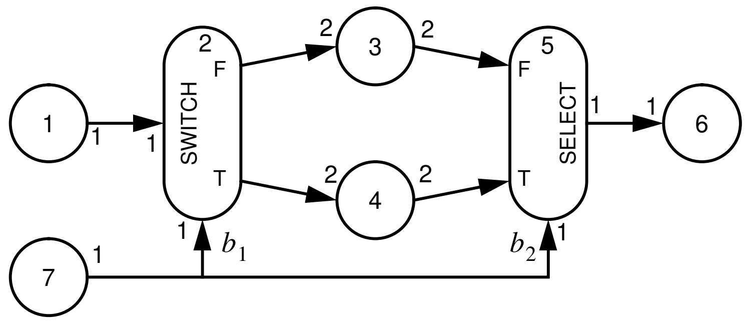

Figure 12: An example of a strongly consistent system that cannot execute in bounded memory if a FIFO protocol is enforced on the arcs.

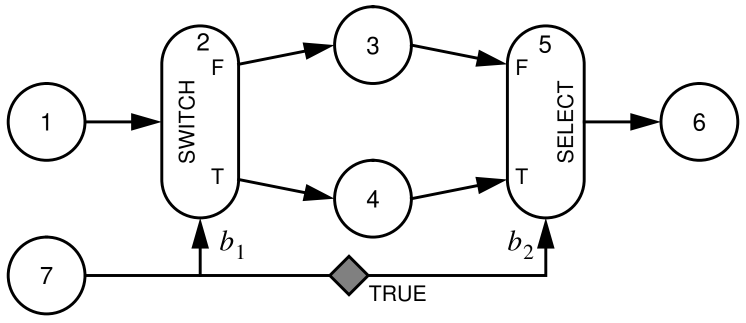

Figure 13: A graph that is strongly consistent but cannot execute in bounded memory if a FIFO schema is enforced on the arcs.

which is always true by the multiplication rule in probability. Consequently, this graph is strongly consistent.

The above analysis must be performed carefully, ensuring that when probabilities are combined at a logical operator, they are combined for tokens that are simultaneously consumed. If, for example, a delay were introduced in the path of one of the Boolean streams in figure 11, the analysis might require information about the joint statistics of successive tokens in a Boolean stream. This knowledge may not be available to a compiler, or even to the programmer. Similarly, Booleans that are generated by testing non-Boolean streams may have joint statistics that are difficult or impossible for a compiler to discern. In principle, a compiler that does elaborate semantic analysis of the program and has complete information about the inputs might be able to use this type of analysis to verify consistency, but such a compiler is not practical. A practical compiler will alert the programmer with a warning when a conditionally consistent dataflow graph is encountered, and the conditions cannot be proven to be satisfied.

3.4. The Limitations of Consistency

In all cases above, inconsistency indicates a peculiar graph that is probably erroneous. This suggests, at minimum, defining languages that do not admit inconsistent graphs. Conditionally consistent graphs could be admitted with a warning, because a compiler cannot always determine the relationships between separate Boolean streams.

The question arises, then, of whether consistency can be used in compilation of dataflow graphs. Synchronous dataflow graphs, for example, can be statically scheduled onto multiple processors [Lee87a]. Moreover, memory can be statically allocated for each arc, leading to particularly simple implementations on ordinary processors [Lee87b]. We wish to determine when these properties extend to more general dataflow graphs. For a given graph, we wish to answer the following questions:

