Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing

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

Abstract—**Large grain data flow (LGDF) programming is natural and convenient for describing digital signal processing (DSP) systems, but its runtime overhead is costly in real time or cost-sensitive applications. In some situations, designers are not willing to squander computing resources for the sake of programmer convenience. This is particularly true when the target machine is a programmable DSP chip. However, the runtime overhead inherent in most LGDF implementations is not required for most signal processing systems because such systems are mostly *synchronous* (in the DSP sense). *Synchronous data flow* (SDF) differs from traditional data flow in that the amount of data produced and consumed by a data flow node is specified *a priori* for each input and output. This is equivalent to specifying the relative sample rates in signal processing system. This means that the scheduling of SDF nodes need not be done at runtime, but can be done at compile time (statically), so the runtime overhead evaporates. The sample rates can all be different, which is not true of most current data-driven digital signal processing programming methodologies. Synchronous data flow is closely related to *computation graphs*, a special case of *Petri nets*.**

This self-contained paper develops the theory necessary to statically schedule SDF programs on single or multiple processors. A class of static (compile time) scheduling algorithms is proven valid, and specific algorithms are given for scheduling SDF systems onto single or multiple processors.

Index Terms—**Block diagram, computation graphs, data flow digital signal processing, hard real-time systems, multiprocessing, Petri nets, static scheduling, synchronous data flow.**

I. Introduction

TO achieve high performance in a processor specialized for signal processing, the need to depart from the simplicity of von Neumann computer architectures is axiomatic. Yet, in the software realm, deviations from von Neumann programming are often viewed with suspicion. For example, in the design of most successful commercial signal processors today [1]–[5], compromises are made to preserve sequential programming. Two notable exceptions are the Bell Labs DSP family [6], [7] and the NEC data flow chip [8], both of which are programmed with concurrency in mind. For the majority, however, preserving von Neumann programming style is given priority.

This practice has a long and distinguished history. Often, a new non-von Neumann architecture has elaborate hardware and software techniques enabling a programmer to write sequential code irrespective of the parallel nature of the underlying hardware. For example, in machines with multiple function units, such as the CDC6600 and Cray family, so called "scoreboarding" hardware resolves conflicts to ensure the integrity of sequential code. In deeply pipelined machines such as the IBM 360 Model 91, interlocking mechanisms [9] resolve pipeline conflicts. In the M.I.T. Lincoln Labs signal processor [10] specialized associative memories are used to ensure the integrity of data precedences.

The affinity for von Neumann programming is not all surprising, stemming from familiarity and a proven track record, but the cost is high in the design of specialized digital signal processors. Comparing two pipelined chips that differ radically only in programming methodology, the TI TMS32010 [2] and the Bell Labs DSP20, a faster version of the DSP1 [6], we find that they achieve exactly the same performance on the most basic benchmark, the FIR (finite impulse response) filter. But the Bell Labs chip outperforms the TI chip on the next most basic benchmark, the IIR (infinite impulse response) filter. Surprisingly, close examination reveals that the arithmetic hardware (multiplier and ALU) of the Bell Labs chip is half as fast as in the TI chip. The performance gain appears to follow from the departure from conventional sequential programming.

However, programming the Bell Labs chip is not easy. The code more closely resembles horizontal microcode than assembly languages. Programmers invariably adhere to the quaint custom of programming these processors in assembler-level languages, for maximum use of hardware resources. Satisfactory compilers have failed to appear.

In this paper, we propose programming signal processors using a technique based on large grain data flow (LGDF) languages [11], which should ease the programming task by enhancing the modularity of code and permitting algorithms to be described more naturally. In addition, concurrency is immediately evident in the program description, so parallel hardware resources can be used more effectively. We begin by reviewing the data flow paradigm and its relationship with previous methods applied to signal processing. *Synchronous data flow* (SDF) is introduced, with its suitability for describing signal processing systems explained. The advantage of SDF over conventional data flow is that more efficient runtime code can be generated because the data flow nodes can be scheduled at compile time, rather than at runtime. A class of algorithms for constructing sequential (single processor) schedules is proven valid, and a simple

Manuscript received August 15, 1985; revised March 17, 1986. This work was supported in part by the National Science Foundation under Grant ECS-8211071, an IBM Fellowship, and a grant from the Shell Development Corporation.

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

IEEE Log Number 8611442.

heuristic for constructing parallel (multiprocessor) schedules is described. Finally, the limitations of the model are considered.

II. The Data Flow Paradigm

In data flow, a program is divided into pieces (*nodes* or *blocks*) which can execute (*fire*) whenever input data are available [12], [13]. An algorithm is described as a *data flow graph*, a directed graph where the nodes represent functions and the arcs represent data paths, as shown in Fig. 1. Signal processing algorithms are usually described in the literature by a combination of mathematical expressions and block diagrams. Block diagrams are *large grain data flow* (LGDF) graphs, [14]–[16], in which the nodes or blocks may be *atomic* (from the Greek *atomos*, or indivisble), such as adders or multipliers, or nonatomic (large grain), such as digital filters, FFT units, modulators, or phase locked loops. The arcs connecting blocks show the signal paths, where a *signal* is simply an infinite stream of data, and each data token is called a *sample*. The complexity of the functions (the *granularity*) will determine the amount of parallelism available because, while the blocks can sometimes be executed concurrently, we make no attempt to exploit the concurrency inside a block. The functions within the blocks can be specified using conventional von Neumann programming techniques. If the granularity is at the level of signal processing subsystems (second-order sections, butterfly units, etc.), then the specification of a system will be extremely natural and enough concurrency will be evident to exploit at least small-scale parallel processors. The blocks can themselves represent another data flow graph, so the specification can be hierarchical. This is consistent with the general practice in signal processing where, for example, an adaptive equalizer may be treated as a block in a large system, and may be itself a network of simpler blocks.

LGDF is ideally suited for signal processing, and has been adopted in simulators in the past [17]. Other signal processing systems use a data-driven paradigm to partition a task among cooperating processors [18], and many so called "block diagram languages" have been developed to permit programmers to describe signal processing systems more naturally. Some examples are Blodi [19], Patsi [20], Blodib [21], Lotus [22], Dare [23], Mitsyn [24], Circus [25], and Topsim [26]. But these simulators are based on the principle of "next state simulation" [20], [27] and thus have difficulty with multiple sample rates, not to mention asynchronous systems. (We use the term "asynchronous" here in the DSP sense to refer to systems with sample rates that are not related by a rational multiplicative factor.) Although true asynchrony is rare in signal processing, multiple sample rates are common, stemming from the frequent use of decimation and interpolation. The technique we propose here handles multiple sample rates easily.

In addition to being natural for DSP, large grain data flow has another significant advantage for signal processing. As long as the integrity of the flow of data is preserved, any implementation of a data flow description will produce the same results. This means that the same software description of a signal processing system can be simulated on a single processor or multiple processors, implemented in specialized hardware, or even, ultimately, compiled into a VLSI chip [28].

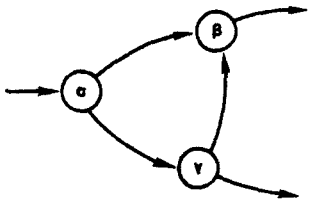

Fig. 1. A three node data flow graph with one input and two outputs. The nodes represent functions of arbitrary complexity, and the arcs represent paths on which sequences of data (*tokens* or *samples*) flow.

