Synchronous Data Flow

EDWARD A. LEE, MEMBER, IEEE, AND DAVID G. MESSERSCHMITT, FELLOW, IEEE

Data flow is a natural paradigm for describing DSP applications for concurrent implementation on parallel hardware. Data flow programs for signal processing are directed graphs where each node represents a function and each arc represents a signal path. Synchronous data flow (SDF) *is a special case of data flow (either atomic or large grain) in which the number of data samples produced or consumed by each node on each invocation is specified* a priori. *Nodes can be scheduled statically (at compile time) onto single or parallel programmable processors so the run-time overhead usually associated with data flow evaporates. Multiple sample rates within the same system are easily and naturally handled. Conditions for correctness of SDF graph are explained and scheduling algorithms are described for homogeneous parallel processors sharing memory. A preliminary SDF software system for automatically generating assembly language code for DSP microcomputers is described. Two new efficiency techniques are introduced,* static buffering *and an extension to SDF to efficiently implement conditionals.*

I. Data Flow and Synchronous Data Flow: An Introduction

For concurrent implementation, a signal processing task is broken into subtasks which are then automatically, semiautomatically, or manually scheduled onto parallel processors, either at compile time (*statically*) or at run-time (*dynamically*). Automatic breakdown of an ordinary sequential computer program is an appealing concept [1], but the success of techniques based on traditional imperative programming is limited; imperative programs do not often exhibit the concurrency available in the algorithm. If the programmer provides the breakdown as a natural consequence of the programming methodology, we should expect more efficient use of concurrent resources.

Synchronous data flow (*SDF*) is a special case of data flow [2]–[6], a hardware and software methodology popular among computer scientists for parallel computation. Under the data flow paradigm, algorithms are described as directed graphs where the nodes represent computations (or functions) and the arcs represent data paths. A second-order recursive digital filter described as a data flow graph is shown in Fig. 1. In that example, an essentially infinite stream of input data is expected, so the nodes specify computations performed infinitely often. This is typical of signal processing applications, and is an important property often lacking in more general applications.

Manuscript received August 29, 1986; revised March 13, 1987. This research was sponsored in part by an IBM faculty development grant and by the National Science Foundation under Grant ECS-8211071.

The authors are with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, USA.

IEEE Log Number 8716426.

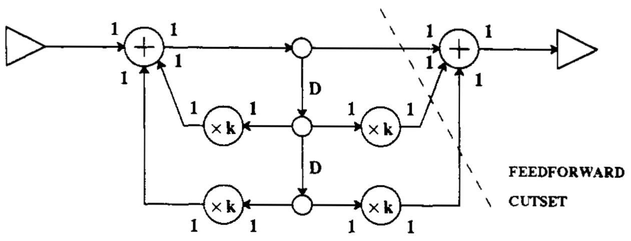


Fig. 1. A data flow graph for a second-order recursive digital filter. The empty circles are "fork" nodes, which simply replicate each input sample on all output paths. The "*D*" on two of the arcs indicates delay, and the "1" adjacent to each node input or output indicates that a single token is consumed or produced when the node fires.

The data flow principle is that any node can *fire* (perform its computation) whenever input data are available on its incoming arcs. A node with no input arcs may fire at any time. This implies that many nodes may fire simultaneously, hence the concurrency. Because the program execution is controlled by the availability of data, data flow programs are said to be *data-driven* [7]. To preserve the integrity of the computation, nodes must be free of *side effects*. For example, a node may not write to a memory location which is later read by another node unless the two are explicitly connected by an arc. The only influence one node has on another is the data passing through the arcs.

In Fig. 1, each input and output of each node has the number "1" adjacent to it, which in our notation indicates that when the node fires, a single sample (or *token*) will be consumed or produced on each arc. A synchronous data flow graph is one for which these numbers may be specified for every node *a priori*. That is, the number of tokens produced or consumed must be independent of the data. We expect that most nodes for signal processing applications will be synchronous, and we can take advantage of this dominant synchrony. Asynchronous nodes, as shown in Fig. 2, can be used to get conditional execution of a subgraph of a data flow graph. It is equivalent to the functional statement

$$z = \text{if } (x) \text{ then } f(y) \text{ else } g(y).$$

The asynchronous nodes are the *switch* and *select* nodes. Although describing an important capability, the graph in

0018-9219/87/0900-1235$01.00 © 1987 IEEE

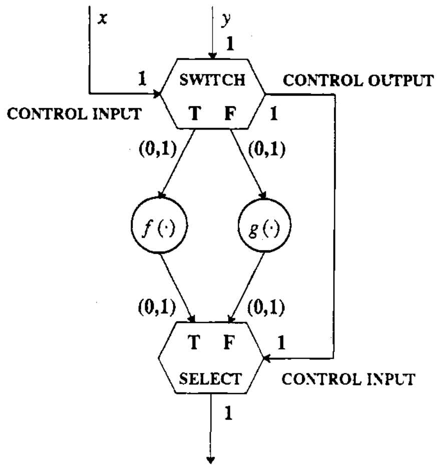


Fig. 2. Two asynchronous nodes are shown used in a conditional construct analogous to an if-then-else. The *switch* routes tokens to one of two output paths depending on a control input and the *select* selects tokens from one of two inputs depending on the control token. The notation "(0, 1)" indicates that when the node fires, either zero or one sample will be produced or consumed.

Fig. 2 is not an SDF graph; nonetheless, we show later (Section V) that it can be handled efficiently in at least one application, code generation for DSP microcomputers.

We call the graph in Fig. 1 a *homogeneous* SDF graph, to indicate that all nodes produce or consume a single sample on each input or output arc in each invocation. It may be implemented by constructing a schedule that invokes each node once and then repeats indefinitely. To understand how to construct such a schedule we need to understand precisely the meaning of the delay "*D*" on two of the arcs.

The term *delay* is used in the signal processing sense, corresponding to a sample offset between the input and the output. We define a *unit delay* on an arc from node *A* to node *B* to mean that the *n*th sample consumed by *B* will be the $(n - 1)$th sample produced by *A*. This implies that the first sample *B* consumes is not produced by *A* at all, but is part of the initial state of the arc buffer. Indeed, a delay of *d* samples on an arc is implemented in our model simply by initializing the arc buffer with *d* zero samples. The inscription *dD* will be placed near the arc to illustrate the delay.

Nodes cannot fire in an arbitrary order. The precedences between invocations of the nodes are illustrated in Fig. 3.

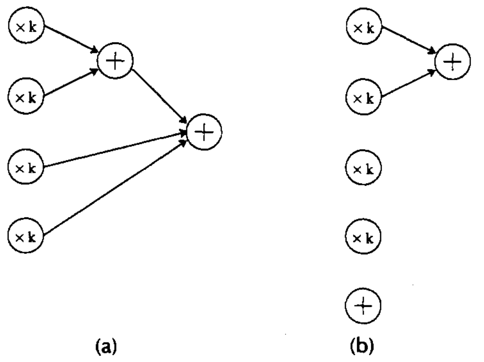


Fig. 3. Precedence graphs for one cycle of a blocked periodic schedule for the SDF graph of Fig. 1. The graph in (b) assumes that unit delays have been put on the feed-forward cutset shown with a dotted line in Fig. 1. The "fork" nodes are assumed to run in zero time, and so are omitted.