1. Can we define a schedule at compile time for execution of the graph?

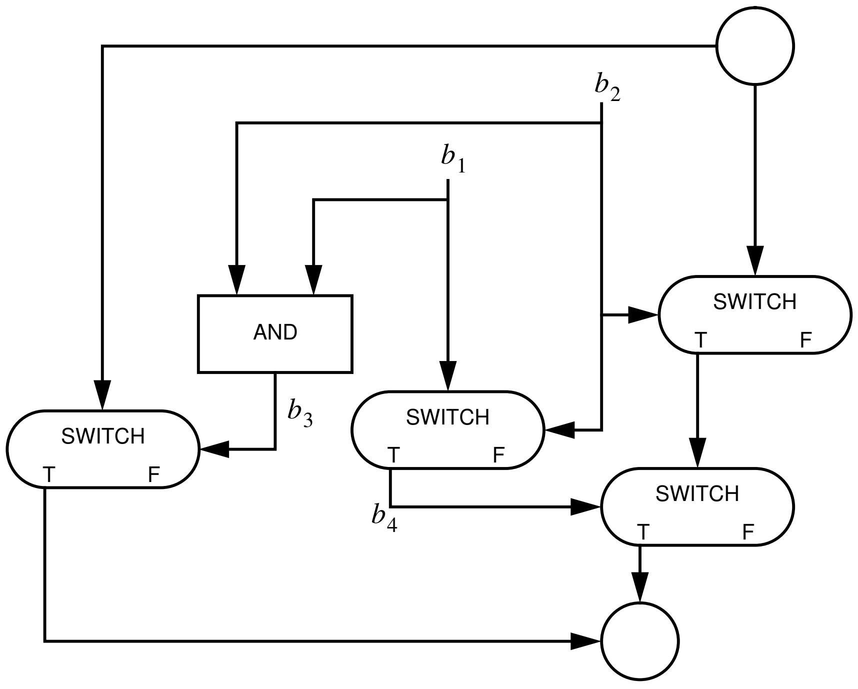

Figure 11: An example of a graph that appears at first glance to be only conditionally consistent, but on further analysis proves to be strongly consistent.

they are only valid for tokens that are simultaneously consumed. However, even with this restriction, they can be used sometimes to determine that conditionally consistent graphs are in fact strongly consistent.

Consider the example in figure 11. This dataflow graph can be shown to be consistent if and only if

$$p_3 = p_2 p_4 . \tag{16}$$

In other words,

$$C(\vec{p}) = \begin{cases} \text{TRUE;} & p_3 = p_2 p_4 \\ \text{FALSE;} & otherwise \end{cases} . \tag{17}$$

From the relationships in figure 10,

$$p_3 = Pr[b_1, b_2] \tag{18}$$

$$p_4 = Pr[b_1 | b_2] . \tag{19}$$

Hence condition (16) is equivalent to

$$Pr[b_1, b_2] = p_2 Pr[b_1 | b_2] , \tag{20}$$

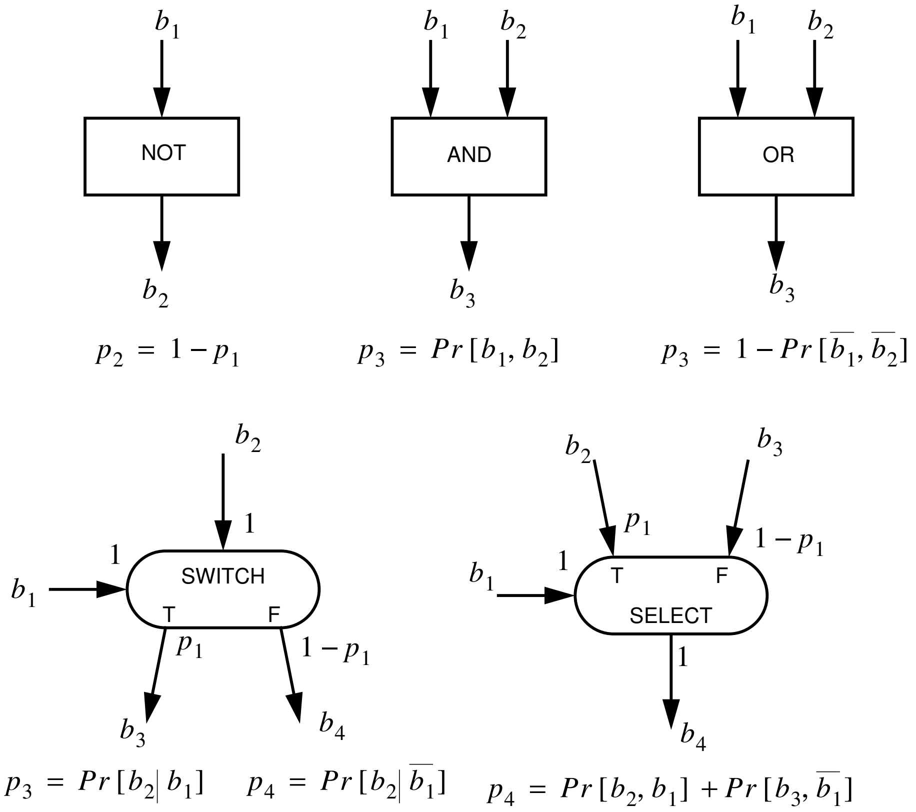

Figure 10: Actors that operate on Boolean streams may produce new Boolean streams that are interrelated as shown in these examples.

$$\Gamma(p_c) = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & (1-p_c) & -1 \end{bmatrix}. \tag{13}$$

This assumes that actors 1 and 3 consume and produce only single tokens when they fire. Unless

$$p_c = 0, \tag{14}$$

meaning that c is always false, this matrix has full rank. Since given the information we have, there is no reason to believe that $p_c = 0$, the graph is not strongly consistent.

Intuitively, each time actor 1 fires, it supplies a token to both the SWITCH (actor 2) and actor 3. However, actor 3 can only consume that token if the SWITCH also provides it with a token, which occurs only if the Boolean c is false. Hence, each time c is true, another token accumulates on arc 2. These tokens will accumulate indefinitely, requiring unbounded memory.