III. Synchronous Data Flow Graphs

In this paper we concentrate on *synchronous* systems. At the risk of being pedantic, we define this precisely. A block is a function that is invoked when there is enough input available to perform a computation (blocks lacking inputs can be invoked at any time). When a block is invoked, it will *consume* a fixed number of new input samples on each input path. These samples may remain in the system for some time to be used as old samples [17], but they will never again be considered new samples. A block is said to be *synchronous* if we can specify *a priori* the number of input samples consumed on each input and the number of output samples produced on each output each time the block is invoked. Thus, a synchronous block is shown in Fig. 2(a) with a number associated with each input or output specifying the number of inputs consumed or the number of outputs produced. These numbers are part of the block definition. For example, a digital filter block would have one input and one output, and the number of input samples consumed or output samples produced would be one. A 2:1 decimator block would also have one input and one output, but would consume two samples for every sample produced. A *synchronous data flow (SDF) graph* is a network of synchronous blocks, as in Fig. 2(b).

SDF graphs are closely related to computation graphs, introduced in 1966 by Karp and Miller [29] and further explored by Reiter [30]. Computation graphs are slightly more elaborate than SDF graphs, in that each input to a block has two numbers associated with it, a *threshold* and the number of samples consumed. The threshold specifies the number of samples required to invoke the block, and may be different from the number of samples consumed by the block. It cannot, of course, be smaller than the number of samples consumed. The use of a distinct threshold in the model, however, does not significantly change the results presented in this paper, so for simplicity, we assume these two numbers are the same. Karp and Miller [29] show that computations specified by a computation graph are determinate, meaning that the same computations are performed by any proper execution. This type of theorem, of course, also underlies the validity of data flow descriptions. They also give a test to determine whether a computation terminates, which is potentially useful because in signal processing we are mainly interested in computations that do not terminate. We assume that signal processing

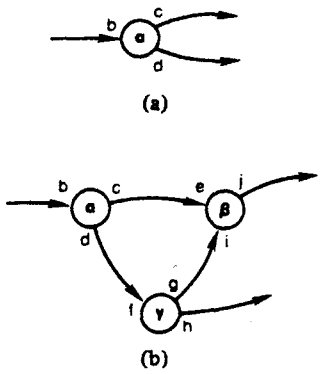

Fig. 2. (a) A synchronous node. (b) A synchronous data flow graph.

systems repetitively apply an algorithm to an infinite sequence of data. To make it easier to describe such applications, we expand the model slightly to allow nodes with no inputs. These can fire at any time. Other results presented in [29] are only applicable to computations that terminate, and therefore are not useful in our application.

Computation graphs have been shown to be a special case of *Petri nets* [31]–[33] or *vector addition systems* [34]. These more general models can be used to describe asynchronous systems. There has also been work with models that are special cases of computation graphs. In 1971, Commoner and Holt [35] described *marked directed graphs*, and reached some conclusions similar to those presented in this paper. However, marked directed graphs are much more restricted than SDF graphs because they constrain the number of samples produced or consumed on any arc to unity. This excessively restricts the sample rates in the system, reducing the utility of the model. In 1968, Reiter [36] simplified the computation graph model in much the same way (with minor variations), and tackled a scheduling problem. However, his scheduling problem assumes that each node in the graph is a processor, and the only unknown is the firing time for the invocation of each associated function. In this paper we preserve the generality of computation graphs and solve a different scheduling problem, relevant to data flow programming, in which nodes represent functions that must be mapped onto processors.

Implementing the signal processing system described by a SDF graph requires *buffering* the data samples passed between blocks and *scheduling* blocks so that they are executed when data are available. This could be done *dynamically*, in which case a runtime supervisor determines when blocks are ready for execution and schedules them onto processors as they become free. This runtime supervisor may be a software routine or specialized hardware, and is the same as the control mechanisms generally associated with data flow. It is a costly approach, however, in that the supervisory overhead can become severe, particularly if relatively little computation is done each time a block is invoked.

SDF graphs, however, can be scheduled statically (at compile time), regardless of the number of processors, and the overhead associated with dynamic control evaporates. Specifically, a *large grain compiler* determines the order in which nodes can be executed and constructs sequential code for each processor. Communication between nodes and between processors is set up by the compiler, so no runtime control is required beyond the traditional sequential control in the processors. The LGDF paradigm gives the programmer a natural interface for easily constructing well structured signal processing programs, with evident concurrency, and the large grain compiler maps this concurrency onto parallel processors. This paper is dedicated mainly to demonstrating the feasibility of such a large grain compiler.

IV. A Synchronous Large Grain Compiler

We need a methodology for translating from an SDF graph to a set of sequential programs running on a number of processors. Such a compiler has the two following basic tasks.

- Allocation of shared memory for the passing of data between blocks, if shared memory exists, or setting up communication paths if not.
- Scheduling blocks onto processors in such a way that data is available for a block when that block is invoked.

The first task is not an unfamiliar one. A single processor solution (which also handles asynchronous systems) is given by the buffer management techniques in Blosim [17]. Simplifications of these techniques that use the synchrony of the system are easy to imagine, as are generalizations to multiple processors, so this paper will concentrate on the second task, that of scheduling blocks onto processors so that data are available when a block is invoked.

Some assumptions are necessary.

- The SDF graph is nonterminating (cf. [29], [30]) meaning that it can run forever without deadlock. As mentioned earlier, this assumption is natural for signal processing.
- The SDF graph is connected. If not, the separate graphs can be scheduled separately using subsets of the processors.

- The SDF graph is nonterminating (cf. [29], [30]) meaning that it can run forever without deadlock. As mentioned earlier, this assumption is natural for signal processing.

Specifically, our ultimate goal is a periodic admissible parallel schedule, designated PAPS. The schedule should be *periodic* because of the assumption that we are repetitively applying the same program on an infinite stream of data. The desired schedule is *admissible*, meaning that blocks will be scheduled to run only when data are available, and that a finite amount of memory is required. It is *parallel* in that more than one processing resource can be used. A special case is a periodic admissible *sequential* schedule, or PASS, which implements an SDF graph on a single processor. The method for constructing a PASS leads to a simple solution to the problem of constructing a PAPS, so we begin with the sequential schedule.

A. Construction of a PASS

A simple SDF graph is shown in Fig. 3, with each block and each arc labeled with a number. (The connections to the outside world are not considered, and for the remainder of this paper, will not be shown. Thus, a block with one input from the outside will be considered a block with no inputs, which can therefore be scheduled at any time. The limitations of this approximation are discussed in Section V.) An SDF graph can

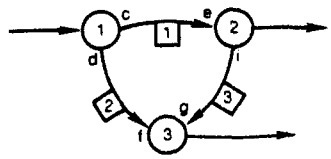

Fig. 3. SDF graph showing the numbering of the nodes and arcs. The input and output arcs are ignored for now.