A *schedule* determines when and where (on which processor) nodes fire. A *blocked* schedule is a periodic schedule where each cycle terminates before the next cycle begins. The SDF graph of Fig. 1 can be implemented with any blocked schedule for which each cycle satisfies the precedences of Fig. 3. (For a different approach that permits overlapped cycles, see [8]–[11].) Blocked schedules for three processors that satisfy the precedences of Fig. 3 are illustrated in Fig. 4. For the moment we are ignoring interprocessor communication time, and we are boldly assuming that the "fork" nodes have zero execution time. Also, we assume that a single multiplication takes as long as an addition with three inputs.

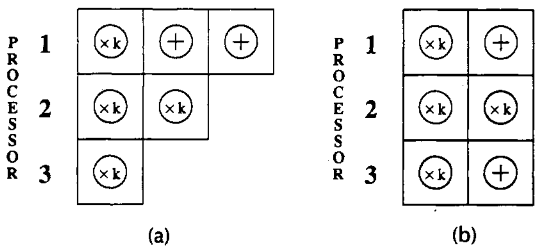


Fig. 4. One cycle of blocked schedules for three processors; the schedules satisfy the precedences of Fig. 3 without (a) and with (b) pipelining. A multiplication is assumed to take the same amount of time as a three-input addition. Again, the "forks" are assumed to take zero time.

In Fig. 3(b), we have assumed that unit delays have been put on the arcs in the cutset marked with a dotted line in Fig. 1. These delays affect only the first output sample. For computations that repeat infinitely often, there is very little penalty for putting delays on *feed-forward* cutsets. This is called *pipelining*, and often results in multiprocessor schedules with greatly enhanced computation rates (*throughput*). The schedules shown are *static* becuse they are generated at compile time and do not vary at run-time. Static scheduling is possible for all correctly constructed SDF graphs (more about this later).

For this example, the *iteration period* is the length of one cycle of a blocked schedule and is the reciprocal of the throughput. For many applications we will try to minimize the iteration period. The iteration period is bounded from below because of directed loops. The *iteration period bound* for homogeneous SDF graphs has been shown to be the worst case (over all directed loops) of the total computation time in a loop divided by the number of delays in the loop [9], [12]. This assumes that pipelining is acceptable. The iteration period bound for Fig. 1 is two, and is met by the schedule in Fig. 4(b).

We have ignored interprocessor communication time, as well as any other overhead associated with invoking a node. Although the overhead associated with a static schedule is likely to be less than a dynamic schedule, it is likely to be nontrivial, particularly for parallel implementations, which have interprocessor communication time. (Efficient single processor implementations can be generated as shown below.) The nodes in Fig. 1 are very simple, most likely elementary indivisible operations in a programmable processor. Such SDF graphs are said to be *atomic* (from the Greek *atomos* meaning indivisible). Since the execution time of each node is short, any overhead associated with each node invocation will add up to a substantial portion of the com-

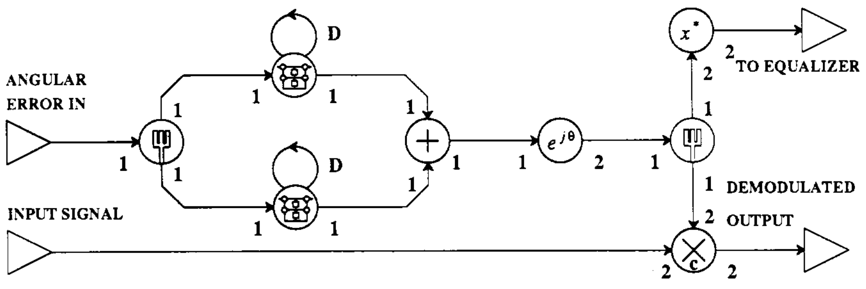


Fig. 5. A large-grain SDF graph describing part of the computation in a voice-band data modem (the PLL in Fig. 6); it uses two second-order recursive digital filters. Several of the signals in this graph are complex, implemented as the real part followed by the imaginary part. For example, the node marked $e^{j\theta}$ computes the cosine and sine of the input and outputs them as real and imaginary parts, respectively. A complex signal has twice the sample rate of a comparable real signal, therefore. Note that a multiplicity of sample rates is easily handled under the SDF paradigm. The node marked x^* computes the complex conjugate of its input, and the multiplier is a complex multiplier.

putation time. A larger granularity would reduce the overhead. The SDF graph of Fig. 5 has two nodes which are second-order recursive filters. The second-order filters may themselves be implemented using the SDF graph of Fig. 1, or they may be considered elementary operations, implemented, for example, in assembly language. If the granularity is at the level of signal processing subsystems (second-order sections, FIR filters, LMS adaptive filters, trigonometric functions, etc.), the paradigm is called *large grain data flow* (LGDF) [6], [13]–[15]. The term SDF applies to both atomic and large grain synchronous data flow.

Since a recursive digital filter stores state variables, *self-loops* are required, as shown in Fig. 5. The computation within the node can, therefore, be free of side effects, with its state being fed back over the self-loop. The graph in Fig. 5 represents the computation in the PLL node of the voice-band data modem illustrated in Fig. 6. It describes an implementation of a 2400-bit/s, 600-Bd, frequency-division-multiplexed, full-duplex data modem with band-splitting filters and a fractionally spaced passband adaptive equalizer [16]–[18].

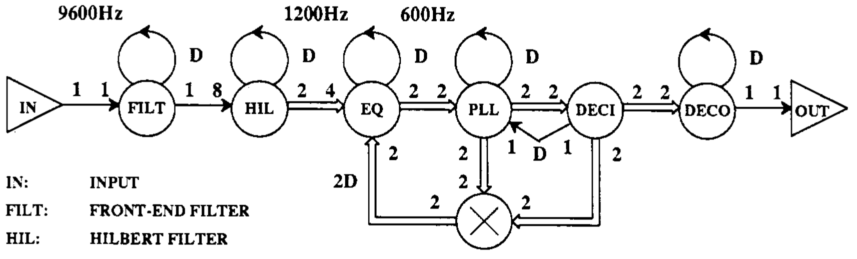
Hierarchical graphical descriptions of applications are extremely appealing. A designer could begin with a top-level construction like that of Fig. 6, and develop the details by building graphs such as Fig. 5. Once all the nodes are defined, a static schedule can be generated, and a parallel implementation deployed. An experimental graphical interface for such construction is described by Hait [19].

The SDF paradigm permits multiple sample rates in the same system. This is a significant departure from previous models, and is an important consideration for most practical DSP systems. The modem in Fig. 6 explicitly shows three sample rates, as well as both real and complex signal paths. Complex signals are implemented as alternating real and imaginary parts on the same arc. Systems where all sample rates are rational multiples of all other sample rates are called *synchronous* in the signal processing literature. Synchronous DSP systems are easily described using the SDF paradigm, hence the name for the paradigm.

The above introduction has skirted several important issues. It has implied that static scheduling is possible for SDF graphs, so that the overhead usually associated with the dynamic scheduling of data flow graphs evaporates. But it is possible for SDF graphs to be incorrectly constructed, in which case static schedules cannot be found. After reviewing some related work, we describe an analysis technique for SDF graphs which yields necessary and sufficient conditions for correctness of the graph and shows how to proceed with the scheduling. We then use a simple scheduling algorithm to construct a parallel implementation of the voice-band data modem. An experimental software SDF programming system called Gabriel is briefly described.

