DATA PARALLELISM IN GRAPHICAL SIGNAL FLOW REPRESENTATIONS OF ALGORITHMS

by

Edward A. Lee

Memorandum No. UCB/ERL M92/110

21 September 1992

DATA PARALLELISM IN GRAPHICAL SIGNAL FLOW REPRESENTATIONS OF ALGORITHMS

by

Edward A. Lee

Memorandum No. UCB/ERL M92/110

21 September 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

August 13, 1992

DATA PARALLELISM IN GRAPHICAL SIGNAL FLOW REPRESENTATIONS OF ALGORITHMS

Edward A. Lee

Department of Electrical Engineering and Computer Science

University of California

Berkeley, California 94720

ABSTRACT

Signal flow graphs with dataflow semantics have been used in signal processing system simulation, algorithm development, and real-time system design. Dataflow semantics implicitly expose function parallelism by imposing only a partial ordering constraint on the execution of functions. They are also capable of representing data parallelism. This paper shows how the synchronous dataflow model [17] can be used to graphically define algorithms while exposing their data parallelism to a compiler or hardware synthesis tool. A "recursive iterator" notation is introduced to achieve scalable graphical representations for certain algorithms. The SDF model is then extended to multidimensional streams to represent and exploit data parallelism in certain signal processing applications. The method is by no means general, but appears to have broad enough applicability to be useful. The resulting semantics are related to reduced dependence graphs used in systolic array design and to multidimensional streams in the declarative language Lucid. They are more distantly related the stream-oriented functional languages Silage, and streams in the dataflow language Id and the "synchronous" languages Lustre and Signal.

1.0 Motivation

To get competitive real-time implementations of signal processing applications, it is necessary to exploit design-time information about the algorithms. No computation or control function should be deferred to run-time if it can be performed at the time software is compiled or

The author gratefully acknowledges the support of the National Science Foundation, the Semiconductor Research Corporation, AT&T Bell Labs, Philips, and Rockwell.

hardware is designed. Hence, specification languages should reveal to the compiler or hardware synthesis tool as much static information as possible.

This applies as much to control flow as data operations. Control-flow operations performed at run-time interfere with optimization possibilities, particularly exploitation of parallelism. In software, conditional branches are a key problem. Compiler writers and computer architects know that anywhere from 1/10 to 1/3 of all instructions are conditional branches, even for applications with totally predictable control flow. I contend that alternative representations of algorithms could make analysis of the flow of control much easier, and hence make parallelizing compilers much more effective, particularly for signal processing applications.

In this paper I show how a graphical programming environment like those commonly used for signal processing can be adapted to expose data parallelism. In particular, we set the following objectives for the syntax and semantics of graphical programs:

1. *Hierarchy.* This is essential to manage complexity, and is standard in graphical design environments today.
2. *Scalability.* In particular, the graphical definition of a computational module must be invariant under problem size.
3. *Static flow of control where possible.* When flow of control is predictable, the semantics must be simple enough that a compiler can completely analyze it.
4. *Exploitable function and data parallelism.* Graphical languages have been used to exploit function but not data parallelism. We need both.

As a simple example, an FIR filter should be modular so that it can be used as a parameterized unit. It should be a hierarchical block that can be used without concern for its internal structure. Its internal structure, however, for applications requiring fast execution, should expose all exploitable parallelism to a compiler. So the filter should not be expressed using C-style "for" or "do-while" loops, which are difficult to analyze for static control flow and data parallelism. Although compilers have gotten better at such analysis, experience shows that languages with simpler semantics such as Silage [13] (used in the Mentor/EDC DSPstation) or the flowgraph language

used in the Comdisco Procoder [21] can be compiled more efficiently. But in a graphical language, data parallelism in the FIR filter should be exposed without "hard-wiring" the filter order into the graphical specification, as would be required in most currently available graphical programming environments. A notable exception is the Mentor/EDC DSPstation which supports a mixed graphical and textual functional specification that is scalable. The approach proposed in this paper goes much further, however.

2.0 Background

Much of the motivation for this work stems from our design environments Gabriel [5] and Ptolemy [7]. In these systems, signal processing algorithms are described graphically and either simulated or used to synthesize a real-time implementation. We will periodically refer in this paper to actors of the type used in these systems. Gabriel uses only one model of computation, called synchronous dataflow (SDF). Ptolemy is much more flexible, although in this paper we will only refer to its dataflow capability.

2.1 Dataflow

In the dataflow model of computation, a program can be represented as a graph, where the nodes represent actors (functions or operators with functional semantics) and the arcs represent paths taken by tokens (which carry data) [11]. The actors can be fine grain (atomic machine operations) or large grain (functions of arbitrary complexity) [27]. In figure 1, each token produced by

Figure 1. A graphical dataflow program.

actor A is consumed by actor B. Each arc represents a semi-infinite ordered set of tokens. The ordering need not be chronological (tokens may be produced or consumed out of order) as long as data precedences implied by the ordering are respected. This is the source of much of the data parallelism we will exploit in this paper. A given token on an arc is produced once and consumed

once, although it may be referenced more than once during the firing of B, and may even be referenced in subsequent firings (the actor may access "past" tokens, as done in [5][7]). The firings of actors are ordered in the same sense as the tokens on an arc are ordered. Actors may fire out of order in time, or may have several simultaneous firings on different processors, as long as data precedences are satisfied.