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 in Fig. 3, 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 graph in Fig. 3 we get

$$\Gamma = \begin{bmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & i & -g \end{bmatrix}. \tag{1}$$

This matrix can be 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 block is invoked. This difference should clearly be zero for a correctly constructed graph, so the Γ entry describing a self-loop should be zero.

We can replace each arc with a FIFO queue (buffer) to pass data from one block to another. The size of the queue will vary at different times in the execution. Define the vector $\boldsymbol{b}(n)$ to contain the queue sizes of all the buffers at time n. In Blosim [17] buffers are also used to store old samples (samples that have been "consumed"), making implementations of delay lines particularly easy. These past samples are not considered part of the buffer size here.

For the sequential schedule, only one block can be invoked at a time, and for the purposes of scheduling it does not matter how long it runs. Thus, the index n can simply be incremented each time a block finishes and a new block is begun. We specify the block invoked at time n with a vector $\boldsymbol{v}(n)$, which has a one in the position corresponding to the number of the block that is invoked at time n and zeros for each block that is not invoked. For the system in Fig. 3, in a sequential schedule, $\boldsymbol{v}(n)$ can take one of three values,

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

depending on which of the three blocks is invoked. Each time a block 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 buffer queues caused by invoking a node is given by

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

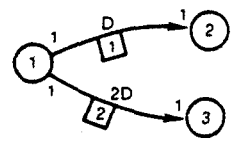

Fig. 4. An example of an SDF graph with delays on the arcs.

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

The simple computation model is powerful. First we note that the computation model handles delays. 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 nth sample consumed by B will be the $(n-1)$th sample produced by A. This implies that the first sample the destination block consumes is not produced by the source block 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 setting an initial condition for (3). Specifically, the initial buffer state, $\boldsymbol{b}(0)$, should have a d in the position corresponding to the arc with the delay of d units.

To make this idea firm, consider the example system in Fig. 4. The symbol "D" on an arc means a single sample delay, while "$2D$" means a two-sample delay. The initial condition for the buffers is thus

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

Because of these initial conditions, block 2 can be invoked once and block 3 twice before block 1 is invoked at all. Delays, therefore, affect the way the system starts up.

Given this computation model we can

- find necessary and sufficient conditions for the existence of a PASS, and hence a PAPS;
- find practical algorithms that provably find a PASS if one exists;
- find practical algorithms that construct a reasonable (but not necessarily optimal) PAPS, if a PASS exists.

We begin by showing that a necessary condition for the existence of a PASS is

$$\text{rank } (\Gamma) = s - 1 \tag{5}$$

where s is the number of blocks in the graph. We need a series of lemmas before we can prove this. The word "node" is used below to refer to the blocks because it is traditional in graph theory.

Lemma 1: All topology matrices for a given SDF graph have the same rank.

Proof: Topology matrices are related by renumbering of nodes and arcs, which translates into row and column permutations in the topology matrix. Such operations preserve the rank. Q.E.D.

Lemma 2: A topology matrix for a *tree* graph has rank $s-1$ where s is the number of nodes (a tree is a connected graph without cycles, where we ignore the directions of the arcs).

Proof: Proof is by induction. The lemma is clearly true for a two-node tree. Assume that for an N node tree

$rank(\Gamma_N) = N - 1$. Adding one node and one link connecting that node to our graph will yield an $N + 1$ node tree. A topology matrix for the new graph can be written

$$\Gamma_{N+1} = \left[\begin{array}{c|c} \Gamma_N & \boldsymbol{O} \\ \hline \boldsymbol{\rho}^T & \end{array}\right]$$

where $\boldsymbol{O}$ is a column vector full of zeros, and $\boldsymbol{\rho}^T$ is a row vector corresponding to the arc we just added. The last entry in the vector $\boldsymbol{\rho}^T$ is nonzero because the node we just added corresponds to the last column, and it must be connected to the graph. Hence, the last row is linearly independent from the other rows, so $rank(\Gamma_{N+1}) = rank(\Gamma_N) + 1$. Q.E.D.

Lemma 3: For a connected SDF graph with topology matrix Γ

$$\text{rank } (\Gamma) \geq s - 1$$

where s is the number of nodes in the graph.

Proof: Consider any spanning tree τ of the connected SDF graph (a spanning tree is a tree that includes every node in the graph). Now define Γ_τ to be the topology matrix for this subgraph. By Lemma 2 rank $(\Gamma_\tau) = s - 1$. Adding arcs to the subgraph simply adds rows to the topology matrix. Adding rows to a matrix can increase the rank, if the rows are linearly independent of existing rows, but cannot decrease it. Q.E.D.

Corollary: rank (Γ) is $s - 1$ or s.

Proof: Γ has only s columns, so its rank cannot exceed s. Therefore, by Lemma 3, s and $s - 1$ are the only possibilities. Q.E.D.

Definition 1: An *admissible sequential schedule* ϕ is a nonempty ordered list of nodes such that if the nodes are executed in the sequence given by ϕ, the amount of data in the buffers ("buffer sizes") will remain nonnegative and bounded. Each node must appear in ϕ at least once.

A *periodic admissible sequential schedule* (PASS) is a periodic and infinite admissible sequential schedule. It is specified by a list ϕ that is the list of nodes in one period.

For the example in Fig. 6, $\phi = \{1, 2, 3, 3\}$ is a PASS, but $\phi = \{2, 1, 3, 3\}$ is not because node 2 cannot be run before node 1. The list $\phi = \{1, 2, 3\}$ is not a PASS because the infinite schedule resulting from repetitions of this list will result in an infinite accumulation of data samples on the arcs leading into node 3.

Theorem 1: For a connected SDF graph with s nodes and topology matrix Γ, rank $(\Gamma) = s - 1$ is a necessary condition for a PASS to exist.

Proof: We must prove that the existence of a PASS of period p implies rank $(\Gamma) = s - 1$. Observe from (3) that we can write

$$\boldsymbol{b}(p) = \boldsymbol{b}(0) + \Gamma \boldsymbol{q}$$

where

$$\boldsymbol{q} = \sum_{n=0}^{p-1} \boldsymbol{v}(n).$$

Since the PASS is periodic, we can write

$$\boldsymbol{b}(np) = \boldsymbol{b}(0) + n\Gamma\boldsymbol{q}.$$

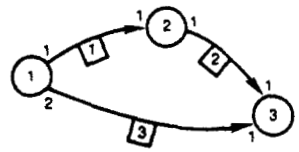

Fig. 5. Example of a defective SDF graph with sample rate inconsistencies. The topology matrix is

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix} \quad \text{rank } (\Gamma) = s = 3.$$

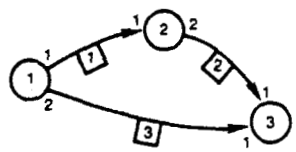

Fig. 6. An SDF graph with consistent sample rates has a positive integer vector $\boldsymbol{q}$ in the nullspace of the topology matrix Γ.

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \quad \boldsymbol{q} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \in \eta(\Gamma)$$

Since the PASS is admissible, the buffers must remain bounded, by Definition 1. The buffers remain bounded if and only if

$$\Gamma\boldsymbol{q} = \boldsymbol{O}$$

where $\boldsymbol{O}$ is a vector full of zeros. For $\boldsymbol{q} \neq \boldsymbol{O}$, this implies that $rank(\Gamma) < s$ where s is the dimension of $\boldsymbol{q}$. From the corollary of Lemma 3, $rank(\Gamma)$ is either s or $s - 1$, and so it must be $s - 1$. Q.E.D.