II. Related Paradigms and Programs

Many so-called "block diagram languages" have been developed to permit programmers to describe signal pro-



IN:	INPUT
FILT:	FRONT-END FILTER
HIL:	HILBERT FILTER
EQ:	ADAPTIVE EQUALIZER
PLL:	PHASE LOCKED LOOP
DECI:	DECISION
DECO:	DECODER
OUT:	OUTPUT

Fig. 6. An SDF graph showing a voice-band data modem. Note the multiplicity of sample rates. For emphasis, signal paths that carry complex signals are shown with double lines, although these arcs are no different from the arcs carrying real signals.

cessing systems more naturally. Some examples are BLODI [20], PATSI [21], BLODIB [22], LOTUS [23], DARE [24], MITSYN [25], Circus [26], and TOPSIM [27]. These examples are mostly simulation programs, not intended for real-time, cost-sensitive implementation. They are based on updating the state of the system according to some time increment, and hence are time-driven. Simulators using data-driven dynamic control have also been developed [28]–[30], but the overhead of dynamic control is often excessive. SDF avoids these costs by static scheduling.

Signal flow graphs, special cases of SDF graphs, have been used to describe linear, single-sample-rate systems for static parallel implementation [31]. The scheduling method in [31] does not consider the periodic nature of a desired schedule, and therefore does not always identify the best schedules. It also does not support multiple sample rates. In spite of these deficiencies, this representation has been endorsed by others [32], [33]. Multiprocessor implementations of algorithms specified as homogeneous atomic SDF graphs have been explored in [8]–[11], where overlapped cycles of period schedules are permitted. This admirable work has some deficiencies in some applications, however. Primarily, it has no provision for multiple sample rates, thus restricting the range of applications. Also, the complexity of the scheduling technique may become unmanageable for some important target architectures. Finally, S. Y. Kung *et al.* are investigating the use of hierarchical flow graphs for VLSI array processors [34].

An immediate potential objection to the SDF paradigm is that general SDF graphs can be expressed as simpler homogeneous SDF graphs, where the number of samples produced and consumed is always unity, as shown in Fig. 7(a). This means that the prior art applicable to homogeneous SDF graphs can be applied to multiple-sample-rate SDF graphs [9], [31]–[33], [35], [36]. However, the transformation is not always simple, as illustrated in Fig. 7(b), and may require replicating some of the nodes. A systematic method for performing this transformation is described in [37] where the transformation is used to find the iteration period bound for SDF graphs.

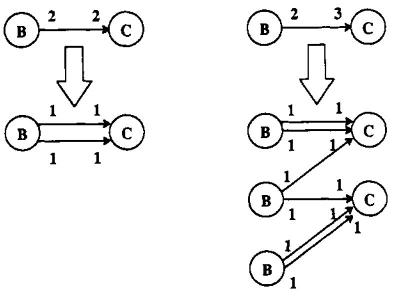


Fig. 7. Two transformations of general SDF graphs into homogeneous SDF graphs. The transformation is not always trivial.

Reduced dependence graphs are specifications of systems in terms of periodic acyclic precedence graphs, where only one period is illustrated, and its dependence on previous periods is done by indexing [38], [39]. The resulting description is similar to homogeneous data flow graphs. Reduced dependence graphs are used to describe *regular iterative algorithms*, which can then be systematically mapped onto processor arrays. This approach is suitable for descriptions of well-structured algorithms to be implemented in systolic arrays. The range of applications is again excessively limited for our objectives.

Computation graphs were introduced in 1966 by Karp and Miller [40] and were further explored by Reiter [41]. They are essentially equivalent to SDF graphs, but our use of the model differs significantly. Karp and Miller concentrate on fundamental theoretical considerations, for example, proving that computation graphs are *determinate*, meaning that any admissible execution yields the same result. Such a theorem, of course, also underlies the validity of data flow. Other early analysis using the general computation graph model concentrates on graphs that *terminate*, or deadlock, after some time. Most DSP applications, however, do not terminate, so these results are not useful in this application. Simplified versions of the model have been explored by Commoner and Holt [35] and Reiter [36], but the restrictions imposed on the model are excessive. Computation graphs have been shown to be a special case of *Petri nets* [42]–[44] or *vector addition systems* [45]. These more general models can be used to describe asynchronous systems, but implementations generally require expensive dynamic control.

III. Implementation Architectures

It is useful to consider the hardware which might be used to implement a computation described by an SDF graph. We consider three models,

- a sequential processor,
- isomorphic hardware mapping, and
- homogeneous parallel processors sharing memory without contention.

The first model cannot take advantage of available concurrency but does take advantage of the appealing programmer interface that is possible for SDF. In Section V we describe an experimental system for SDF programming of a single programmable DSP microcomputer.

The second model is a fancy name for an obvious implementation which has separate hardware for each node in the graph. In this case, SDF is a description of hardware as well as a description of the computation; the SDF graph is isomorphic with the implementation architecture. In some ways, SDF is useful as hardware description, but our intent is that SDF be used primarily for functional description. In this case, an isomorphic implementation architecture is not likely to be efficient, in that much of the hardware is likely to be idle much of the time.

The nodes of an SDF graph can be statically scheduled onto parallel processor architectures. One class of parallel architectures that we consider is homogeneous parallel processors sharing memory without contention. A practical programmable DSP of this type uses extensive pipelining with interleaved concurrent programs [46]. We are currently investigating SDF programming of multiple commercially available programmable DSPs. In this case, the processors will not share memory without contention, so we will have to consider communication delays in an attempt to optimize the schedule.

With these target architectures in mind, we now consider SDF in more detail.

IV. Verifying Correctness and Scheduling

We assume that an SDF graph describes a repetitive computation to be performed on an infinite stream of input data, so the desired schedule is periodic. It is not always possible to construct a practical periodic schedule for an SDF graph, however. Consider the SDF graph of Fig. 8(a). To start the

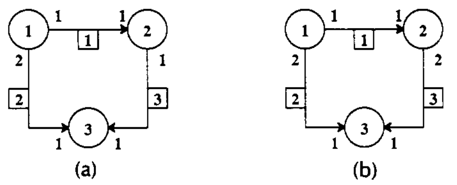


Fig. 8. (a) An example of a defective SDF graph with sample rate inconsistencies. (b) A corrected SDF graph with consistent sample rates. The flags attached to the arcs simply identify them with a number.

computation, node 1 can be invoked because it has no input arcs and hence needs no data samples. After invoking node 1, node 2 can be invoked, after which node 3 can be invoked. This sequence can be repeated. But node 1 produces twice as many samples on arc 2 as node 3 consumes. An infinite repetition of this schedule, therefore, causes an infinite accumulation of samples in the buffer associated with arc 3. This implies an unbounded memory requirement, which is clearly not practical.

In a DSP sense, the SDF graph has *inconsistent sample rates*. Node 3 expects as inputs two signals with the same sample rate but gets two signals with different sample rates. The SDF graph of Fig. 8(b) does not have this problem. A *periodic admissible sequential schedule* repeats the invocations {1, 2, 3, 3]. Node 3 is invoked twice as often as the other two. It is possible to automatically check for consistent sample rates and simultaneously determine the relative frequency with which each node must be invoked. To do this, we need a little formalism.