Some additional notation will help to make the ideas below precise. Let $\vec{p}$ be a vector containing the Boolean probabilities in a system. Then define an *indicator* function $C(\vec{p})$ that has value "true" if $\Gamma(\vec{p})$ has the appropriate rank for this particular $\vec{p}$. Otherwise, it has value "false". Hence, a particular graph will be said to be consistent subject to $C(\vec{p})$. It is strongly consistent if $C(\vec{p})$ is always true, and inconsistent if $C(\vec{p})$ is always false. For the system in figure 9, (14) is equivalent to

$$C(\vec{p}) = \begin{cases} \text{TRUE;} & p_c = 0 \\ \text{FALSE;} & otherwise \end{cases}. \tag{15}$$

3.3. Mutually Dependent Booleans

The if-then-else of figure 8 is consistent subject to condition (11), which we know to be trivially true, so the graph is strongly consistent. In general, however, there may be dependencies between Boolean streams that are harder to discern. In particular, when Boolean streams are interrelated, a graph that appears to be only conditionally consistent may in fact be strongly consistent. The interrelationships between Boolean streams can come about from logical operators, such as those in figure 10. In that figure, $Pr[b_1, b_2]$ means the probability that two *simultaneously consumed* tokens from the streams b_1 and b_2 are true. Similarly, $Pr[\overline{b_1}, \overline{b_2}]$ means the probability that they are both false, and $Pr[b_2 | b_1]$ means the probability that a token b_2 is true given that the *simultaneously consumed* token b_1 is true. These relationships must be applied with care because

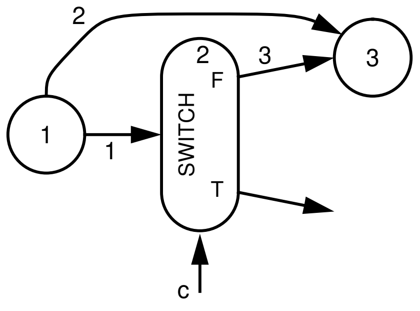

Figure 9: A dataflow graph that is consistent if and only if c is always false.

$$\Gamma(\vec{p}) = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & (1-p_1) & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & (p_2-1) & 0 & 0 \\ 0 & p_1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -p_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} . \tag{9}$$

Here, $\vec{p}$ is a vector with all Boolean probabilities. Analogous to (4), we need to find a vector $\vec{r}(\vec{p})$ such that

$$\Gamma(\vec{p})\,\vec{r}(\vec{p}) = \vec{o} \tag{10}$$

$\Gamma(\vec{p})$ must have rank 6 (or equivalently, its null space must have rank 1). It is easy to verify that it has rank 6 if and only if

$$p_1 = p_2, \tag{11}$$

which fortunately is true trivially, since b_1 and b_2 emanate from the same actor. A suitable vector in the nullspace of $\Gamma(\vec{p})$ is given by

$$\vec{r}(\vec{p}) = \begin{bmatrix} 1 & 1 & (1-p_1) & p_1 & 1 & 1 & 1 \end{bmatrix}^T . \tag{12}$$

Notice that this vector works for any numerical value of p_1 and p_2. One interpretation for this is that, on average, for every firing of actor 1, actor 3 will fire $(1-p_1)$ times, and actor 4 will fire p_1 times. This if intuitive. Since p_1 is not an integer, in general, it makes little sense to talk of finding the smallest integer vector in the nullspace of $\Gamma(\vec{p})$. However, the information gained from any solution to (10) is useful. It tells us (a) that a solution exists, and (b) in what proportion the actors in the actors in the system should fire as functions of the Boolean proportions in the system. We will strive for a schedule such that $\vec{r}(\vec{p})/\|\vec{r}(\vec{p})\|$ is a vector giving the probability that each actor will fire given that one actor fires. Furthermore, we will extend this model below in such a way that for most graphs we can find a similar vector that always has integer values and can be used to construct compile-time schedules.

3.2. Consistency

In [Lee91], the above *token-flow* model is used to define *consistency* for dynamic dataflow graphs. Two kinds of consistency are identified, *strong* and *weak*. Strong consistency means that there exists a vector $\vec{r}(\vec{p})$ such that (10) is satisfied for any allowed value of the Boolean proportions $\vec{p}$ in the system. Weak consistency means that (10) is satisfied only for certain values of the Boolean proportions $\vec{p}$. The if-then-else of figure 8 is consistent subject to the condition in (11). Since this condition is always satisfied, the dataflow graph is strongly consistent.