This theorem tells us that if we have a SDF graph with a topology matrix of rank s, that the graph is somehow defective, because no PASS can be found for it. Fig. 5 illustrates such a graph and its topology matrix. Any schedule for this graph will result either in deadlock or unbounded buffer sizes, as the reader can easily verify. The rank of the topology matrix indicates a *sample rate inconsistency* in the graph. In Fig. 6, by contrast, a graph without this defect is shown. The topology matrix has rank $s - 1 = 2$, so we can find a vector $\boldsymbol{q}$ such that $\Gamma\boldsymbol{q} = \boldsymbol{O}$. Furthermore, the following theorem shows that we can find a positive integer vector $\boldsymbol{q}$ in the nullspace of Γ. This vector tells us how many times we should invoke each node in one period of a PASS. Referring again to Fig. 6, the reader can easily verify that if we invoke node 1 once, node 2 once, followed by node 3 twice, that the buffers will end up once again in their initial state. As before, we prove some lemmas before getting to the theorem.

Lemma 4: Assume a connected SDF graph with topology matrix Γ. Let $\boldsymbol{q}$ be any vector such that $\Gamma\boldsymbol{q} = \boldsymbol{O}$. Denote a connected path through the graph by the set $B = \{b_1, \cdots, b_L\}$ where each entry designates a node, and node b_1 is connected to node b_2, node b_2 to node b_3, up to b_L. Then all q_i, $i \in B$ are zero, or all are strictly positive, or all are strictly negative. Furthermore, if any q_i is rational then all q_i are rational.

Proof: By induction. First consider a connected path of two nodes, $B_2 = \{b_1, b_2\}$. If the arc connecting these two nodes is the jth arc, then

$$q_{b_1}\Gamma_{jb_1} + q_{b_2}\Gamma_{jb_2} = 0$$

(by definition of the topology matrix, the jth row has only two entries). Also by definition, Γ_{jb_1} and Γ_{jb_2} are nonzero integers of opposite sign. The lemma thus follows immediately for B_2.

Now assuming the lemma is true for B_n, proving it true for B_{n+1} is trivial, using the same reasoning as in the proof for B_2, and considering the connection between nodes b_n and b_{n+1}.

Corollary: Given an SDF graph as in Lemma 4, either all q_i are zero, or all are strictly positive, or all are strictly negative. Furthermore, if any one q_i is rational, then all are.

Proof: In a connected SDF graph, a path exists from any node to any other. Thus, the corollary follows immediately from the lemma.

Theorem 2: For a connected SDF graph with s nodes and topology matrix Γ, and with $rank(\Gamma) = s - 1$, we can find a positive integer vector $\boldsymbol{q} \neq \boldsymbol{O}$ such that $\Gamma\boldsymbol{q} = \boldsymbol{O}$ where $\boldsymbol{O}$ is the zero vector.

Proof: Since $rank(\Gamma) = s - 1$, a vector $\boldsymbol{v} \neq \boldsymbol{O}$ can be found such that $\Gamma\boldsymbol{v} = \boldsymbol{O}$. Furthermore, for any scalar α, $\Gamma(\alpha\boldsymbol{v}) = \boldsymbol{O}$. Let $\alpha = 1/v_1$ and $\boldsymbol{v}' = \alpha\boldsymbol{v}$. Then $v_1' = 1$, and by the corollary to lemma 4, all other elements in $\boldsymbol{v}'$ are positive rational numbers. Let η be a common multiple of all the denominators of the elements of $\boldsymbol{v}'$ and let $\boldsymbol{q} = \eta\boldsymbol{v}'$. Then $\boldsymbol{q}$ is a positive integer vector such that $\Gamma\boldsymbol{q} = \boldsymbol{O}$. Q.E.D.

It may be desirable to solve for the *smallest* positive integer vector in the nullspace, in the sense of the sum of the elements. To do this, reduce each rational entry in $\boldsymbol{v}'$ so that its numerator and denominator are relatively prime. Euclid's algorithm (see for example [37]) will work for this. Now find the *least* common multiple η of all the denominators, again using Euclid's algorithm. Now $\eta\boldsymbol{v}'$ is the smallest positive integer vector in the nullspace of Γ.

We now have a necessary condition for the existence of a PASS, that the rank of Γ be $s - 1$. A sufficient condition and an algorithm for finding a PASS would be useful. We now characterize a class of algorithms that will find a PASS if such exists, and will fail clearly if not. Thus, successful completion of such an algorithm is a sufficient condition for the existence of the PASS.

Definition 2: A *predecessor* to a node x is a node feeding data to x.

Lemma 5: To determine whether a node x in a SDF graph can be scheduled at time i, it is sufficient to know how many times x and its predecessors have been scheduled, and to know $\boldsymbol{b}(0)$, the initial state of the buffers. That is, we need not know in what order the predecessors were scheduled nor what other nodes have been scheduled in between.

Proof: To schedule node η, each input buffer must have sufficient data. The size of each input buffer j at time i is given by $[\boldsymbol{b}(i)]_j$, the jth entry in the vector $\boldsymbol{b}(i)$. From (3) we can write

$$\boldsymbol{b}(i) = \boldsymbol{b}(0) + \Gamma\boldsymbol{q}(i)$$

where

$$\boldsymbol{q}(i) = \sum_{n=0}^{i-1} \boldsymbol{v}(n). \tag{6}$$

The vector $\boldsymbol{q}(i)$ only contains information about how many times each node has been invoked before iteration i. The buffer sizes $[\boldsymbol{b}(i)]_j$ clearly depend only on $[\boldsymbol{b}(0)]_j$ and $[\Gamma\boldsymbol{q}(i)]_j$. The jth row of Γ has only two entries, corresponding to the two nodes connected to the jth buffer, so only the two corresponding entries of the $\boldsymbol{q}(i)$ vector can affect the buffer size. These entries specify the number of times x and its predecessors have been invoked, so this information and the initial buffer sizes $[\boldsymbol{b}(0)]_j$ is all that is needed. Q.E.D.

Definition 3: (Class S algorithms) Given a positive integer vector $\boldsymbol{q}$ s.t. $\Gamma\boldsymbol{q} = \boldsymbol{O}$ and an initial state for the buffers $\boldsymbol{b}(0)$, the ith node is *runnable* at a given time if it has not been run q_i times and running it will not cause a buffer size to go negative. A *class S algorithm* 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 algorithms 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 ("S" for Sequential) construct static schedules by simulating the effects on the buffers of an actual run. That is, the node programs are not actually run. But they could be run, and the algorithm would not change in any significant way. Therefore, any dynamic (runtime) scheduling algorithm becomes a class S algorithm simply by specifying a stopping condition, which depends on the vector $\boldsymbol{q}$. It is necessary to prove that the stopping condition is sufficient to construct a PASS for any valid graph.