A. Formalism

An SDF graph can be characterized by a matrix similar to the incidence matrix associated with directed graphs in graph theory. It is constructed by first numbering each node and arc, as done in Fig. 8, and assigning a column to each node and a row to each arc. The (i, j)th entry in the matrix is the amount of data produced by node j on arc i each time it is invoked. If node j consumes data from arc i, the number is negative, and if it is not connected to arc i, then the number is zero. for the graphs in Fig. 8 we get

$$\Gamma_a = \begin{bmatrix} 1 & -1 & 0 \\ 2 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \qquad \Gamma_b = \begin{bmatrix} 1 & -1 & 0 \\ 2 & 0 & -1 \\ 0 & 2 & -1 \end{bmatrix}. \qquad (1)$$

This matrix is called a *topology matrix*, and need not be square, in general.

If a node has a connection to itself (a *self-loop*), then only one entry in Γ describes this link. This entry gives the net difference between the amount of data produced on this link and the amount consumed each time the node is invoked. This difference should clearly be zero for a correctly constructed graph, so the Γ entry describing a self-loop should be a zero row.

We can replace each arc with a FIFO queue (buffer) to pass data from one node to another. The size of the queue will vary at different times in the execution. Define the vector $\boldsymbol{b}(n)$ to contain the number of tokens in each queue at time n.

For the sequential (single processor) schedule, only one node can be invoked at a time, and for the purposes of scheduling it does not matter how long each node runs. Thus the time index n can simply be incremented each time a node finishes and a new node is begun. We specify the node invoked at time n with a vector $\boldsymbol{v}(n)$, which has a one in the position corresponding to the number of the node that is invoked at time n and zeros for each node that is not invoked. For the systems in Fig. 8, $\boldsymbol{v}(n)$ can take one of three values for a sequential schedule,

$$\boldsymbol{v}(n) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ OR } \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ OR } \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad (2)$$

depending on which of the three nodes is invoked. Each time a node is invoked, it will consume data from zero or more input arcs and produce data on zero or more output arcs. The change in the size of the queues caused by invoking a node is given by

$$\boldsymbol{b}(n + 1) = \boldsymbol{b}(n) + \Gamma \boldsymbol{v}(n). \qquad (3)$$

The topology matrix Γ characterizes the effect on the buffers of invoking a node.

To initialize the recursion (3) we set $\boldsymbol{b}(0)$ to reflect the number of delays on each arc. The initial condition for the queues in Fig. 9 is thus

$$\boldsymbol{b}(0) = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \qquad (4)$$

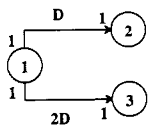


Fig. 9. An example of an SDF graph with delays on the arcs.

Because of these initial conditions, node 2 can be invoked once and node 3 twice before node 1 is invoked at all. Delays, therefore, affect the way the system starts up. Clearly, every directed loop must have at least one delay, or the system cannot be started.

Connections to the outside world are not considered, for now. Thus a node with only inputs from the outside is considered a node with no inputs, which can be scheduled at any time. The limitations of this approximation are discussed in [47].

B. Identifying Inconsistent Sample Rates

Inconsistent sample rates preclude construction of a periodic sequential schedule with bounded memory requirements. A necessary condition for the existence of such a schedule is that rank (Γ) = $s - 1$, where s is the number of nodes. This is proven in [47], so we merely give the intuition here. The topology matrix Γ_a for the graph in Fig.

8(a) has rank three, so no periodic admissible sequential schedule can be constructed. The topology matrix Γ_b for the graph in Fig. 8(b) has rank two, so a schedule can be constructed. It is proven in [47] that a topology matrix with the proper rank has a strictly positive (element-wise) integer vector $\boldsymbol{q}$ in its right nullspace, meaning that $\Gamma \boldsymbol{q}$ is the zero vector. For Fig. 8(b), a set of such vectors is

$$\boldsymbol{q} = J \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

for any positive integer J. Notice that the dimension of $\boldsymbol{q}$ is s, the number of nodes. Notice further that $\boldsymbol{q}$ specifies the number of times we should invoke each node in one cycle of a periodic schedule. Node 3 gets invoked twice as often as the other two nodes, for any positive integer J. That this works is proved using (3) by observing that if each node is invoked the number of times specified by $\boldsymbol{q}$, the amount of data $\boldsymbol{b}(n)$ left in each buffer ends up equal to the amount before the invocations. Hence, the schedule can be repeated infinitely often with finite memory.

Valuable information is obtained from the topology matrix. Its rank can be used to verify consistent sample rates, and its nullspace gives the relative frequency with which nodes must be invoked.

C. Insufficient Delays

Even with consistent sample rates, it may not be possible to construct a periodic admissible sequential schedule. Two examples of SDF graphs with consistent sample rates but no such schedules are shown in Fig. 10. Directed loops with

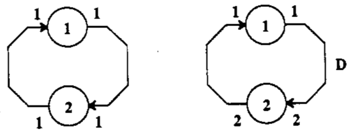


Fig. 10. Two SDF graphs with consistent sample rates but no admissible schedules.

insufficient delays are an error in the construction of the SDF graph and must be identified to the user. It is shown in [47] that a large class of scheduling algorithms will always run to completion if a periodic admissible sequential schedule exists, and will fail otherwise. Running such an algorithm is a simple way of verifying the correctness of the SDF graph. The class of algorithms is described in the next subsection.

D. Scheduling for a Single Processor

Given a positive integer vector $\boldsymbol{q}$ in the nullspace of Γ, one cycle of a periodic schedule invokes each node the number of times specified by $\boldsymbol{q}$. A sequential schedule can be constructed by selecting a *runnable* node, using (3) to determine its effect on the buffer sizes, and continuing until all nodes have been invoked the number of times given by $\boldsymbol{q}$. We define a class of algorithms.

Definition *(Class S Algorithms):* Given a positive integer vector $\boldsymbol{q}$ such that $\Gamma \boldsymbol{q} = 0$ and an initial state for the buffers $\boldsymbol{b}(0)$, the ith node is said to be *runnable* at a given time if it has not been run q_i times and running it will not cause a buffer size to become negative. A *class S algorithm* ("S" for Sequential) is any algorithm that schedules a node if it is runnable, updates $\boldsymbol{b}(n)$, and stops (*terminates*) only when no more nodes are runnable. If a class S algorithm terminates before it has scheduled each node the number of times specified in the $\boldsymbol{q}$ vector, then it is said to be *deadlocked.*

Class S algorithms construct static schedules by simulating the effects on the buffers of an actual run for one cycle of a periodic schedule. That is, the nodes need not actually run. Any dynamic (run-time) scheduling algorithm becomes a class S algorithm simply by specifying a stopping condition, which depends on the vector $\boldsymbol{q}$. It is proven in [47] that any class S algorithm will run to completion if a periodic admissible sequential schedule exists for a given SDF graph. Hence, successful completion of the algorithm guarantees that there are no directed loops with insufficient delay. A suitable class S algorithm for sequential scheduling is

1) Solve for the smallest positive integer vector $\boldsymbol{q} \in \eta(\Gamma)$.
2) Form an arbitrarily ordered list L of all nodes in the system.
3) For each $\alpha \in L$, schedule α if it is runnable, trying each node once.
4) If each node α has been scheduled q_α times, STOP.
5) If no node in L can be scheduled, indicate a deadlock (an error in the graph).
6) Else, go to 3 and repeat.

