Pipeline Interleaved Programmable DSP's: Synchronous Data Flow Programming

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

Abstract—In the companion paper [1], a programmable architecture for digital signal processing is proposed that requires the partitioning of a signal processing task into multiple programs that execute concurrently. In this paper, a synchronous data flow programming method is proposed for programming this architecture, and programming examples are given.

Because of its close connection with block diagrams, data flow programming is natural and convenient for describing digital signal processing (DSP) systems. Synchronous data flow is a special case of data flow (large grain or atomic) in which the number of tokens consumed or produced each time a node is invoked is specified for each input or output of each node. A node (or block) is asynchronous if these numbers cannot be specified a priori. A program described as a synchronous data flow graph can be mapped onto parallel processors at compile time (statically), so the run time overhead usually associated with data flow implementations evaporates. Synchronous data flow is therefore an appropriate paradigm for programming high-performance real-time applications on a parallel processor like the processors in the companion paper. 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.

In this paper, we outline the programming methodology by illustrating how nodes are defined, how data passed between nodes are buffered, and how a compiler can map the nodes onto parallel processors. We give an example of a typically complicated unstructured application: a voiceband data modem. For this example, using a natural partition of the program into functional blocks, the scheduler is able to use up to seven parallel processors with 100 percent utilization. Beyond seven processors, the utilization drops because the scheduler is limited by a recursive computation, the equalizer tap update loop. No attempt has been made to modify the algorithms or their description to make them better suited for parallel execution. This example, therefore, illustrates that modest amounts of concurrency can be effectively used without particular effort on the part of the programmer.

I. INTRODUCTION

To achieve high performance in real-time signal processing and related numeric-intensive computations, the need to depart from the simplicity of von Neumann computer architectures is axiomatic. Pipelining and parallelism are two forms of concurrency that can take advantage of increasing VLSI complexity, but both approaches can considerably complicate the programming. It would be unacceptable to design a digital signal processing (DSP) architecture with spectacular throughput that required an army of specialists to program. In the companion paper [1], we describe an architecture for programmable DSP’s that replaces deeply pipelined architectures with what appear to be cooperating parallel processors. The modified architecture is called a pipeline interleaved processor. The technique is to interleave concurrent processes through the pipeline in such a way that each instruction in each process is completed before the next instruction in the same process is begun. This technique has been used in the peripheral processors of the CDC 6600 [2], the HEP-1 [3], [4], and an experimental architecture at Columbia University [5], all large general-purpose machines. Applied to signal processors, the technique is much simpler and promises to mitigate the difficulties associated with deep pipelining, allowing a potentially large gain in performance through deeper pipelining than is generally considered practical for programmable processors. But since pipeline difficulties are replaced with difficulties associated with parallelism, a reasonably effective solution to the parallel programming problem for modest numbers of processors is a mandatory prerequisite.

For the purposes of this paper, the pipeline interleaved processor described in the companion paper can be viewed simply as a set of synchronized parallel processors sharing memory without contention. Programming a pipeline interleaved processor is, on the surface, no different from programming any multiprocessor system. However, the fact that the application is real-time digital signal processing greatly simplifies the programming problem, opening up the possibility of conceptually simple and elegant programming techniques.

In this paper, we propose programming signal processors using a technique called synchronous data flow (SDF) [6], based on data flow languages [7]. The technique is particularly well suited to programming pipeline interleaved processors, so we describe it in that context, but it is also potentially useful for programming existing single- and multiple-DSP systems and also may be applicable to semicustom hardware description. Data flow eases the programming task by enhancing the modularity of code (allowing efficient reuse of software components [8]) and permitting algorithms to be described more naturally as block diagrams. Concurrency is immediately evident in the data flow description, so the concurrent resources (processor slices) can be used effectively. The theoretical basis for SDF applied to digital signal processing is given
in [6], with only a brief summary of the results given here. Further detail and more examples are given in [9].

A. Data Flow

For concurrency, a program is broken into subtasks, which are then automatically, semiautomatically, or manually scheduled onto parallel processors or processor slices, either at compile time (statically) or at run time (dynamically). Automatic breakdown of an ordinary sequential program is an appealing concept [10], but the success of existing techniques is limited. If the programmer provides the breakdown as a natural consequence of the programming methodology, we should expect more efficient use of concurrent resources.