Theorem 3: Given a SDF graph with topology matrix Γ and given a positive integer vector $\boldsymbol{q}$ s.t. $\Gamma\boldsymbol{q} = \boldsymbol{O}$, if a PASS of period $p = \mathbf{1}^T\boldsymbol{q}$ exists, where $\mathbf{1}^T$ is a row vector full of ones, any class S algorithm will find such a PASS.

Proof: It is sufficient to prove that if a PASS ϕ of any period p exists, a class S algorithm will not deadlock before the termination condition is satisfied.

Assume that a PASS ϕ exists, and define $\phi(n)$ to be its first n entries, for any n such that $1 \leqslant n \leqslant p$. Assume a given class S algorithm iteratively constructs a schedule, and define $\chi(n)$ to be the list of the first n nodes scheduled by iteration n.

We need to show that as n increases, the algorithm will build $\chi(n)$ and not deadlock before $n = p$, when the termination condition is satisfied. That is, we need to show that for all $n \in (1, \cdots, p)$, there is a node that is runnable for any $\chi(n)$ that the algorithm may have constructed.

If $\chi(n)$ is any permutation of $\phi(n)$, then the $(n + 1)$th entry in ϕ is runnable by Lemma 5 because all necessary predecessors must be in $\phi(n)$, and thus in $\chi(n)$. Otherwise, the first node α in $\phi(n)$ and not in $\chi(n)$ is runnable, also by Lemma 5. This is true for all $n \in (1, \cdots, p)$, so the algorithm will not deadlock before $n = p$.

At $n = p$, each node i has been scheduled q_i times because no node can be scheduled more that q_i times (by Definition 3). and $p = \mathbf{1}^T\boldsymbol{q}$. Therefore, the termination condition is satisfied, and $\chi(p)$ is a PASS. Q.E.D.

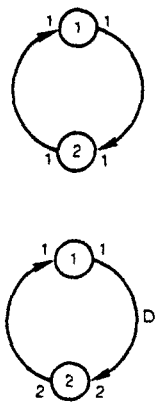

Fig. 7. Two SDF graphs with consistent sample rates but no admissible schedule.

Theorem 3 tells us that if we are given a positive integer vector q in the nullspace of the topology matrix, that class S algorithms will find a PASS with its period equal to the sum of the elements in the vector, if such a PASS exists. It is possible, even if rank $(\Gamma) = s - 1$ for no PASS to exist. Two such graphs are shown in Fig. 7. Networks with insufficient delays in directed loops are not computable.

One problem remains. There are an infinite number of vectors in the nullspace of the topology matrix. How do we select one to use in the class S algorithm? We now set out to prove that given any positive integer vector in the nullspace of the topology matrix, if a class S algorithm fails to find a PASS then no PASS of any period exists.

Lemma 6: Connecting one more node to a graph increases the rank of the topology matrix by at least one.

The proof of this lemma follows the same kinds of arguments as the proof of Lemma 2. Rows are added to the topology matrix to describe the added connections to the new node, and these rows must be linearly independent of rows already in the topology matrix.

Lemma 7: For any connected SDF graph with s nodes and topology matrix Γ, a connected subgraph L with m nodes has a topology matrix Γ_L for which

$$rank(\Gamma) = s - 1 \Rightarrow rank(\Gamma_L) = m - 1$$

i.e., all subgraphs have the right rank.

Proof: By contraposition. We prove that

$$rank(\Gamma_L) \neq m - 1 \Rightarrow rank(\Gamma) \neq s - 1.$$

From the corollary to Lemma 3, if $rank(\Gamma_L) \neq m - 1$ then $rank(\Gamma_L) = m$. Then $rank(\Gamma) \geq m + (s - m) = s$, by repeated applications of Lemma 6, so $rank(\Gamma) = s$. Q.E.D.

The next lemma shows that given a nullspace vector q, in order to run any node the number of times specified by this vector, it is not necessary to run any other node more than the number of times specified by the vector.

Lemma 8: Consider the subgraph of a SDF graph formed by any node α and all its immediate predecessors (nodes that feed it data, which may include α itself). Construct a topology matrix Γ for this subgraph. If the original graph has a PASS, then by Theorem 1 and Lemma 7, rank $(\Gamma) = m - 1$ where m is the number of nodes in the subgraph. Find any positive integer vector q s.t. $\Gamma q = O$. Such a vector exists because of Theorem 2. Then it is never necessary to run any predecessor β more than q_β times in order to run α x times, for any $x \leq q_\alpha$.

Proof: The node α will not consume any data produced by the yth run of β for any $y > q_\beta$. From the definition of Γ and q we know that $aq_\alpha = bq_\beta$ where a and b are the amount of data consumed and produced on the link from β to α. Therefore, running β only q_β times generates enough data on the link to run α q_α times. More runs will not help. Q.E.D.

Theorem 4: Given a SDF graph with topology matrix Γ and a positive integer vector q s.t. $\Gamma q = O$, a PASS of period $p = \mathbf{1}^T q$ exists if and only if a PASS of period Np exists for any integer N.

Proof:

Part 1: It is trivial to prove that the existence of a PASS of period p implies the existence of a PASS of period Np because the first PASS can be composed N times to produce the second PASS.

Part 2: We now prove that the existence of a PASS ϕ of period Np implies the existence of a PASS of period p. Consider the subset θ of ϕ containing the first q_α runs of each node α. If θ is the first p elements of ϕ then it is a schedule of period p and we are done. It it is not, then there must be some node β that is executed more than q_β times before all nodes have been executed q times. But by Lemma 8, these "more than q" executions of β cannot be necessary for the later "less than or equal to q" executions of other nodes. Therefore, the "less than or equal to q" executions can be moved up in the list ϕ so that they precede all "more than q" executions of β, yielding a new PASS ϕ' of period Np. If this process is repeated until all "less than q" executions precede all "more than q" executions, then the first p elements of the resulting schedule will constitute a schedule of period p. Q.E.D.

Corollary: Given any positive integer vector $q \in \eta(\Gamma)$, the null space of Γ, a PASS of period $p = \mathbf{1}^T q$ exists if and only if a PASS exists of period $r = \mathbf{1}^T v$ for any other positive integer vector $v \in \eta(\Gamma)$.

Proof: For any PASS at all to exist, it is necessary that $rank(\Gamma) = s - 1$, by Theorem 1. So the nullspace of Γ has dimension one, and we can find a scalar c such that

$$q = cv.$$

Furthermore, if both of these vectors are integer vectors, then c is rational and we can write

$$c = \frac{n}{d}$$

where n and d are both integers. Therefore,

$$dq = nv.$$

By Theorem 4, a PASS of period $p = \mathbf{1}^T q$ exists if and only if a PASS of period $dp = \mathbf{1}^T(dq)$ exists. By Theorem 4 again, a PASS of period dp exists if and only if a PASS of period $r = \mathbf{1}^T v$ exists. Q.E.D.

Discussion: The four theorems and their corollaries have great practical importance. We have specified a very broad

class of algorithms, designated *class S algorithms*, which, given a positive integer vector q in the nullspace of the topology matrix, find a PASS with period equal to the sum of the elements in q. Theorem 3 guarantees that these algorithms will find a PASS if one exists. Theorems 1 and 2 guarantee that such a vector q exists if a PASS exists. The corollary to Theorem 4 tells us that it does not matter what positive integer vector we use from the nullspace of the topology matrix, so we can simplify our system by using the smallest such vector, thus obtaining a PASS with minimum period.