The only question remaining for single processor schedules is the complexity of the first step above. Our technique is simple. We begin with any node A in the graph and assume it will be run once in one cycle of the periodic schedule (i.e., let $q_A = 1$). Assume node B is connected to node A. We can find q_B with a simple division, possibly getting a fraction, but always getting a rational number. A node cannot be invoked a fractional number of times, so we will have to correct for this later. We do the same for any node C adjacent to B. A simple recursive algorithm computes these rational numbers in linear time (a linear function of the number of arcs, not the number of nodes). The resulting vector $\boldsymbol{q}$ has rational entries and is in the nullspace of Γ. To get the smallest *integer* vector in the nullspace of Γ we use Euclid's algorithm to find the least common multiple of all the denominators. Actually, three simultaneous objectives are accomplished with one pass through the graph. Sample rate consistency is checked, a vector (with rational entries) in the nullspace of Γ is found, and Euclid's algorithm is used to find the least common multiple of all the denominators.

SDF offers concrete advantages for single processor implementations. The ability to interconnect modular blocks of code (nodes) in a natural way could considerably ease the task of programming high-performance signal processors, even if the blocks of code themselves are programmed in assembly language. But a single processor implementation cannot take advantage of the explicit concurrency in an SDF description. The next section is dedicated to explaining how the concurrency in the description can be used to improve the throughput of a multiprocessor implementation.

E. Scheduling for Parallel Processors

Clearly, if a workable schedule for a single processor can be generated, then a workable schedule for a multiprocessor system can also be generated. Trivially, all the computation could be scheduled onto only one of the processors. Usually, however, the throughput can be increased substantially by distributing the load more evenly. It is shown in [47] that the multiprocessor scheduling problem can be reduced to a familiar problem in operations research for which good heuristic methods are available. We again give the intuition without the details. We assume for now homogeneous parallel processors sharing memory without contention, and consider only blocked schedules.

A blocked periodic admissible *parallel* schedule is a set of lists $\{\Psi_i; i = 1, \cdots, M\}$ where M is the number of processors, and Ψ_i specifies a periodic schedule for processor i. If $\boldsymbol{p}$ is the *smallest* positive integer vector in the nullspace of Γ then a cycle of a schedule must invoke each node the number of times given by $\boldsymbol{q} = J\boldsymbol{p}$ for some positive integer J. J is called the *blocking factor*, and for blocked schedules, there is sometimes a speed advantage to using J greater than unity. If the "best" blocking factor is known, then construction of a good parallel schedule is not hard.

The task of the scheduler is to construct a schedule that avoids deadlocks and minimizes the *iteration period*, defined more generally to be the run-time for one cycle of the schedule divided by J. The first step is to construct a graph describing the precedences in $\boldsymbol{q} = J\boldsymbol{p}$ invocations of each node. The graph will be acyclic. A precise class S algorithm accomplishing this construction is given in [47] so we merely illustrate it with the example in Fig. 11(a). Node 1 should be invoked twice as often as the other two nodes, so $\boldsymbol{p} = [2\ 1\ 1]^T$. Further, given the delays on two of the arcs, we note that there are three periodic admissible sequential schedules with unity blocking factor, $\phi_1 = \{1, 3, 1, 2\}$, $\phi_2 = \{3, 1, 1, 2\}$, or $\phi_1 = \{1, 1, 3, 2\}$. A schedule that is not admissible is $\phi_1 = \{2, 1, 3, 1,\}$, because node 2 is not immediately runnable. Fig. 11(b) shows the precedences involved

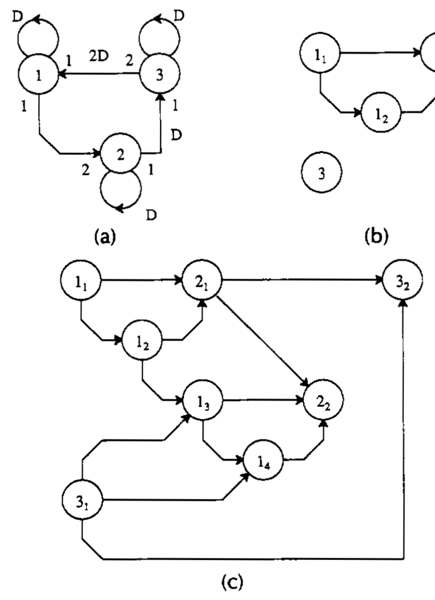


Fig. 11. (a) An SDF graph with self-loops. (b) An acyclic precedence graph for unity blocking factor, $J = 1$. (c) An acyclic precedence graph for $J = 2$.

in all three schedules. Fig. 11(c) shows the precedences using a blocking factor of two ($J = 2$).

The self-loops in Fig. 11(a) imply that successive invocations of the same node cannot overlap in time. Some practical SDF implementations have such precedences in order to preserve the integrity of the buffers between nodes. In other words, two processors accessing the same buffer at the same time may not be tolerable, depending on how the buffers are implemented. The self-loops are also required, of course, if the node has a *state* that is updated when it is invoked. We will henceforth assume that all nodes have self-loops, thus avoiding the potential implementation difficulties.

If we have two processors available, a schedule for $J = 1$ is

$$\Psi_1 = \{3\}$$
$$\Psi_2 = \{1, 1, 2\}.$$

When this system starts up, nodes 3 and 1 will run concurrently. The precise timing of the run depends on the run-time of the nodes. If we assume that the run-time of node 1 is a single time unit, the run-time of node 2 is two time units, and the run-time of node 3 is three time units, then the timing is shown in Fig. 12(a). The shaded region rep-

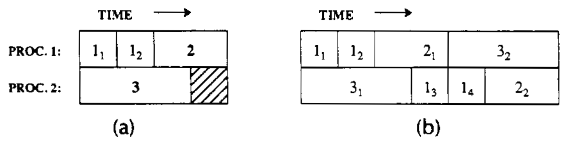


Fig. 12. One period of each of two periodic schedules for the SDF graph of Fig. 11. In (a) $J = 1$ while in (b) $J = 2$.

resents idle time. A schedule constructed for $J = 2$, using the precedence graph of Fig. 11(c) will perform better. An example is

$$\Psi_1 = \{3, 1, 3\}$$
$$\Psi_2 = \{1, 1, 2, 1, 2\}$$

and its timing is shown in Fig. 12(b). There is no idle time, so no faster schedule exists.

The construction of the acyclic precedence graph is handled by the class S algorithm given in [47]. The remaining problem of constructing a parallel schedule given an acyclic precedence graph is a familiar one. It is identical with assembly line problems in operations research, and can be solved for the optimal schedule, but the problem is NP complete [48]. This may not be serious for small SDF graphs, and for large ones we can use well studied heuristic methods, the best being members of a family of "critical path" methods [49]. An early example, known as the Hu-level-scheduling algorithm [50], closely approximates an optimal solution for most graphs [49], [51] and is simple.

V. A Preliminary SDF System

We are developing an experimental SDF programming system called Gabriel. It is written in Franz Lisp under Unix[1] 4.2bsd. Lisp was selected primarily because it eases the task of rapid prototyping. The Franz Lisp dialect was selected

[1]Unix is a trademark of AT&T.

because of its extensive support for foreign functions, so nodes written in different languages can be intermixed in a single SDF graph, in prinicple. In this section we describe the principles of Gabriel, emphasizing its use of the theory described above. The details of the complete system will be reported separately.