Dividing a program into subtasks is not new to programmers; structured programming has insisted on it throughout most of the history of computers. The usual technique for breaking up a program is to divide it into subroutines, functions, or procedures. But these are all ill suited to parallel execution. Furthermore, procedures are not usually a natural way of describing DSP algorithms; functional blocks interconnected with signal flow paths are more suitable.

DSP systems are often described using block diagrams consisting of functional blocks connected by data flow paths. An example that we will discuss extensively, a voiceband data modem, is illustrated in Fig. 1. The figure illustrates an implementation of a 2400 bit/s, 600 baud frequency division multiplexed full duplex voiceband data modem with band-splitting filters, and a fractionally spaced passband adaptive equalizer [11]-[13]. Block diagrams are often viewed as descriptions of hardware realizations, which are usually inherently highly concurrent. Block diagram descriptions are modular, meaning that once a block is defined it is easily reused. They can also be hierarchical, where a block may itself represent another network, yielding programs with much of the elegance of structured programming. Mixed mode programming, where frequently used blocks are programmed in assembly language and less common blocks are programmed in a high-level language, is possible. Hardware descriptions can also be mixed with assembly language or higher-level functional descriptions, in principle. Concurrency can be automatically extracted without requiring undue programming effort.

The block diagram of Fig. 1 is a data flow graph. Data flow is a hardware and software technique used for concurrent computation [7], [14]-[17]. The fundamental premise behind data flow graphs is that each node (or block) represents a function that can be invoked whenever input data are available to it. Because the program execution is controlled by the availability of data, data flow programs are said to be data driven [18]. 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.

Functions may be elemental (addition, multiplication, etc.) or nonelemental (digital filters, FFT units, modulators, phase-locked loops, etc.), and the directed arcs represent paths taken by successive data samples. The complexity of the functions (or the "granularity") will determine the amount of parallelism available. If the granularity is at the level of signal processing subsystems (second-order sections, butterfly units, etc.), the paradigm is called large grain data flow (LGDF) [7], [19]-[21]. Otherwise, it is called atomic data flow, from the Greek word atomos, meaning indivisible. Atomic data flow is obviously a special case of LGDF. Data flow specifications are related to the experimental computer language techniques known as applicative languages, single assignment languages, and functional languages [7].

Historically, the data flow principle has been used to design computer architectures for parallel computation [14], [16], [20]. Such architectures are specialized to evaluate data flow graphs quickly. A comprehensive survey of data flow architectures is given by Srinivas [22]. So-called "data flow languages" have been developed for programming data flow machines such as VAL [23] and ld [24], but these are actually functional languages that can be easily translated into data flow graphs [7]. Our proposal here is different in that the language itself is graphical, and we propose to compile it for execution on a pipeline interleaved processor, which is not a data flow architecture.

Common objections leveled against data flow center around the overhead required to schedule and synchronize the hardware or software blocks. However, it has been shown that very low overhead is required for most types of synchronization required in digital signal processing [6]. In particular, for synchronous DSP systems, in which sample rates are known and are rational multiples of one another, it is possible to statically (at compile time) schedule blocks onto parallel processors. In Fig. 1, sample rates are shown explicitly. A data flow graph from which relative sample rates can be determined (the absolute numbers are not important, only their relation to one another is) is called a synchronous data flow graph (SDF graph).

A precise definition is required. A node in a data flow graph is a function that fires when a sufficient number of
input samples (tokens) are available to perform a computation. When a node is invoked, it will consume a fixed number of new tokens on each input path. A node is said to be synchronous if we can specify a priori the number of tokens consumed on each input and the number of tokens produced on each output each time the node is invoked. These numbers are part of the node definition. A synchronous node 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. For example, a digital filter node has one input and one output, and the number of tokens consumed or produced is one. A 2:1 decimator node would also have one input and one output, but would consume two tokens for every token produced. An SDF graph is a network of synchronous nodes, as in Fig. 2(b). The voiceband data modem of Fig. 1 is reproduced in Fig. 3 with the number of tokens consumed and produced associated with each input and output. We will say more about this, but first we review related models for representing DSP algorithms.

B. History and Related Models

The rich collection of related methods for defining DSP systems is worth reviewing.