However, not all dataflow graphs are strongly consistent. Consider the example in figure 9. The topology matrix for this system is well-defined finite subsequence (called a *complete cycle*) of the infinite stream b_i. Furthermore, we will see that we never need to know or estimate the numerical value of p_i. All manipulations that use it can use it symbolically. For the initial development, for clarity and intuition, we will assume the probabilistic interpretation. The complete cycle interpretation will follow.

Non-SDF actors other than SWITCH and SELECT can be modeled as well, as long as they have representations like that in figure 7. This even includes non-determinate actors as in [Kos78] (see [Lee91]). In this paper, however, we will use only SWITCH and SELECT in our examples because they are traditional and familiar in the dataflow literature.

Consider the if-then-else program shown in figure 8. The SWITCH directs the incoming token to one of two subsystems, 3 or 4, depending on the Boolean supplied by actor 7. Since the Boolean is supplied to two actors, it is implicitly forked into two Boolean streams labeled b_1 and b_2. It will be important to recognize these as two Boolean streams. In this figure, the numbers adjacent to the arcs will be used only to identify them.

Assuming that all unmarked actors produce and consume a single token when they fire, the topology matrix for this system is given by:

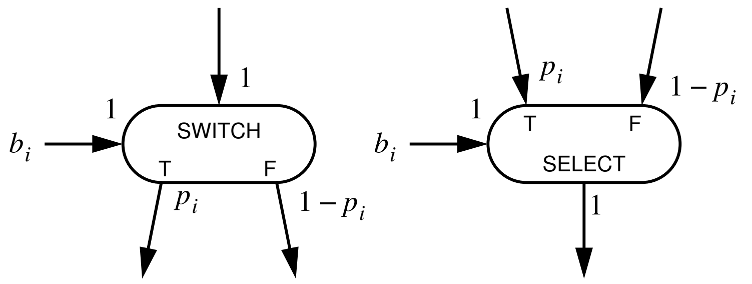

Figure 7: Dynamic dataflow actors annotated with the expected number of tokens produced or consumed per firing as a function of p_i, the probability that a token from the stream b_i is TRUE.

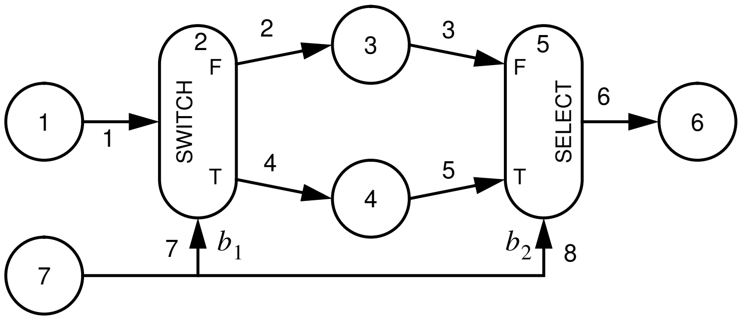

Figure 8: An if-then-else dataflow graph. The numbers adjacent to the arcs merely identify them.

actors [Den75], also used in [Wen75], [Tur81], and [Pin85], and are the same as the distributor and selector in [Dav82]. Neither actor is SDF because the number of tokens consumed or produced is not fixed. It depends on the Boolean control input. In the case of the SWITCH, the firing of downstream actors will be data-dependent. In the case of the SELECT actor, the firing of upstream actors will be data-dependent. In both cases, run-time control is required.

The characterization of actors for SDF graphs can be extended to graphs containing SWITCH and SELECT actors. The number of inputs consumed and produced must now be represented as a function of Booleans in the system.

3.1. The Token Flow Model

Loosely speaking, the balance equations require that in the long run, the number of tokens produced on an arc must equal the number of tokens consumed. This can be checked by relating the average number of tokens consumed and produced to the average number of TRUEs in the Boolean streams in the system. This is shown for the SWITCH and SELECT actors in figure 7. Loosely, p_i is the long term proportion of TRUEs in the Boolean stream b_i, which supplies the control inputs.

Several rigorous interpretations of p_i in figure 7 are possible. The most general interpretation of p_i is that it is a formal placeholder for an unknown quantity that determines number of tokens produced or consumed. In a probabilistic formulation, we model the Boolean stream b_i as a random process, and p_i is the marginal probability that a token from b_i is TRUE. This interpretation requires that the marginal probability be constant across firings. Equivalently, b_i is stationary is the mean. For practical dataflow graphs, this requirement is overly restrictive. Fortunately, for most dataflow graphs, p_i can be interpreted as the proportion of TRUE tokens in a