A. Simulation and Code Generation

In Gabriel, a programmer describes the topology of the data flow graph (currently using "connect" commands, but graphically in the future). Gabriel then constructs a single-processor schedule and invokes it for a specified number of cycles. The system is currently used in two different ways. First, the nodes that are invoked may perform signal processing functions and display signals or results. This is the *simulation* mode. Second, the nodes that are invoked may generate code for a DSP microcomputer. This is the *code generation* mode. The only difference between the two modes is in the definition of the nodes.

A Gabriel node can have some or all of the following:

1) *parameters* (e.g., FFT order),
2) *state variables* (e.g., memory in a biquad),
3) *inputs and outputs* (Gabriel supplies the buffers),
4) an *initialization routine* (e.g., initialize state variables),
5) a *run-time routine* (e.g., compute FFT), and
6) a *termination routine* (e.g., clean-up).

The initialization routines for all the nodes that have them are invoked once before the schedule is invoked. Then the run-time routines are invoked according to the schedule. After completing the specified number of cycles of the periodic schedule, the termination routines for all nodes that have them are invoked once.

The code generation mode is illustrated in Fig. 13 for a single processor. The Lisp routines associated with each of the Gabriel nodes write DSP code into a file when invoked. Gabriel begins by constructing a schedule, as shown. It proceeds by invoking initialization routines. In this case only the filter has an initialization routine, and that routine generates code to initialize memory for the state variables of the filter. The run-time routines are then invoked according to the schedule, for one cycle of the periodic schedule only. Each run-time routine can either generate code directly to perform its function or generate code to call a subroutine. The only node here that calls a subroutine is the filter. This node has a termination routine that generates the subroutine definition.

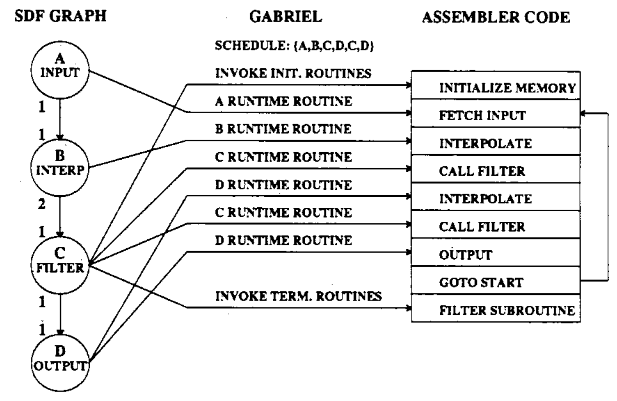


Fig. 13. An illustration of the mechanism by which Gabriel generates code for a DSP microcomputer from an SDF graph. First a schedule is constructed. Then the initialization routine for each of node that has one is run. In this example, only the filter node has an initialization routine, and it generates code to initialize the state variable memory of the filter. Then the run-time routines are invoked according to the schedule, and each generates code. In the case of the filter node, the code generated at this time is merely a subroutine call. The subroutine is then defined by the termination routine.

The first DSP microcomputers we have selected to target are the Motorola DSP56000 and AT&T Bell Labs DSP 32. These are state-of-the-art processors with manageable pipelining. Using a small set of nodes we have verified the principles described in this paper. To support a new processor all that is required is to generate a new set of nodes that are code generating nodes for that processor. No changes to Gabriel are required.

It is important to ensure that it is easy to create new nodes, but we expect that users will be strongly biased towards using standard nodes out of a node library, rather than writing their own. For this reason, the node library must be carefully selected for its generality, efficiency, and simplicity. Basic signal processing functions, such as digital filters, will be part of any standard library. Furthermore, since the nodes are code generators, considerable intelligence can be built into them. For example, a filter node may incorporate a filter design and synthesis package. Its parameters would be a description of the desired frequency response.

B. Static Buffering

An efficiency issue concerns the buffering of data between nodes. Code must be generated for the DSP to manage the buffers. In implementing an SDF graph, a direct approach is to use FIFO queues, but these are costly, requiring considerable run-time overhead even on processors with efficient support for circular buffers. For full generality, we need a mechanism that is functionally equivalent to a FIFO queue, but requiring less overhead. We accomplish this by statically computing the location of the input samples and the destination of the output samples for each invocation of a node. Thus the code that is generated merely reads its inputs directly from specified memory locations and writes its outputs directly to specified memory locations. We call this *static buffering*.

We now define static buffering precisely. Suppose that a single cycle of a periodic schedule contains q invocations of the node A. Denote these invocations $A_1, A_2, \cdots, A_q$. Recall that Gabriel invokes the code generator A_i only once but the code generated will be run infinitely often on the target processor. *Static buffering* means that every time the code generated by A_i runs it accesses the same locations for its inputs and outputs. This does not always occur unless we take some care in the design of the buffers. Consider the simple two-node system and its schedule in Fig. 14. Four memory locations, arranged as a circular buffer, suffice to buffer the data between nodes. If we use four locations, however, the buffer will not be static. The first run of the A_1 code writes its output data to locations 1 and 2. The first run of A_2 writes to 3 and 4, and A_3 to 1 and 2. The second run of the A_1 code, therefore, must write its output to locations 3 and 4. So the memory locations cannot be statically

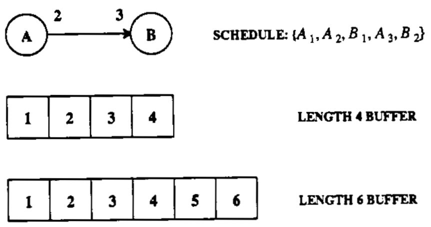


Fig. 14. For this simple example, given the schedule shown, the maximum number of samples in the buffer at any one time is four, but the buffer must be of length 6 in order to satisfy the static buffering condition.

built into the code. If a buffer of length 6 is used, however, every invocation of the A_1 code writes its data to locations 1 and 2.

We now show precisely the condition on the length of the buffer between two nodes so that the buffering is static. There are many ways to do this, so we select a direct method. Consider a node A that gets invoked q times and produces i samples to a circular buffer with total length N (the number of memory locations allocated). Static buffering occurs if and only if

$$iq = KN$$

for some integer K. This follows because there are iq samples written to the buffer between one invocation of A_i and the next. Thus if a buffer of length N is to be static there must exist an integer K such that

$$N = iq/K. \tag{5}$$

In the example of Fig. 14, for node A, $i = 2$ and $q = 3$, so static buffering requires that

$$N = 6/K$$

The only possible buffer lengths are 1, 2, 3, and 6, corresponding to K equal to 6, 3, 2, and 1. The Gabriel scheduler supplies the information that the buffer must be at least of length 4 (the number of data samples in the buffer at any time is a function of the schedule, and for the schedule shown the maximum is 4), so for static buffering the buffer must be of length 6.

For a buffer to be static it must satisfy a similar condition for the destination node (B in Fig. 14) as well as the source node. Suppose that the destination node is invoked p times in one cycle of the schedule and consumes j samples on each invocation. Then for a length N buffer to be static there must exist an integer L such that

$$N = pj/L.$$

From the result in Section IV-B, however, we know that $pj = iq$ so this condition is automatically satisfied if condition (5) is satisfied.

We can now summarize the static buffering strategy. Select the smallest N in the admissible set (given by (5)) that is greater than or equal to the minimum buffer size supplied by the scheduler. Occasionally, such an N will not exist (consider, for example, an arc with a large delay), in which case we need to increase the blocking factor. Increasing the blocking factor will increase q. It is expected that this situation will be rare, and its impact on code efficiency is not currently clear.