1) Block Diagram Languages: Many so-called block diagram languages have been developed to permit users to implement signal processing algorithms more easily, at least in simulation, rather than real time. Block diagrams are an appropriate description of DSP systems. Some examples are BLODI [25], PATSI [26], BLOODIB [27], LOTUS [28], DARE [29], MITSYN [30], Circus [31], and TOPSIM [32]. But these simulators are based on the principle of next-state simulation [26], [33] 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 relatively rare in digital signal processing, multiple sample rates are common, stemming from the frequent use of decimation and interpolation. The technique we propose in this paper shares the appropriateness of block diagram languages, and it handles multiple sample rates easily. Furthermore, a pipeline interleaved processor architecture combined with a data flow programming paradigm easily handles systems with limited asynchrony.

2) LGDF: One way to avoid the limitations of next-state simulators, but retain the convenience of expressing algorithms as block diagrams, is to use LGDF, as done in BLOSIM [34], [35], a single-processor digital signal processing simulator that naturally accommodates multiple sample rates and asynchronous systems. Dynamically allocated linked lists are used to buffer data between data flow nodes. The scheduling is straightforward; a simple heuristic is used to construct a list of all nodes approximately ordered according to their precedences. Nodes are invoked in the order specified by this list. When a node is invoked, its code checks its input buffers to determine whether there are adequate data and runs until such data are exhausted. If a node has no inputs, it runs for a specified number of cycles. The system continues forever or until a deadlock occurs, in which no node in the list has data on its input buffers. This type of control is effectively dynamic because the order of execution of node functions is determined at run time, in response to changing conditions. The ordering of the list of nodes approximately according to their precedences helps ensure that most nodes are ready to run when their input buffers are checked, but it is not required for correct execution. Another way of ensuring this is to maintain a list of runnable nodes, which is expanded each time a node generates enough data to run another node.

Such dynamic control mechanisms are traditional in data flow implementations, but they involve considerable overhead. First of all, in BLOSIM, buffers are unbounded in size so that a node does not need to check its output buffers to see if there is room. The amount of memory required for the buffers is difficult to determine, and even for fully synchronous systems may depend on the order in which the nodes are invoked (the schedule). In programmable DSP’s, where memory is a scarce commodity, such inefficiencies must be avoided. Second, the supervisory overhead can be substantial. Checking input buffers, and possibly trying many nodes before one is finally run, consumes program execution time, also a scarce commodity in real-time signal processing applications. On the other hand, this mechanism supports asynchronous communication between nodes, and no elaborate scheduling is required for correct execution of the program. This gives it a generality that is extremely useful for simulation, but probably too costly for real-time implementations on DSP’s.

3) Signal Flow Graphs: Some work with synchronous data flow descriptions has centered around signal flow graphs, originally used to describe linear single-sample-rate systems [36]. Crochiere and Oppenheim [36] systematically translate signal flow graphs into acyclic precedence graphs. The method is quite simple, based on the observation that arcs with unit delays are not precedence
relationships, but arcs without delays are. Equivalently, arcs with delays can be broken and replaced with I/O operations. But the method does not consider the repetitive nature of a desired schedule and therefore does not always properly indicate long-term precedences when more than one delay is present in a loop. It also does not support multiple-sample rates. In spite of these deficiencies, Braffman [37] and Zeman [38] both recommend this method to obtain acyclic precedence graphs, and then both apply critical path methods to the scheduling problem.

4) Single-Sample Rate SDF: The term “signal flow graph” is often used to describe single-sample-rate data flow graphs, regardless of whether the system is linear. Multiprocessor implementations of algorithms specified this way have been explored at the Georgia Institute of Technology [39]. Algorithms are assumed to operate repeatedly on an infinite stream of data, and optimal periodic schedules can be systematically generated. One of the implementations proposed is called skewed single-instruction multiple data (SSIMD) [40], [41], in which a set of processing elements perform the same functions, but skewed in time with respect to one another. This allows much more flexibility than traditional SIMD and is particularly well suited to simple signal processing tasks.

The technique has been generalized to what are called cyclostatic systems [39], in which the function performed by each processor is periodic, but each processor can perform different functions. Optimal scheduling can be done for such systems, as long as the hardware can be tailored to the application. The admirable work on cyclostatic systems has some deficiencies in our application, however. Primarily, it has no provision for multiple sample rates, thus restricting the range of applications.