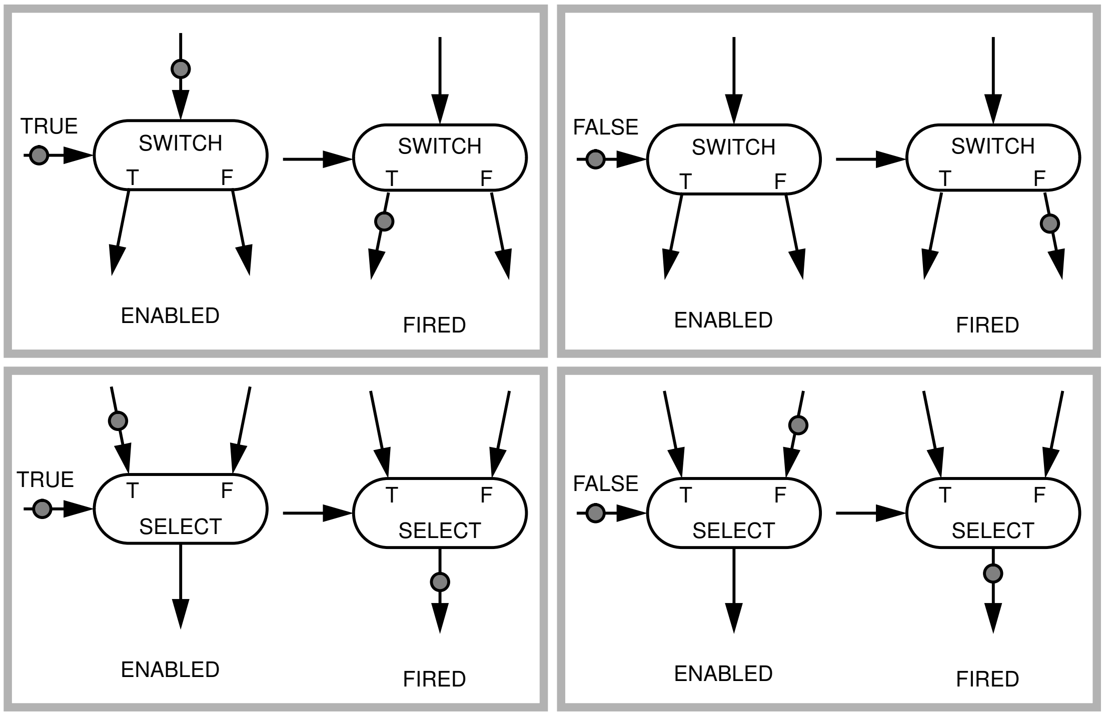

Figure 6: Dynamic dataflow actors consume and produce tokens depending on Booleans in the system.

invocations can be assembled into an acyclic precedence graph (APG). The procedure for doing this is straightforward; it consists of simulating a dynamically scheduled run until each actor i has been added to the APG r_i times. When an actor becomes enabled, an invocation of that actor is added to the precedence graph, and precedence arc connections are made to previous actors whose firings supplied the enabling data. Initially, only source actors (which have no input paths) and actors that have enough delays on their input paths are enabled. But once an invocation is added to the precedence graph, a data structure is updated to indicate the data that would be produced by a firing of that actor. This extra data, in turn, enables other actors.

The method for constructing the APG can be described precisely. Let $\vec{b}(n)$ be a vector denoting the number of tokens in each arc at step n of the procedure. Hence $\vec{b}(0)$ denotes the initial state of the graph, indicating which arcs contain how many delays. Suppose that at the initial state a certain actor i is enabled. Then an invocation of that actor can be put in the APG with no ancestors, indicating that there are no precedents to be honored. The state of the system is then updated by constructing a vector $\vec{q}(n)$ with a 1 in the i^{th} position, and zeros everywhere else, and writing

$$\vec{b}(n+1) = \vec{b}(n) + \Gamma\vec{q}(n)\,. \qquad \textbf{(6)}$$

By only letting $\vec{q}(n)$ indicate enabled actors, $\vec{b}(n)$ is guaranteed to always have non-negative entries.

The process is repeated with the new system state. Whenever a new actor is added to the APG, arcs are created to indicate precedences. No actor i will be added to the APG more than r_i times, and the graph will be declared complete when

$$\sum_{n=0}^{N} \vec{q}(n) = \vec{r}. \qquad \textbf{(7)}$$

Notice that

$$\vec{b}(N) = \vec{b}(0) + \Gamma\vec{r} = \vec{b}(0)\,, \qquad \textbf{(8)}$$

indicating that the state of the arcs after performing all invocations in the APG will be the same as it was before.

Once the APG is constructed, there are many scheduling alternatives. For single processor targets, some reasonable scheduling objectives might include minimization of data or program memory requirements. For multiprocessor targets, minimizing makespan, or maximizing flow-time are more likely objectives. For some examples of scheduling heuristics that have been applied, see [Sih91].