C. Asynchrony

It was mentioned in Section I that conditional operations such as those shown in Fig. 2 are not possible within the SDF model because they involve asynchronous nodes. The special class of asynchronous graphs represented by Fig. 2 can be handled efficiently nonetheless. In this subsection we propose a technique.

The graph in Fig. 2 can be divided into three synchronous subgraphs. The first consists of the switch, select, and any part of the system connected synchronously to them (not shown). The other two subgraphs are represented by the $f(\cdot)$ and $g(\cdot)$ nodes. Since a hierarchy is natural, the $f(\cdot)$ and $g(\cdot)$ nodes may themselves be complicated graphs, possibly with asynchronous nodes. Three schedules can be generated, one for each subgraph. The control signal into the select comes through the switch as shown to ensure that the switch is scheduled before the select in the subgraph containing them. The code segment we wish to generate has the form

```
        if (x) then goto label1;
        code for g(·)
        goto label2;
label1: code for f(·);
label2: code for select;
```

The switch is a special node that must have some knowledge of Gabriel internals. When it is invoked, it generates the code corresponding to the first line above. Then it recursively calls the Gabriel supervisor to invoke the schedule for $g(\cdot)$. When one cycle of that schedule has been invoked, Gabriel returns control to the switch node which writes the third line, writes the label on the fourth line, and invokes the schedule corresponding to $f(\cdot)$. When that returns, the switch writes the label on the fifth line and is finished; the main schedule proceeds. Interestingly, it also appears possible to extend the parallel scheduler to efficiently handle this type of graph. Thus with a slight extension of the SDF model we achieve generality by permitting conditional constructs.

Although apparently equivalent to the *if-then-else* capability of functional languages, constructs of the type shown in Fig. 2 are quite limited compared to the full potential of data flow. The switch and select nodes must occur in pairs, connected as shown. Furthermore, $f(\cdot)$ and $g(\cdot)$ subgraphs cannot have any connection to each other, synchronous or not, nor any connection to the rest of the system except through the switch and select nodes. Any graph containing switch and select nodes will have to be carefully checked for errors. More general asynchronous techniques require some study of the fundamentals of data flow. While there has been some effort [52], much work needs to be done before systematic asynchronous firing rules can be made to work in their full generality.

D. Generating Parallel Programs

The next logical step is to support a parallel architecture. For multiprocessor systems, given an SDF graph and the run-times of the nodes, Gabriel constructs the schedules, but does not execute the nodes or generate code, at this point. We use the scheduling functions to estimate the performance of an implementation of the modem of Fig. 6 on a π processor [46]. The π processor is a practical homogeneous parallel programmable digital signal processor with shared memory that has been proposed by the authors; it uses interleaved independent processes in a deep pipeline. The complete SDF graph for the model contains 28 large grain and atomic nodes and three sample rates. Before running the parallel scheduler we manually put delays on some of the feed-forward cutsets. Using the Hu-level-scheduling algorithm, up to seven parallel processors are fully utilized (have no significant idle time in one cycle of the periodic schedule). For seven processors, the iteration bound is met, so adding more processors does not speed up the schedule. We conclude that this application exhibits surprising concurrency, and that relatively simple scheduling algorithms are adequate to exploit the concurrency.

VI. Conclusion

We have outlined a paradigm called synchronous data flow for the description of digital signal processing algorithms. The description permits interpolation and decimation, and restricts neither the granularity (complexity) of a node nor the language in which it is programmed. It is hierarchical and encourages a structured methodology for building a system. Most importantly, SDF graphs explicitly display concurrency and permit automatic scheduling onto parallel processors. We illustrated how the SDF paradigm can be used to generate code for DSP microcomputers, including the management of limited forms of asynchrony that support conditionals. We also introduced the notion of static buffering. Using these techniques, we believe that compilers can be constructed which efficiently map SDF descriptions onto a wide variety of hardware architectures, thereby eliminating many of the costly translations from one description to another that are necessary under current methodologies.

References

[1] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-speed multiprocessors and compilation techniques," *IEEE Trans. Comput.*, vol. C-29, no. 9, pp. 763–776, Sept. 1980.

[2] J. B. Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, Nov. 1980.

[3] J. B. Dennis and D. P. Misunas, "A computer architecture for highly parallel signal processing," in *Proc. 1974 Nat. Comput. Conf.*, pp. 402–409, 1974.

[4] I. Watson and J. Gurd, "A practical data flow computer," *Computer*, vol. 15, no. 2, pp. 51–57, Feb. 1982.

[5] A. L. Davis and R. M. Keller, "Data flow program graphs," *Computer*, vol. 15, no. 2, pp. 26–41, Feb. 1982.

[6] W. B. Ackerman, "Data flow languages," *Computer*, vol. 15, no. 2, pp. 15–25, Feb. 1982.

[7] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data driven and demand driven computer architecture," University of Newcastle upon Tyne, Newcastle upon Tyne, UK, Tech. Rep., 1981.

[8] D. A. Schwartz and T. P. Barnwell, III, "Cyclo-static solutions: Optimal multiprocessor realizations of recursive algorithms," in *VLSI Signal Processing, II*, S-Y. Kung, R. E. Owen, and J. G. Nash, Eds. New York, NY: IEEE PRESS, 1986.

[9] D. A. Schwartz, "Synchronous multiprocessor realizations of shift-invariant flow graphs," Ph.D. dissertation, Georgia Inst. Technol. Tech. Rep. DSPL-85-2, July 1985.

[10] T. P. Barnwell, C. J. M. Hodges, and M. Randolf, "Optimum implementation of single time index signal flow graphs on synchronous multiprocessor machines," in *Proc. Int. Conf. on Acoustics, Speech, and Signal Processing* (May 3–5, 1982).

[11] T. P. Barnwell and D. A. Schwartz, "Optimal implementation of flow graphs on synchronous multiprocessors," in *Proc. 1983 Asilomar Conf. on Circuits and Systems*, Nov. 1983.

[12] M. Renfors and Y. Neuvo, "The maximum sampling rate of digital filters under hardware speed constraints," *IEEE Trans. Circuits and Syst.*, vol. CAS-28, no. 3, pp. 196–202, Mar. 1981.

[13] A. L. Davis, "The architecture and system method of DDM1: A recursively structured data driven machine," in *Proc. 5th Annu. Symp. on Computer Architecture*, pp. 210–215, Apr. 1978.

[14] J. Rumbaugh, "A data flow multiprocessor," *IEEE Trans. Comput.*, vol. C-26, no. 2, p. 138, Feb. 1977.

[15] R. G. Babb, "Parallel processing with large grain data flow techniques," *Computer*, vol. 17, no. 7, pp. 55–61, July 1984.

[16] G. Ungerboeck, "Fractional tap-spacing and consequences for clock recovery in data modems," *IEEE Trans. Commun.*, vol. COM-24, pp. 856–864, Aug. 1976.

[17] R. D. Gitlin and S. B. Weinstein, "Fractionally-spaced equalization: An improved digital transversal equalizer," *Bell Syst. Tech. J.*, vol. 60, no. 2, Feb. 1981.

[18] D. D. Falconer, "Jointly adaptive equalization and carrier recovery in two-dimensional digital communication systems," *Bell Syst. Tech. J.*, vol. 55, no. 3, Mar. 1976.

[19] D. J. Hait, "The BLOSIM simulation program," U. C. Berkeley, Master's Rep., Nov. 11, 1985.