5) Reduced Dependence Graphs: Explored recently at Stanford University, 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 [42], [43]. The resulting description is close to a data flow graph with the sample rate again restricted to be uniform throughout the system. Reduced dependence graphs are used to describe regular iterative algorithms, which can then be mapped onto processor arrays. This approach seems particularly well suited to descriptions of well-structured algorithms to be implemented in systolic arrays. The range of applications is again excessively limited for our objectives.

6) Computation Graphs: Computation graphs are LGDF graphs which, like signal flow graphs and reduced dependence graphs, are restricted to modeling synchronous systems. Unlike these previous models, however, computation graphs naturally accommodate multiple sample rates. The differences between SDF graphs and computation graphs are so minor as to be insignificant, but our use of the model differs significantly.

Computation graphs were introduced in 1966 by Karp and Miller [44] and were further explored by Reiter [45]. The analysis concentrated on fundamental theoretical considerations with, for example, a proof 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 concentrated 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 [46] and Reiter [47], but the restrictions imposed on the model are excessive. More recently, we have applied the model to parallel programming synchronous digital signal processing [6]. A scheduler maps the synchronous data flow description onto parallel processors at compile time, so no run-time control is required beyond the traditional sequential control in the processors. The SDF paradigm gives the programmer a natural interface for easily constructing well-structured signal processing programs, with evident concurrency, and the scheduler statically maps this concurrency onto parallel processors.

7) Petri Nets: Computation graphs have been shown to be a special case of Petri nets [48]–[50] or vector addition systems [51]. These more general models can be used to describe asynchronous systems, but implementations generally require expensive dynamic control flow.

II. Implementation of SDF Programming

The implementation of SDF for pipeline interleaved processors requires

1) a way to specify the topology of an application (the SDF graph),
2) a way to specify the functions associated with each node, and
3) a systematic method for mapping these specifications onto processor slices.

The SDF graph can be generated using a graphical interface [52], making prototyping DSP systems that use only standard nodes particularly easy. In this paper, we assume that the code defining the function of a node is written in the assembly language of the processor. The design of a suitable high-level language is a problem we do not address here. We concentrate instead on the third problem, for which it is not obvious that a satisfactory solution exists.

Implementing the signal processing system described by an SDF graph requires buffering the tokens passed between nodes and scheduling nodes so that they are executed when data are available. Our goal is a compiler that translates node definitions and an SDF graph into efficient sequential code for a parallel processor. Note that our compiler is not translating a high-level language into machine code, although this could ultimately be part of its function. It begins by scheduling, using the techniques in [6], and proceeds with code generation.
A. Buffers

Each arc in the SDF graph corresponds to an FIFO queue that can be implemented as a circular buffer. Circular buffers are easily supported using the modulo addressing described in the companion paper [1]. The size of each buffer depends on the schedule. To see this, consider the trivial system shown in Fig. 4. Assume for simplicity that we are to schedule this system onto a single processor slice; the buffering problem is the same in the multiprocessor case. If the schedule is a periodic repetition of the sequences \( \{ \text{in}, \text{fir4}, \text{out} \} \), then the buffers need only store one new token at a time and can therefore be of length one. If the schedule is a periodic repetition \( \{ \text{in}, \text{in}, \text{fir4}, \text{fir4}, \text{out}, \text{out} \} \), then buffers of length two are required.

It is desirable to be able to access not just the new tokens in a buffer, but also past tokens [35]. The buffer "buf1" can then be used to implement the delay line of the four-tap FIR filter. The programmer therefore does not need to implement data structures that can easily be provided by the circular buffering mechanism.

Each buffer is a set of contiguous memory locations plus two pointers. Both pointers are set up by the scheduler for modulo-mode autoincrement-decrement. The write pointer is used by the node that puts data into the buffer. Before this node is invoked, the write pointer should point to the next location to be written (an empty location in the buffer). The read pointer is used by the node taking data from the buffer. Before that node is invoked, the read pointer should point to the token in the input buffer four tokens before the oldest token of interest in the buffer. Thus, in Fig. 4, before invoking fir4, the read pointer of "buf1" points to the last token in the delay line of the FIR filter. The code within a node must increment all read or write pointers that it uses to indicate consumption or production of tokens.