Given these theorems, we now give a simple sequential scheduling algorithm that is of class S, and therefore will find a PASS if one exists.

1) Solve for the smallest positive integer vector $q \in \eta(\Gamma)$.
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.

Theorem 3 tells us that this algorithms will not deadlock if a PASS exists. Two SDF graphs which cause deadlock and have no PASS are shown in Fig. 7.

Since the runtime is the same for any PASS (the one machine available is always busy), no algorithm will produce a better runtime than this one. However, class S algorithms exist which construct schedules minimizing the memory required to buffer data between nodes. Using dynamic programming or integer programming, such algorithms are easily constructed.

A large grain data flow programming methodology offers concrete advantages for single processor implementations. The ability to interconnect modular blocks of code 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. The gain is somewhat analogous to that experienced in VLSI design through the use of standard cells. For synchronous systems, the penalty in runtime overhead is minimal. But a single processor implementation cannot take advantage of the concurrency in a LGDF description. The remainder of this paper is dedicated to explaining how the concurrency in the description can be used to improve the throughput of a multiprocessor implementation.

B. Constructing a PAPS

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. However, in general, the runtime can be reduced substantially by distributing the load more evenly. We show in this section how the multiprocessor scheduling problem can be reduced to a familiar problem in operations research for which good heuristic methods are available.

We assume a tightly coupled parallel architecture, so that communication costs are not the overriding concern. Furthermore, we assume homogeneity; all processors are the same, so they process a node in a SDF graph in the same amount of time. It is not necessary that the processors be synchronous, although the implementation will be simpler if they are.

A periodic admissible *parallel* schedule (PAPS) 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 ϕ is the corresponding PASS with the smallest possible period P_s, then it follows that the total number P_p of block invocations in the PAPS should be some integer multiple J of P_s. We could, of course, choose $J = 1$, but as we will show below, schedules that run faster might result if a larger J is used. If the "best" integer J is known, then construction of a good PAPS is not too hard.

For a sequential schedule, precedences are enforced by the schedule. For a multiprocessor schedule, the situation is not so simple. We will assume that some method enforces the integrity of the parallel schedules. That is, if a schedule on a given processor dictates that a node should be invoked, but there is no input data for that node, then the processor halts until these input data are available. The task of the scheduler is thus to construct a PAPS that minimizes the runtime for one period of the PAPS divided by J, and avoids deadlocks. The mechanism to enforce the integrity of the communication between blocks on different processors could use semaphores in shared memory or simple "instruction-count" synchronization, where no-ops are executed as necessary to maintain synchronicity among processors, depending on the multiprocessor architecture.

The first step is to construct an acyclic precedence graph for J periods of the PASS ϕ. A precise (class S) algorithm will be given for this procedure below, but we start by illustrating it with the example in Fig. 8. The SDF graph in Fig. 8 is neither an acyclic nor a precedence graph. Examination of the number of inputs consumed and outputs produced for each block reveals that block 1 should be invoked twice as often as the other two blocks. Further, given the delays on two of the arcs, we note that there are several possible minimum period PASS's, e.g., $\phi_1 = \{1, 3, 1, 2\}$, $\phi_2 = \{3, 1, 1, 2\}$, or $\phi_1 = \{1, 1, 3, 2\}$, each with period $P_s = 4$. A schedule that is not a PASS is $\phi_1 = \{2, 1, 3, 1\}$ because node 2 is not immediately runnable. Fig. 9(a) shows the precedences involved in all three schedules. Fig. 9(b) shows the precedences involved in two repetitions of these schedules ($J = 2$).

If we have two processors available, a PAPS for $J = 1$ (Fig. 9(a)) is

$$\psi_1 = \{3\}$$

$$\psi_2 = \{1,\ 1,\ 2\}.$$

When this system starts up, blocks 3 and 1 will run concurrently. The precise timing of the run depends on the runtime of the blocks. If we assume that the runtime of block 1 is a single time unit, the run time of block 2 is 2 time units, and the runtime of block 3 is 3 time units, then the timing is shown in Fig. 10(a). We assume for now that the entire system is resynchronized after each execution of one period of the PAPS.

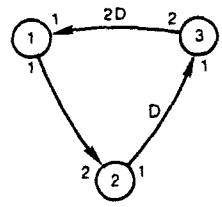

Fig. 8. An example.

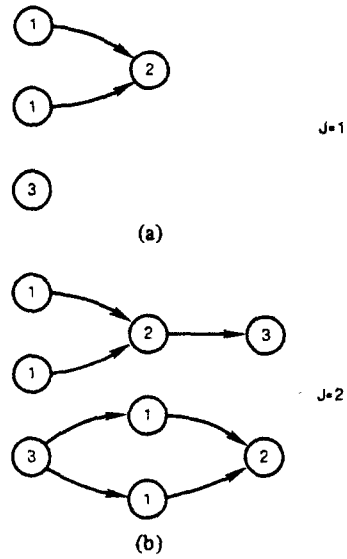

Fig. 9. Acyclic precedence graphs for (a) a minimum period ($J = 1$), and (b) a double period ($J = 2$) schedule.

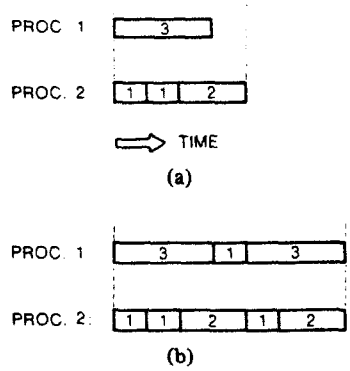

Fig. 10. Two schedules generated from the acyclic precedence graphs of Fig. 9.

A PAPS constructed for $J = 2$, using the precedence graph of Fig. 9(b), will however, perform better. Such a PAPS is given by

$$\psi_1 = \{3,\ 1,\ 3\}$$

$$\psi_2 = \{1,\ 1,\ 2,\ 1,\ 2\}$$

and its timing is shown in Fig. 10(b). Since both processors are kept always busy, this schedule is better than the $J = 1$ schedule, and no better schedule exists.

The 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 solution is combinatorial in complexity. This may not be a problem for small SDF graphs, and for large one we can use well-studied heuristic methods, the best being members of a family of "critical path" methods [38]. An early example, known as the Hu-level scheduling algorithm [39], closely approximates an optimal solution for most graphs [40], [38], and is simple. To implement this method, a *level* is determined for each node in the acyclic precedence graph, where the level of a given node is the worst case of the total of the runtimes of nodes on a path from the given node to the terminal node of the graph. The terminal node is a node with no successors. If there is no unique terminal node, one can be created with zero runtime. This node is then considered a successor to all nodes that otherwise have no successors. Fig. 11(a) shows the levels for the $J = 1$ precedence graph and Fig. 11(b) shows them for the $J = 2$ precedence graph, for the example of Fig. 8. Finally, the Hu-level scheduling algorithm simply schedules available nodes with the highest level first. When there are more available nodes with the same highest level than there are processors, a reasonable heuristic is to schedule the ones with the longest runtime first. Such an algorithm produces the schedules shown in Fig. 10, the optimal schedules for the given precedence graphs.