3. Dynamic Dataflow

Although SDF is adequate for representing large parts of many algorithms, it is rarely sufficient for expressing an entire program. A more general dataflow model is needed in order to express data-dependent iteration, conditionals, and recursion. The addition of two actors, SWITCH and SELECT, enables all of these except recursion (for a discussion of recursion see [Lee91]). The behavior of these actors is shown in figure 6. These are minor variations of the original Dennis where $\vec{o}$ is a vector full of zeros, and $\vec{r}$ is the *repetition vector* containing the r_i for each actor. Printz calls (4) the "balance equations" [Pri91]. For the topology matrix in (9), one solution is

$$\vec{r} = \begin{bmatrix} 1 \\ 10 \\ 100 \\ 10 \\ 1 \end{bmatrix} . \tag{5}$$

In fact, this solution is the smallest one with integer entries.

For a connected SDF graph, it is shown in [Lee87a] that a necessary condition to be able to construct an admissible periodic schedule is that the rank of Γ be equal to one less than the number of actors in the graph. This means that the null space of Γ has dimension one. From (4) we see that $\vec{r}$ must lie in the null space of Γ. It is also shown in [Lee87a] that when the rank is correct, there always exists a vector that contains only integers and lies in this null space. Solving for the smallest such vector is a simple procedure that involves finding a fractional solution and applying Euclid's algorithm to find the least common multiple of all the denominators. This technique has been implemented in the Gabriel [Bie90] and Ptolemy [Buc91] systems, and it is simple and fast. Furthermore, the technique can be re-targeted to a wide variety of architectures, including arrays of processors.

In order to construct a periodic schedule we must rule out the possibility of deadlock [Lee87a]. In this paper, we will simply assume the graphs we work with do not deadlock.

2.4. Inconsistency

For some SDF graphs, (4) has no solution. An example is shown in figure 5. The error here is that actor 3 expects two streams with the same rate of token flow, and it is getting two streams with different rates of flow. There is no schedule that can repeatedly run this graph with bounded memory.

2.5. Disconnected Graphs

For disconnected graphs, if each graph is itself consistent, then the null space of Γ has dimension equal to the number of disconnected graphs. In this case, the scheduling problem is easily partitioned into smaller problems each of which involves only a connected subgraph.

2.6. Static Scheduling

Scheduling an SDF graph begins with solving for the smallest integer vector $\vec{r}$ in the null space of the topology matrix. This vector, as illustrated in (5), indicates how many times each actor should be invoked in one cycle of a periodic schedule. Given this set of numbers, the required

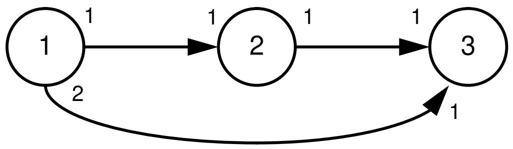

Figure 5: An inconsistent SDF graph.

These tokens can be viewed as causing an offset between the tokens produced and the tokens consumed on the arc. Hence, such an initial token is called a unit delay. A delay can enable the firing a downstream actor. For instance, at the start of execution, actor 2 in figure 3 is enabled.

2.3. Topology matrix and repetitions vector

Because of the predictable production and consumption of tokens, SDF graphs can be scheduled statically, at compile time. When the graph is going to be repeatedly executed, this can result in considerable speedup or simpler hardware, compared to run-time scheduling. For signal processing applications the graph is typically repeatedly executed a very large number of times on a continuous stream of input data. If the application has real-time constraints, then the speedup achieved by scheduling at compile time can be critical.

When an SDF graph is to be executed repeatedly, the compiler should construct just one cycle of a periodic schedule. The first step is to determine how many invocations of each actor should be included in each cycle. This can be determined using information about the number of samples consumed and produced. Consider the connection of three actors shown in figure 4. Let I_i denote the number of tokens consumed by the i^{th} actor, and $O_{i\mathrm{i}}$ denote the number of tokens produced by the i^{th} actor, as shown in the figure. Let r_i denote the number of times the i^{th} actor is repeated in the each cycle of the iterated schedule. Then it must be true that

$$r_1 O_1 = r_2 I_2 \tag{1}$$

$$r_2 O_2 = r_3 I_3 \tag{2}$$

These two equations ensure that the number of tokens produced on each arc is equal to the number consumed on that arc in each cycle of the iterated schedule. Indeed, the first step in finding a schedule for an SDF graph is to solve a set of such equations, one for each arc in the graph, for the unknowns r_i.