This use of the buffer is illustrated in Fig. 5. A buffer of length four and the schedule \( \{ \text{in}, \text{fir4}, \text{out} \} \) are assumed. The first buffer state, labeled initial, shows the write pointer pointing to the first location in the buffer and the read pointer pointing to the second. We assume the buffer is initially full of zero tokens, so the FIR filter delay line is initialized to contain zeros. Invoking the in node causes a new sample to be written into the first location. The pointer is incremented by one, indicating that one new token at a time can be of length one. If the schedule is a periodic repetition \( \{ \text{in}, \text{in}, \text{fir4}, \text{fir4}, \text{out}, \text{out} \} \), then buffers of length two are required.

This is all the code defining the node. It assumes that on entry to the subroutine,

1) the address register "ry1" contains the read pointer for the input buffer, and "ry2" contains the write pointer for the output buffer;

2) "ry1" points to the token in the input buffer four iterations old;

3) "ry2" points to the next empty location in the output buffer; and

4) "rx1" points to the coefficient for the oldest token.

These assumptions are part of the definition of the node.
Hence, in addition to the subroutine above, declarations like the following are required as part of the definition of the node:

```plaintext
input name, ry1;
output name, ry2;
param coefficients, rx1.4;
```

The names can be used to distinguish inputs and outputs when there are more than one of each. Notice that the definition of the node is not affected by the length of the buffers and therefore is independent of the schedule. This same node can be used in the graph shown in Fig. 6 where, because of the interpolator, a buffer of length four at the input to fir4 will not work. If the schedule is \{in, interp, fir4, fir4, out, out\}, then a buffer of length five is required.

To initialize the system, the buffers must be defined by allocating memory setting the read and writer pointers to their initial values. The compiler, after scheduling to determine the length of the buffers, generates the following code to initialize the buffers “buf1” and “buf2” in Fig. 4. Suppose that the schedule is \{in, fir4, out\}. Then the following π code will set up the buffers:

```plaintext
buf1: ram(4);
buf1_write: ram(1);
buf1_read: ram(1);
buf2: ram(1);
buf2_write: ram(1);
buf2_read: ram(1);
```

This code is generated by the compiler and executed only once. Notice that this code can be trimmed down by observing that the read/write pointers to a length one buffer are always the same. We will ignore such obvious optimizations for now.

When the program is running and collecting input samples, to invoke the node fir4 the registers “rx1,” “ry1,” and “ry2” must be set. Each invocation of fir4 therefore proceeds as follows (this code is generated by the compiler):

```plaintext
ry1 = buf1_read;
ry2 = buf2_write;
rx1 = coefficients;
call fir4;
buf1_read = ry1;
buf2_write = ry2;
```

Set read pointer.
Set write pointer.
Set parameter pointer.
Call the subroutine.
Update the read pointer.
Update the write pointer.

Pseudo-op to allocate four memory locations.
Pseudo-op to allocate one memory location.
Pseudo-op to allocate one memory location.
Pseudo-op to allocate one memory location.
Pseudo-op to allocate one memory location.
Pseudo-op to allocate one memory location.

Point to first location of length 4 buf.
Write to buffer write pointer location.
Point to second location of length 4 buf.
Write to buffer read pointer location.
Point to length 1 buf.
Write to buffer write pointer location.
Point to length 1 buf.
Write to buffer read pointer location.

B. Delays

In [6], it is observed that delays are managed quite simply in SDF. A delay is a property of the arc connecting two nodes. That is, if there is a unit delay on the arc connecting node A to node B, then the nth token consumed by B will be the \((n - 1)th\) token produced by A. The first token consumed by B is therefore not produced by A at all, but is rather part of the initial state of the buffer. A delay on an arc is exactly equivalent to an initial sample on the arc, implying that delays are simply introduced in the initialization of the buffer.

Consider the example in Fig. 4. A unit delay on the arc from in to fir4 would imply that fir4 can be invoked before in. Indeed, on a single processor, we would now have three possible schedules with unity blocking factor: \{in, fir4, out\}, \{fir4, in, out\}, and \{fir4, out, in\}. Depending on whether the scheduler elects to run fir4 before in,
the buffer will be initialized in one of the two states shown in Fig. 7, where the zero indicates the initial sample put there by the compiler.

Delays are required in directed loops in an SDF graph. Consider the graph in Fig. 8. Without the unit delay, neither node can be invoked. The unit delay implies an initial sample on the arc from A to B and therefore implies that B can be scheduled. A suitable schedule is therefore a periodic repetition of \{ B, A \}.