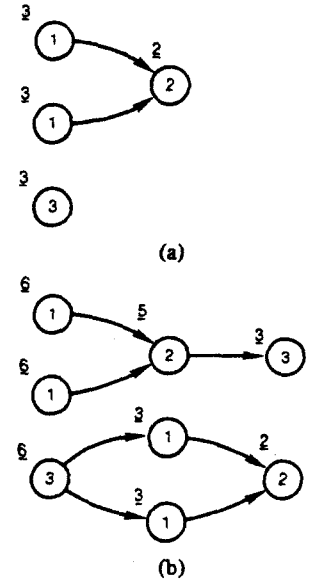

Fig. 11. The two acyclic precedence graphs of Fig. 9 with the levels indicated.

We now give a class S algorithm that systematically constructs an acyclic precedence graph. First we need to understand how we can determine when the execution of a particular node is necessary for the invocation of another node.

Consider a SDF graph with a single arc a connecting node η to node α. Assume this arc is part of a SDF graph with topology matrix Γ. The number of samples required to run α j times is $-j\Gamma_{a\alpha}$ where $\Gamma_{a\alpha}$ is the entry in the topology matrix corresponding to the connection between arc a and the node α. Of these samples, b_a are provided as initial conditions. If $b_a \geq -j\Gamma_{a\alpha}$ then there is no dependence of the jth run of α on η. Otherwise, the number of samples required of η is $-j\Gamma_{a\alpha} - b_a$. Each run of η produces $\Gamma_{a\eta}$ samples. Therefore, the jth

run of α depends on the first d runs of η where

$$d = \left\lceil \frac{-j\Gamma_{a\alpha} - b_a}{\Gamma_{a\eta}} \right\rceil \tag{7}$$

and where the notation $\lceil \cdots \rceil$ indicates the ceiling function.

Now we give a precise algorithm. We assume that we are given the smallest integer vector $\boldsymbol{v}$ in the nullspace of Γ and the "best" multiple J, so that we wish to construct an acyclic precedence graph with the number of repetitions of each node given by $J\boldsymbol{v}$. We will discuss later how we get J. Each time we add a node to the graph we will increment a counter i, update the buffer state $\boldsymbol{b}(i)$, and update the vector $\boldsymbol{q}(i)$ defined in (6). This latter vector indicates how many instances of each node have been put into the precedence graph. We let L designate an arbitrarily ordered list of all nodes in the graph.

INITIALIZATION:
$i = 0$;
$\boldsymbol{q}(0) = \mathbf{O}$;

The Main Body:
while nodes are runnable {
  for each $\alpha \in L$ {
    if α is runnable **then** {
      create the $(q_\alpha(i)+1)$th instance of the node α;
      for each input arc a on α {
        let η be the predecessor node for arc a;
        compute d using (7);
        if $d<0$ **then let** $d=0$;
        establish precedence links with the first d instances of η;
      }
      let $\boldsymbol{v}(i)$ be a vector with zeros except a 1 in position α;
      let $\boldsymbol{b}(i+1) = \boldsymbol{b}(i) + \Gamma\boldsymbol{v}(i)$;
      let $i = i+1$;
    }
  }
}

We now turn our attention to obtaining J. In the example in Figs. 8 through 11, increasing J may improve the schedule. There are also graphs where no finite J yields an optimal schedule. However, as J increases, the cost of implementing the periodic schedule increases because of the memory cost of storing the schedule. One possible technique is to increase J until each increase results in negligible improvement in the schedule. This is an issue deserving further study.

V. Limitations of the Model

We rely on experience to claim that most signal processing systems are adequately described by SDF graphs. However, the model does not describe all systems of interest. In this section, we explore some specific limitations.

A. Conditionals

The SDF model permits conditional control flow within a node, but not on a greater scale. While large-scale conditional control flow is a mainstay in general-purpose computing, it is rare in signal processing. Occasionally, however, it is required, and therefore must be supported by any practical programming system. Two types on conditional control may be required, *data dependent* or *state dependent*. An example of a system with data dependent control flow contains a node that passes its input sample to its first output if the sample is less than some threshold, and to its second output otherwise. Such a node is an *asynchronous* node because it is not possible to specify *a priori* how many samples will be produced on each output when the node is invoked. Systems with asynchronous nodes are dealt with in the next subsection.

State dependent control flow refers to such control structures as iteration where the number of iterations does not depend on data coming into the system from outside. Such iteration is easily handled by the SDF model. On a small scale, of course, it may be handled entirely within a node. On a larger scale, it may be handled by replicating a node as many times as required. The iteration is then managed by the scheduler.

B. Asynchronous Graphs

Although rare in signal processing, asynchronous graphs do exist. That is, we can conceive of nodes where the amount of data consumed or produced on the input or output paths is data dependent, so no fixed number can be specified statically. The simplest solution is to divide a graph into synchronous subgraphs connected only by asynchronous links. Then these subgraphs can be scheduled on different processors with an asynchronous communication protocol enforced in interprocessor communication. Such a protocol is generally readily available in multiprocessor systems. The asynchronous links are then handled by the scheduler as if they were connections to the outside world (discussed in the next subsection).

Another solution that is not so simple but may sometimes yield better performance in exceptional circumstances, is to implement a runtime supervisor, as done in [17]. The runtime supervisor would only handle the scheduling of entire synchronous subsystems, a much smaller task than scheduling all the nodes.

C. Connections to the Outside World

The SDF model does not adequately address the possible real-time nature of connections to the outside world. Arcs into a SDF graph from the outside world, like those shown in Figs. 1 and 2, are ignored by the scheduler. It may be desirable to schedule a node that collects inputs as regularly as possible, to minimize the amount of buffering required on the inputs. As it stands now, the model cannot reflect this, so buffering of input data is likely to be required.

D. Data Dependent Runtimes

In the construction of a PAPS, we assume the runtime of each node is known *a priori*. The runtime, however, may be data dependent. However, in hard real-time applications, it must also be bounded, independent of the data. The schedule must perform even with worst case data that causes maximum runtimes for all nodes. In this situation, there is no disadvantage to scheduling using the worst case runtimes.

VI. Conclusion

This paper describes the theory necessary to develop a signal processing programming methodology that offers pro-

grammer convenience without squandering computation resources. Programs are described as block diagrams where connections between blocks indicate the flow of data samples, and the function of each block can be specified using a conventional programming language. Blocks are executed whenever input data samples are available. Such a description is called *large grain data flow* (LGDF). The advantages of such a description are numerous. First, it is a natural way to describe signal processing systems where the blocks are second order recursive digital filters, FFT butterfly operators, adaptive filters, and so on. Second, such a description exhibits much of the available concurrency in a signal processing algorithm, making multiple processor implementations easier to achieve. Third, program blocks are modular, and may be reused in new system designs. Program blocks are viewed as black boxes with input and output data streams, so reusing a program block simply means reconnecting it in a new system. Fourth, multiple sample rates are easily described under the programming paradigm.