These equations can be written concisely by constructing a *topology matrix* Γ that contains the integer O_i in position (j, i) if the i^{th} actor produces O_i tokens on the j^{th} arc. It also contains the integer $-I_i$ in position (j, i) if the i^{th} actor consumes I_i tokens from the j^{th} arc. For example, the nested iteration shown in figure 2 has the following topology matrix

$$\Gamma = \begin{bmatrix} 10 & -1 & 0 & 0 & 0 \\ 0 & 10 & -1 & 0 & 0 \\ 0 & 0 & 1 & -10 & 0 \\ 0 & 0 & 0 & 1 & -10 \end{bmatrix}. \tag{3}$$

Then the system of equations to be solved is

$$\Gamma \vec{r} = \vec{o} \tag{4}$$

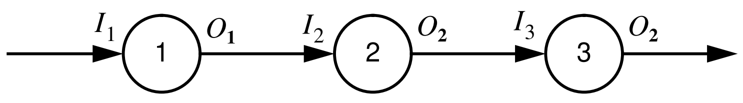

Figure 4: Three actors annotated with the number of tokens consumed and produced on each firing.

2. Synchronous Dataflow

Synchronous dataflow (SDF) is a special case of dataflow where the number of tokens consumed or produced by a given actor when it fires is fixed and known at compile time. In repeated firing of the same actor, the same behavior will be repeated. Figure 1 shows some SDF actors. In the left-most box, the actor has two operands, and the presence of one token on each input enables the firing of the actor. When it fires, it produces one token on its output. In the middle, a single token enables the firing, but multiple tokens (two shown) are produced. The right-most actor is enabled by multiple tokens, and produces a single token when it fires.

2.1. Manifest iteration

Consider the dataflow graph in figure 2. The numbers adjacent to the inputs and outputs of the actors indicate how many tokens are consumed and produced each time the actor fires. Consequently, the third actor fires ten times for every firing of the second, which in turn fires ten times for every firing of the first. There is nothing in this model to prevent simultaneous firing of successive invocations of the same actor, so this schema solves the first open problem listed by Dennis in [Den75], providing the semantics of a "parallel for" in dataflow. The iteration is specified in a truly data-driven way. Of course, if we restrict the dataflow actors to be SDF, then the iteration can only be manifest, where the number of cycles of the iteration is known at compile-time.

2.2. Delays

A dataflow graph may contain initial tokens on an arc. These will be indicated in this paper with a diamond, as shown in figure 3.

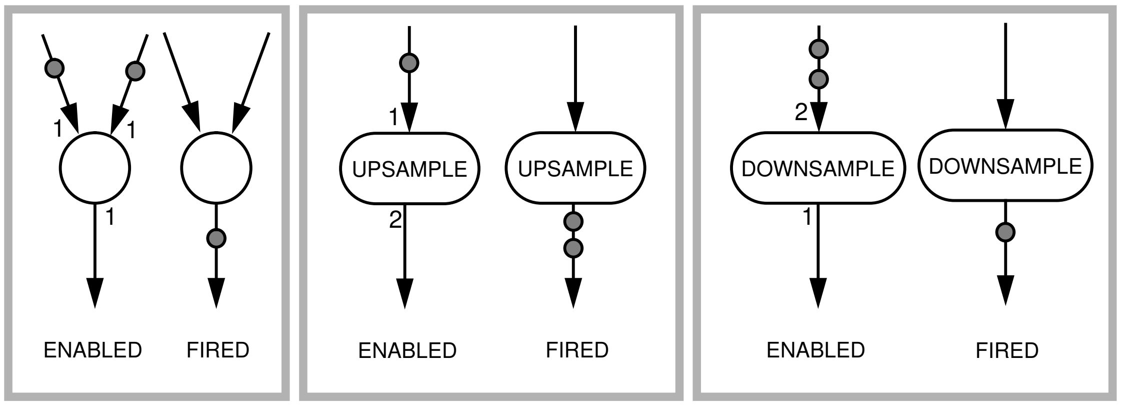

Figure 1: Synchronous dataflow actors consume and produce fixed numbers of tokens.

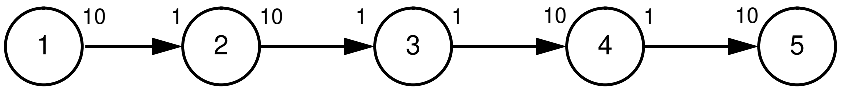

Figure 2: A nested iteration expressed as an SDF graph.

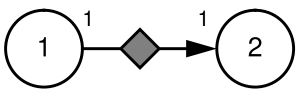

Figure 3: A unit delay represents an initial token on an arc.

A dataflow graph is a directed graph with *actors* at the nodes. The arcs represent the flow of data. Although there are many variations in implementation, conceptually arcs behave as FIFO queues, sometimes with limited capacity, sometimes not. While tagged tokens [Arv82] achieve behavior equivalent to a FIFO queue, they do not require that tokens be produced or consumed in order. Tokens are tagged with context information so that their conceptual position in the queue is maintained independent of the order in which they are processed. We will assume in this paper that FIFO behavior is enforced on the arcs. This can lead to simpler run-time implementation than a tagged-token schema.