C. Static Scheduling

Static scheduling of SDF graphs is described in [6]. That paper establishes criteria for the correctness of an SDF graph. In particular, for a graph to be correct, it must have a periodic admissible schedule. A periodic schedule is said to be admissible if the amount of data in the buffers remains bounded and nonnegative with infinite repetition of the schedule.

There are two types of defective graphs. The first type has directed loops with insufficient delays. Two examples are shown in Fig. 9. The first, Fig. 9(a) has a delay-free loop, known to be noncomputable. The second has a delay in the loop, but is immediately deadlocked because node B requires two inputs to be invoked.

The second type of defective graph has sample rate inconsistencies. An example is shown in Fig. 10. If node A is run first, node B can be run second, followed by node C. Now only node A can be run again, so the schedule is repeated. Each time through the periodic schedule \{ A, B, C \}, however, one more token is left on the arc from A to C. A buffer on this arc would require infinite memory to continue this periodic schedule in perpetuity. No schedule overcomes this problem. The programmer must be informed of this error.

In [6], necessary and sufficient conditions for correctness of an SDF graph are given. A broad class of algorithms designated class S (for sequential) algorithms will find a periodic admissible schedule if one exists, solving the single-processor scheduling problem. A pipeline interleaved processor is a multiprocessor, however.

For the multiprocessor case, a class S algorithm is given for translating an SDF graph into an acyclic precedence graph for one or more periods of a periodic schedule. Consider the SDF graph of Fig. 11(a). Because of the changes in sample rate, node A must be scheduled twice as frequently as nodes B and C. The unity blocking factor schedule therefore includes two instances of A and one each of B and C. The graph in Fig. 11(b) illustrates the precedences. Node C has no precedences because the delay on the arc leading into it implies that the node can be invoked immediately. The first instance of node A similarly has no precedence, but we assume that the second instance depends on the first instance. This is only true if A has a state, which is equivalent to a self-loop in the SDF graph. The self-loops are shown explicitly. In [6], the same example is considered without this assumption. Node B can only be invoked after both instances of node A have been invoked. The graph in Fig. 11(b) can be used
to construct a schedule for multiple processors. Indeed, given such an acyclic precedence graph, the scheduling problem reduces to the well-studied assembly line problem. General schedule length minimization algorithms are \textit{NP complete} [53], but a large family of \textit{critical path methods} offer simple heuristics that can be shown to perform extremely well. These methods give preference to the path through the precedence graph with the most computation. If all nodes in Fig. 11(b) are assumed to have the same computation time, then the critical path is $A_1 \rightarrow A_2 \rightarrow B$. One critical path method is used quite successfully to schedule the modem example in the next section.

The schedule is sometimes better if more than unity blocking factor is considered [6]. In the example, we may wish to construct a schedule in which one period includes four instances of $A$ and two instances each of $B$ and $C$. The corresponding precedence graph is shown in Fig. 11(c). Scheduling $n$ times the minimum period is called \textit{blocking}, and $n$ is called the \textit{blocking factor}. Fig. 11(c) shows the precedence graph for a blocking factor of 2. If the run times are all equal, the critical path is $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4 \rightarrow B_2$. Any advantage to blocking depends on the number of processors available and the run times of the nodes.

\section{III. Performance}

Just as pipeline bubbles degrade the performance of a pipelined processor, scheduling imperfections will degrade the performance of a synchronous data flow program on a pipeline interleaved processor. The amount of degradation depends on the amount of concurrency in the graph. Smaller granularity will generally lead to enhanced concurrency, but smaller granularity may also require more resources devoted to buffering data. The optimal tradeoff is application dependent. Algorithms with feedback loops can also limit the amount of concurrency available.

Fig. 12 illustrates the 2400 bit/s voiceband data modem. This high-level description is shown as a more detailed SDF graph in Fig. 14. Complex numbers are communicated between nodes as two successive tokens, so many of the nodes produce and consume two tokens on each input and output. The inventory of nodes is shown in Fig. 15. The run time, measured in instructions, for each node includes the setup time, the subroutine call, and the subsequent writes. The run time for a four-tap FIR filter would therefore be 13. The run times represent only one of many possible implementations and can undoubtedly be improved. However, the specific numbers are not as important as the knowledge that we are testing a real (as opposed to contrived) practical example. We expect similar performance for different implementations.