We describe high-efficiency techniques for converting a large grain data flow description of a signal processing system into a set of ordinary sequential programs that run on parallel machines (or, as a special case, a single machine). This conversion is accomplished by a *large grain compiler* so called because it does not translate a high-level language into a low-level language, but rather assembles pieces of code (written in any language) for sequential or parallel execution. Most DSP systems are synchronous, meaning that the sample rate of any given data path, relative to other data paths, is known at compile time. Large grain data flow graphs with such sample rate information are called synchronous data flow graphs. Given sample rate information, techniques are given (and proven valid) for constructing sequential or parallel schedules that will execute deterministically, without the runtime overhead generally associated with data flow. For the multiprocessor case, the problem of constructing a schedule that executes with maximum throughput is shown to be equivalent to a standard operations research problem with well studied heuristic solutions that closely approximate the optimum. Given these techniques, the benefits of large grain data flow programming can be extended to those signal processing applications where performance demands are so severe that little inefficiency for the sake of programmer convenience can be tolerated.

Acknowledgment

The authors gratefully acknowledge helpful suggestions from the anonymous reviewers, R. Rathbone, and R. Righter.

References

[1] *Signal Processing Peripheral*, data sheet for the S28211, AMI, Inc.
[2] *TMS32010 User's Guide*, Texas Instruments, Inc., Dallas, TX, 1983.
[3] T. Tsuda, *et al.*, "A high-performance LSI digital signal processor for communication," in *Proc. IEEE Int. Conf. Commun.*, June 19, 1983.
[4] *Digital Signal Processor*, data sheet for the uPD7720 signal processor interface (SPI), NEC Electronics U.S.A. Inc.
[5] S. Magar, D. Essig, E. Caudel, S. Marshall, and R. Peters, "An NMOS digital signal processor with multiprocessing capability," *ISSCC 85 Dig. Tech. Papers*, Feb. 13, 1985.
[6] R. C. Chapman, Ed., "Digital signal processor," *Bell Syst. Tech. J.*, vol. 60, Sept. 1981.
[7] R. N. Kershaw, *et al.*, "A programmable digital signal processor with 32b floating point arithmetic," *ISSCC 85 Dig. Tech. Papers*, Feb. 13, 1985.
[8] M. Chase, "A pipelined data flow architecture for signal processing: The NEC uPD7281," in *VLSI Signal Processing*. New York: IEEE Press, 1984.
[9] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.
[10] D. B. Paul, J. A. Feldman, and V. J. Sferrino, "A design study for an easily programmable, high-speed processor with a general-purpose architecture," Lincoln Lab, Massachusetts Inst. Technol., Cambridge, MA, Tech. Note 1980-50, 1980.
[11] W. B. Ackerman, "Data flow languages," *Computer*, vol. 15, Feb. 1982.
[12] J. B. Dennis, "Data flow supercomputers," *Computer*, vol. 13, Nov. 1980.
[13] I. Watson and J. Gurd, "A practical data flow computer," *Computer*, vol. 15, Feb. 1982.
[14] A. L. Davis, "The architecture and system method of DDM1: A recursively structured data driven machine," in *Proc. Fifth Ann. Symp. Comput. Architect.*, Apr. 1978, pp. 210–215.
[15] J. Rumbaugh, "A data flow multiprocessor," *IEEE Trans. Comput.*, vol. C-26, p. 138, Feb. 1977.
[16] R. G. Babb, "Parallel processing with large grain data flow techniques," *Computer*, vol. 17, July, 1984.
[17] D. G. Messerschmitt, "A tool for structured functional simulation," *IEEE J. Select. Areas Commun.*, vol. SAC-2, Jan. 1984.
[18] L. Snyder, "Parallel programming and the poker programming environment," *Computer*, vol. 17, July 1984.
[19] Kelly, Lochbaum, and Vyssotsky, "A block diagram compiler," *Bell Syst. Tech. J.*, vol. 40, May 1961.
[20] B. Gold and C. Rader, *Digital Processing of Signals*. New York: McGraw-Hill, 1969.
[21] B. Karafin, "The new block diagram compiler for simulation of sampled-data systems," in *AFIPS Conf. Proc.*, vol. 27, 1965, pp. 55–61.
[22] M. Dertouzous, M. Kaliske, and K. Polzen, "On-line simulation of block-diagram systems," *IEEE Trans. Comput.*, vol. C-18, Apr. 1969.
[23] 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.
[24] W. Henke, "MITSYN—An interactive dialogue language for time signal processing," Res. Lab. Electronics Massachusetts Inst. Technol., Cambridge, MA, RLE-TM-1, Feb. 1975.
[25] T. Crystal and L. Kulsrud, "Circus," Inst. Defense Anal., Princeton, NJ, CRD working paper, Dec. 1974.
[26] *TOPSIM III—Simulation Package for Communication Systems*, user's manual; Dep. Elec. Eng., Politencia di Torino, Italy.
[27] G. Kopec, "The representation of discrete-time signals and systems in program," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, May 1980.
[28] C. S. Jhon, G. E. Sobelman, and D. E. Krekelberg, "Silicon compilation based on a data-flow paradigm," *IEEE Circ. Devices*, vol. 1, May 1985.
[29] 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.
[30] R. Reiter, "A study of a model for parallel computations," Ph.D. dissertation, Univ. Michigan, Ann Arbor, 1967.
[31] J. L. Peterson, "Petri nets," *Comput. Surv.*, vol. 9, Sept. 1977.
[32] ——, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[33] T. Agerwala, "Putting Petri nets to work," *Computer*, p. 85, Dec., 1979.
[34] R. M. Karp and R. E. Miller, "Parallel program schemata," *J. Comput. Syst. Sci.*, vol. 3, no. 2, pp. 147–195, 1969.
[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. Ass. Comput. Mach.*, vol. 14, pp. 590–599, 1968.
[37] R. E. Blahut, *Fast Algorithms for Digital Signal Processing*. Reading, MA: Addison-Wesley, 1985.
[38] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 685–690, Dec., 1974.

[39] T. C. Hu, "Parallel sequencing and assembly line problems," *Operat. Res.*, vol. 9, pp. 841-848, 1961.

[40] W. H. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Trans. Comput.*, vol. C-25, pp. 1235-1238, Dec. 1975.

Edward Ashford Lee (S'80-M'86) received the B.S. degree from Yale University, New Haven, CT, in 1979, the S.M. degree from the Massachusetts Institute of Technology, Cambridge, in 1981, and the Ph.D. degree from the University of California, Berkeley, in 1986.

From 1980 to 1982 he was with Bell Laboratories, as a member of the Technical Staff of the Data Communications Laboratory, where he did exploratory work in voiceband data modem techniques and simultaneous voice and data transmission. Since July 1986 he has been an Assistant Professor in the Department of Electrical Engineering and Computer Science, University of California, Berkeley. His research interests include architectures and software techniques for programmable digital signal processors, parallel processing, real-time software, computer-aided engineering for signal processing and communications, digital communications, and quantization. He has taught a short course at the University of California, Santa Barbara, on telecommunications applications of progammable digital signal processors, has consulted in industry, and holds one patent.

Dr. Lee was the recipient of IBM and GE Fellowships and the Samuel Silver Memorial Scholarship Award. He is a member of Tau Beta Pi.

David G. Messerschmitt (S'65-M'68-SM'78-F'83) 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 Beta 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.