[20] Kelly, Lochbaum, and Vyssotsky, "A block diagram compiler," *Bell Syst. Tech. J.*, vol. 40, no. 3, May 1961.

[21] B. Gold and C. Rader, *Digital Processing of Signals*. New York, NY: McGraw-Hill, 1969.

[22] B. Karafin, "The new block diagram compiler for simulation of sampled-data systems," in *AFIPS Conf. Proc.*, vol. 27, pp. 55–61, 1965.

[23] M. Dertouzous, M. Kaliske, and K. Polzen, "On-line simulation of block-diagram systems," *IEEE Trans. Comput.*, vol. C-18, no. 4, pp. 333–342, Apr. 1969.

[24] G. Korn, "High-speed block-diagram languages for microprocessors and minicomputers in instrumentation, control, and simulation," *Comput. Elec. Eng.*, vol. 4, pp. 143–159, 1977.

[25] W. Henke, "MITSYN–An interactive dialogue language for time signal processing," MIT Res. Lab. Electron. Memo. RLE-TM-1, Feb. 1975.

[26] T. Crystal and L. Kulsrud, "Circus," Inst. for Defense Analysis, Princeton, NJ, CRD Working Paper, Dec. 1974.

[27] Dipartimento di Elettronica, Politecnico di Torino, *TOPSIM III—Simulation Package for Communication Systems—User's Manual*. Torino, Italy: Politecnico di Torino.

[28] D. G. Messerschmitt, "A tool for structured functional simulation," *IEEE J. Selected Areas Commun.*, vol. SAC-2, no. 1, pp. 137–147, Jan. 1984.

[29] ——, "Structured interconnection of signal processing programs," in *Proc. Globecom84*, Dec. 1984.

[30] L. Snyder, "Parallel programming and the Poker programming environment," *Computer*, vol. 17, no. 7, pp. 27–36, July 1984.

[31] R. E. Crochiere and A. V. Oppenheim, "Analysis of linear digital networks," *Proc. IEEE*, vol. 63, no. 4, pp. 581–595, Apr. 1975.

[32] J. P. Brafman, J. Szczupak, and S. K. Mitra, "An approach to the implementation of digital filters using microprocessors," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-26, no. 5, pp. 442–446, Oct. 1978.

[33] J. Zeman and G. S. Moschytz, "Systematic design and programming of signal processors, using project management techniques," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-31, no. 6, pp. 1536–1549, Dec. 1983.

[34] S. Y. Kung, J. Annevelink, and D. Dewilde, "Heirarchical iterative flowgraph integration for VLSI array processers," in *VLSI Signal Processing*, P. R. Capello *et al.*, Eds. New York, NY: IEEE PRESS, 1984, pp. 294–305.

[35] F. Commoner and A. W. Holt, "Marked directed graphs," *J.Comput. Syst. Sci.*, vol. 5, pp. 511–523, 1971.

[36] R. Reiter, "Scheduling parallel computations," *J. Assoc. Comput. Mach.*, vol. 14, pp. 590–599, 1968.

[37] E. A. Lee, "A coupled hardware and software architecture for programmable digital signal processors," Ph.D. dissertation, Memo. UCB/ERL M86/54, EECD Dept., UC Berkeley, 1986.

[38] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. Assoc. Comput. Mach.*, vol. 14, pp. 563–590, 1967.

[39] S. K. Rao, "Regular iterative algorithms and their implementations on processor arrays," Ph.D. dissertation, Informat. Sys. Lab., Stanford Univ., Stanford, CA, Oct. 1985.

[40] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," *SIAM J.*, vol. 14, pp. 1390–1411, Nov. 1966.
[41] R. Reiter, "A study of a model for parallel computations," Ph.D. dissertation, Univ. Michigan, Ann Arbor, 1967.
[42] J. L. Peterson, "Petri nets," *Comput. Surv.*, vol. 9, no. 3, Sept. 1977.
[43] —, *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
[44] T. Agerwala, "Putting Petri nets to work," *Computer*, vol. 1, no. 1, pp. 85–94, Dec. 1979.
[45] R. M. Karp and R. E. Miller, "Parallel program schemata," *J. Comput. Syst. Sci.*, vol. 3, no. 2, pp. 147–195, May 1969.
[46] E. A. Lee and D. G. Messerschmitt, "Pipeline interleaved programmable DSP's: Parts I and II," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, pp. 1320–1333 and 1334–1345, Sept. 1987.
[47] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 2, pp. 24–35, Jan. 1987.
[48] E. G. Coffman, Jr., *Computer and Job Scheduling Theory.* New York, NY: Wiley, 1976.
[49] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. Assoc. Comput. Mach.*, vol. 17, no. 12, pp. 685–690, Dec. 1974.
[50] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Res.*, vol. 9, no. 6, pp. 841–848, 1961.
[51] W. H. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Trans. Comput.*, vol. C-24, pp. 1235–1238, Dec. 1975.
[52] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A formal definition of data flow graph models," *IEEE Trans. Comput.*, vol. C-35, no. 11, pp. 940–948, Nov. 1986.

Edward A. Lee (Member, IEEE) received the B.S. degree from Yale University, New Haven, CT, in 1979, the S.M. degree from MIT, Cambridge, MA, in 1981, and the Ph.D. degree from U.C. Berkeley, Berkeley, CA, in 1986.

From 1979 to 1982 he was a Member of the Technical Staff at Bell Labs in the Advanced Data Communications Laboratory, where he did exploratory work in voice-band data modem techniques and simultaneous voice and data transmission. Since July 1986 he has been an assistant professor at U.C. Berkeley. His research interests include architectures and software techniques for programmable digital signal processors, parallel computation, real-time software, and digital communications. He has taught a short course at U.C. Santa Barbara on programmable digital signal processor development and has consulted in industry. He was a recipient of the 1987 NSF Presidential Young Investigator Award, an IBM Faculty Development Award, the 1987 Sakrison award at U.C. Berkeley, and IBM Fellowship, a GE Fellowship, and the Samuel Silver Memorial Scholarship Award at U.C. Berkeley. He has several publications and one patent.

Dr. Lee is a member of Tau Beta Pi.

David G. Messerschmitt (Fellow, IEEE) received the B.S. degree from the University of Colorado, Boulder, in 1967, and the M.S. and Ph.D. degrees from the University of Michigan, Ann Arbor, in 1968 and 1971, respectively.

He is a Professor of Electrical Engineering and Computer Sciences at the University of California, Berkeley. From 1968 to 1977 he was a Member of the Technical Staff and later Supervisor at Bell Laboratories, Holmdel, NJ, where he did systems engineering, development, and research on digital transmission and digital signal processing (particularly relating to speech processing). His current research interests include analog and digital signal processing, adaptive filtering, digital communications (on the subscriber loop and fiber optics), architecture and software approaches to programmable digital signal processing, communication network design and protocols, and computer-aided design of communications and signal processing systems. He has published over 70 papers and has 10 patents issued or pending in these fields. Since 1977 he has also served as a consultant to a number of companies. He has organized and participated in a number of short courses and seminars devoted to continuing engineering education.

Dr. Messerschmitt is a member of Eta Kappa Nu, Tau Betta Pi, and Sigma Xi, and has several best paper awards. He is currently a Senior Editor of IEEE COMMUNICATIONS MAGAZINE, and is past Editor for Transmission Systems of the IEEE TRANSACTIONS ON COMMUNICATIONS and past member of the Board of Governors of the IEEE Communications Society.