The total amount of computation is 2062 instructions in one period. Notice that one period processes 16 input samples and produces one output sample (which in this case contains four bits of interest). Using the methods described in [6], we constructed the acyclic precedence graph and a schedule for various numbers of processors. With no additional delays beyond those in the feedback loops in Fig. 3, the length of the critical path in the acyclic precedence graph is 642 instructions. Therefore, with unity blocking factor, no schedule can achieve a period of less than 642 instruction cycles. For five or more processors, the period is within 5 percent of this bound. For five processors, however, the processor utilization is a mediocre 62 percent.

\begin{figure}[h]
\centering

\begin{subfigure}{0.4\textwidth}
\centering
\begin{tikzpicture}
  \node (A) at (0,0) {$A$};
  \node (B) at (1,0) {$B$};
  \node (C) at (2,0) {$C$};
  \draw (A) -- (B);
  \draw (B) -- (C);
\end{tikzpicture}
\caption{(a) An SDF graph with delays on its feedforward cutsets. (b) Its precedence graph with unity blocking factor. Note the lack of concurrency.}
\end{subfigure}

\begin{subfigure}{0.4\textwidth}
\centering
\begin{tikzpicture}
  \node (A) at (0,0) {$A$};
  \node (B) at (1,0) {$B$};
  \node (C) at (2,0) {$C$};
  \node (D) at (0,-1) {$D$};
  \draw (A) -- (B);
  \draw (A) -- (D);
  \draw (B) -- (C);
\end{tikzpicture}
\caption{(b) An SDF graph with unity blocking factor, showing much more concurrency. This computation is pipelined.}
\end{subfigure}
\caption{(a) An SDF graph. (b) Its precedence graph with unity blocking factor.}
\end{figure}

A. \textit{A Voiceband Data Modem}

To verify the efficacy of the programming methodology, we examined a real application, a voiceband data modem, rather than relying on standard DSP bench marks. The standard bench marks, IIR and FIR filters and FFT's, are not a good test of the programming method. Its benefits are most evident in complicated, relatively unstructured applications. We selected the modem application because one of us has implemented and fully tested such a modem using the Bell Labs DSP-20. By using essentially the same algorithms, we avoid the need for real-time testing to verify the validity of the implementation. Since the \pi has not been implemented in hardware, real-time testing is not feasible.

Fig. 3 illustrates the 2400 bit/s voiceband data modem.
Fig. 14. A detailed synchronous data flow graph for the voiceband data modem applications. Delays that do not affect the algorithms have been inserted on all feedforward cutsets.

A particularly easy way to improve the schedule is to put delays in the feedforward cutsets (pipelining). Fig. 14 shows such delays. With these delays, the performance of the scheduler is dramatically improved. The minimum period with unity blocking factor is reduced from 642 to 277. The limiting factor now is a directed loop in the graph [54]. For seven or more processors, the achieved period is within 5 percent of the optimum. In Fig. 16, we show the schedule for eight processors.

In Fig. 17, the processor utilization as a function of the number of π slices is shown. This plot should be interpreted with caution. We have plotted the percentage of time within one cycle that the processors are busy, but this assumes that the next cycle begins as soon as the previous cycle ends. Given real-time constraints, this may not be true, and the actual utilization may be lower. In other words, the utilization shown is the maximum utilization if the system is operated at the highest sample rate possible. With this caveat, essentially full utilization is maintained with up to seven π slices. Recall from the companion paper that the computation per unit time of the π processor increases somewhat less than linearly with the number of π slices, suggesting that the maximum throughput is achieved for seven slices.

Fig. 15. An inventory of blocks used in the voiceband data modem. The second column indicates whether the block is likely to be standard in a programming system.
elaborate techniques; additional delays will not help because the length 256 critical path becomes the 16 repetitions of any one of the biquads.

B. 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.

1) 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 of conditional control may be required: data dependent or state dependent. An example with data-dependent control contains a node that passes its input token to its first output if the value of the token is less than some threshold and to its second output otherwise. Such a node is asynchronous because it is not possible to specify a priori how many tokens 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. Alternatively, it may be handled using the data-driven properties of data flow, as shown in Fig. 18. The iteration is then managed by the scheduler.