Dataflow graphs have data-driven semantics. It is the availability of operands that enables an operator. Hence, sequencing constraints follow only from data availability. Although this feature is useful, it has its limitations. Virtually all programming languages provide primitives for directly specifying control flow, such as the familiar if-then-else, do-while, goto, and for loops. A purely data-driven language provides no such primitives, relying instead on actors that direct the flow of tokens. We are so familiar with these control flow constructs that expressing equivalent functionality using data-driven semantics can be awkward. If this awkwardness stems primarily from lack of experience, however, then there is much to be gained by exploring languages that have purely data-driven semantics at the source code level.

Such languages are rare. Even so called "dataflow" languages, such as Val [Mcg82] and ID [Nik88], and functional languages such as DFL (Silage) [Hil89] contain familiar "for" loops and "if-then-else" constructs. These are sometimes translated into a purely data-driven internal representation, such as the machine code for a dataflow machine. In signal processing, however, purely data-driven semantics are attractive. Signal processing is special in that dataflow graphs are a natural representation for algorithms *even in the absence of other practical motivations.*

The intent of this paper is not to propose a particular language design, nor to advocate a particular choice of actors on which to base dataflow graphs. It is instead to develop an analytical technique that applies to a class of languages in which operators are described in terms of consumption and production of tokens. The key property of these languages is that operators and functions operate on streams of tokens. Although we will use a graphical syntax to illustrate example programs, the method is not restricted to graphical languages. For some applications, particularly signal processing, graphical syntax has proven attractive (see [Bie90] for one example and references to dozens of others). For others it has not, and textual functional languages are more attractive. To date, however, very few experiments have been done with graphical syntax of the type discussed in this paper. A textual syntax with the same or related semantics could easily be created, but it would demand more of the reader without adding materially to the exposition in this paper.

The method in this paper is inspired by the algebraic techniques of Benveniste, et. al. [Ben88], [Ben90a/b], [LeG90], which can accomplish some of the same objectives for a different class of languages. Fortunately, for dataflow semantics our method is usually simpler than an adaptation to dataflow of that in [Ben88].

This paper begins with a brief review of SDF, emphasizing the analytical properties that can be used for static scheduling. The analysis and scheduling methodology for dynamic dataflow will follow the same pattern, but computations will be symbolic rather than numeric. The token flow model as presented in [Lee91] is reviewed and then extended to model cyclic firings, rather than just long term averages. Consistency and strong consistency are defined and analytical methods are given for checking for them. "Complete cycles" are then defined, enabling the systematic construction of "annotated schedules", or static schedules annotated with the Boolean conditions under which actors fire.

THE TOKEN FLOW MODEL

Joseph Buck and Edward A. Lee

Dept. of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720
internet address: jbuck@EECS.Berkeley.EDU, eal@EECS.Berkeley.EDU

(presented at Data Flow Workshop, Hamilton Island, Australia, May 1992)

Abstract

This paper reviews and extends an analytical model for the behavior of dataflow graphs with data-dependent control flow. The number of tokens produced or consumed by each actor is given as a symbolic function of the Booleans in the system. Long-term averages can be analyzed to determine consistency of token flow rates. Short-term behavior can be analyzed to construct an annotated schedule, or a static schedule that annotates each firing of an actor with the Boolean conditions under which that firing occurs. Necessary and sufficient conditions for bounded-length schedules, as well as sufficient conditions for determining that a dataflow graph can be scheduled in bounded memory are given. Annotated schedules can be used to generate efficient implementations of the algorithms described by the dataflow graphs.

1. Introduction

The principal strength of dataflow graphs is that they do not over-specify an algorithm by imposing unnecessary sequencing constraints between operators. Instead, they specify a partial order, where sequencing constraints are imposed only by data precedences. Since the representation does not over-constrain the order of operations, a scheduler has the freedom it needs to adequately exploit deep pipelines, to maximize re-use of limited hardware resources, or to exploit parallel processing units. To get these benefits, compilers for pipelined or parallel machines often heavily rely on dataflow analysis. The most successful compilers either address only the modest parallelism in pipelined and superscalar machines or restrict themselves to domain-specific areas such as signal processing. In both cases, only dataflow graphs with deterministic control flow have been handled. The general class of such graphs is called *synchronous dataflow (SDF)* [Lee87a]. However, the class of applications that fit this model is too restricted. In compilers, the SDF model constrains the dataflow analysis to lie within so-called "basic blocks", regions of code delimited by branches or branch destinations. In parallelizing schedulers, it limits the use of dataflow techniques to overly specialized domains with restricted run-time decision making. The model must be broadened.

The authors gratefully acknowledge the support of DARPA (J-FBI 90-073), NSF (MIP-8657523), and The Sony Corporation.