2) Asynchronous Data Flow Graphs: Although rare in signal processing, asynchronous data flow 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 statistically. Implementations of a timing recovery PLL might have such asynchrony, for example.

The simplest solution for an asynchronous data flow graph is to divide the graph into synchronous subgraphs connected only by asynchronous links (or not connected at all). These subgraphs can then be scheduled on different processor slices with an asynchronous communication protocol enforced in interprocessor communication. The asynchronous links are handled by the scheduler as if they were connections to the outside world (discussed in the next subsection). To support asynchronous communication between processor slices, hardware support for semaphores would be desirable. The usual synchronization problems associated with such systems of semaphores [57] can be avoided because processor slices are actually completely synchronous, even if the programs running on them are not. Instructions from different processor slices are always interleaved through the pipeline in the same order, and a single system clock controls all discrete events.

Asynchronous signal processing, which is sometimes quite difficult in today's DSP's, is actually much easier on a pipeline interleaved processor. Consider a system with trivial asynchrony consisting of a modem receiver and a modem transmitter, operating asynchronously, with no connection between them. Such a system is tricky to implement in real time on a DSP. A data flow graph of the modem is actually disconnected since no data flow between the transmitter and receiver. The disconnected parts can be scheduled separately onto different processor slices, greatly simplifying the problem. The only potential difficulty is in the I/O. If the same I/O ports are used (the $\pi$ provides only one set of I/O ports, so they would have to be shared), then the nodes that manage the collection or delivery of data to the outside would be asynchronous.

Another solution that is not so simple but may sometimes yield better performance is to implement a run-time supervisor, as done in [35]. The run-time supervisor would handle only the scheduling of entire synchronous subsystems, probably a much smaller task than scheduling all the nodes. We are investigating this technique further.

3) Connections to the Outside World: The SDF model
does not adequately address the real-time nature of connections to the outside world. Arcs into an SDF graph from the outside world 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 extra buffering of input data is likely to be required. However, the buffering needs to handle only data that arrive during one period of a periodic schedule. Assuming a synchronous system, this is a deterministic number of samples that the scheduler can precompute. The I/O techniques mentioned in the companion paper are therefore adequate.

4) Data-Dependent Run Times: In the construction of a schedule for each of the processor slices, the execution time of each node is assumed known a priori [6]. Knowledge of the execution times permits the scheduler to optimize for maximum throughput and to synchronize processor slices by only strategically inserting the appropriate number of no-ops in the code. This is called instruction count synchronization.

While the execution time of individual instructions is invariably independent of data in DSP's for real-time applications, the run time for a given node may be data dependent. In hard real-time applications, the run time must be bounded, with the bound independent of the data. The schedule must perform even with worst case data that cause maximum run times for all nodes. In this situation, there is no disadvantage to scheduling assuming the worst case run times.

IV. CONCLUSIONS

We have proposed a pipeline interleaved approach for the design of high-performance programmable DSP's in the companion paper [1]. Deep pipelining is used to gain maximum hardware advantage, but the problems generally associated with such pipelining are avoided by making the pipelining invisible to the programmer. Instead, the programmer sees a set of parallel processors (processor slices) that share memory. A synchronous data flow programming paradigm is proposed for programming such a processor. It has been shown [6] that a compiler can generate efficient code for each processor slice, given an SDF description of an application. The advantages of such a description are as follow.

- Concurrency can be automatically enhanced.
- Multiple sample rates can be supported.

In addition, extensions to support limited asynchrony are possible.

The main disadvantage of an SDF programming paradigm is that processing resources may be lost due to scheduling inefficiencies. This is a fundamental problem with parallel implementations. The amount of the loss is dependent on the granularity and structure of the application. For the voiceband data modem application, up to seven π slices can be used with no loss due to scheduling.

Hardware features that have been included in the π to support this type of programming are

- modulo autoincrement/decrement addressing modes (for circular buffers);
- data-independent execution times for all instructions;
- and
- memory semaphores with a full/empty discipline (for asynchronous systems).

Further work includes designing and testing a hardware prototype of the π, finding systematic techniques for describing and managing asynchrony, and developing techniques for minimizing the amount of memory used for buffering. In addition, more efficient code generation techniques are apparently possible and are being explored.

REFERENCES