The diamond on the path between B and C is a *delay,* which we can interpret as an initial token on the arc. It implies that the n-th token consumed by C is the $(n-1)$-th produced by B. An initial value for the delay must be specified. A macro actor can have *state,* which we will model simply as a self-loop with a delay.

2.2 Functional, Stream-Oriented Languages

In figure 1, an arc represents a stream. Equivalent textual representations are easy to devise. In Silage [13], a textual signal processing language with similar semantics, every symbol in the language represents a stream. Instead of "actors" the language has functions and operators with functional semantics. Consider the following segment of a program:

```
x = 1 + x@1;
x@@1 = 0.0;
```

The symbol "x" represents an infinite stream. The language is declarative, so the order of the statements makes no difference. The language has the notion of a global cycle, and a simple reference to a symbol "x" can be thought of as referring to the "current value" of x, or to the entire stream. The syntax "x@1" is similar to the delay in figure 1, and is related to the "fby" or "cby" operator in Lucid [2][26] and the "->" operator in Lustre [9]. It refers to the previous value, or equivalently to the stream shifted by one token. The syntax "x@@1" initializes the stream x with a value 0.0 so that the first reference to x@1 is defined. An equivalent graphical syntax is shown in figure 2. The actor labeled "1" simply produces a continuous stream of ones. The initial value

of the delay is shown adjacent to its diamond. In this paper, we will use a graphical syntax with only occasional pointers to equivalent textual syntax.

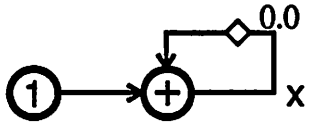

Figure 2. A graphical equivalent to the Silage program given in the text.

The semantics of streams gets much more complicated when streams with different rates are permitted. Suppose, for example, that for every token in stream X there are two tokens in stream Y. Since streams are infinite, this relationship cannot be ignored. One approach is to associate with each stream a "clock," as done in Lustre [9], Signal [3], and to some extent, Silage [13]. The clock of Y has twice as many ticks per unit time, and only every second token in Y aligns with a token in X. For the most flexible of these languages, Signal, a powerful algebraic methodology has been developed to reason about relationships between clocks [3].

Our approach is different. The clocks are replaced by relative rates of production and consumption. There is no concept of simultaneity of tokens (tokens in different streams lining up). We argue that our approach is more in-keeping with the spirit of dataflow [10][11], and is more easily parallelized at compile time. As explained below, it can support multidimensional streams, combining thus the best features of Lucid with the best features of the "synchronous" languages Lustre and Signal.

2.3 Synchronous Dataflow

For several years, we have been developing software environments for signal processing that are based on a special case of dataflow that we call synchronous dataflow (SDF) [17]. The Gabriel [5] and Ptolemy [7] programs use this model. It has also been used in Aachen [23] in the COSSAP system and at Carnegie Mellon [22] for programming the Warp. As above, SDF graphs consist of networks of actors connected by arcs that carry data. But the actors are constrained to produce and consume a fixed integer number of tokens on each input or output path when they fire [17]. The term "synchronous" refers to this constraint, and arises from the observation that the rates of production and consumption of tokens on all arcs are related by rational multiples. Unlike

the "synchronous" languages Lustre and Signal, however, there is no notion of clocks. Tokens form ordered sequences, with only the ordering being important.

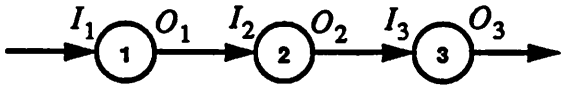

Figure 3. A simple synchronous dataflow graph.

Consider the simple graph in figure 3. The symbols adjacent to the inputs and outputs of the actors represent the number of tokens consumed or produced. Most SDF properties follow from the *balance equations*, which for the graph in figure 3 are

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

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

The symbols r_i represent the number of firings (repetitions) of an actor in a cyclic schedule. Given a graph, the compiler solves the balance equations for these values. Given this solution, a precedence graph can be automatically constructed specifying the partial ordering constrains between firings [16]. From this precedence graph, good compile-time schedules can be automatically constructed [24][25].

SDF allows compact and intuitive expression of predictable control flow and is easy for a compiler to analyze. Consider for instance the nested iteration described in figure 4. The balance

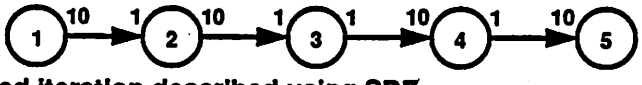

Figure 4. Nested iteration described using SDF.

equations can be collected into matrix form

$$\Gamma \vec{r} = \vec{o} \tag{3}$$

where $\vec{o}$ is the zero vector, and the topology matrix is given by

$$\Gamma = \begin{bmatrix} 10 & -1 & 0 & 0 & 0 \\ 0 & 10 & -1 & 0 & 0 \\ 0 & 0 & 1 & -10 & 0 \\ 0 & 0 & 0 & 1 & -10 \end{bmatrix}. \tag{4}$$

The smallest integer solution to the balance equations is

$$\vec{r}^T = [1\ 10\ 100\ 10\ 1], \tag{5}$$

which indicates that for every firing of actor 1, there will be 10 firings of actor 2, 100 of 3, 10 of 4, and 1 of 5.

The application of this model to multirate signal processing is described in [6]. An application to vector operations is shown in figure 5, where two FFTs are multiplied. Both function and

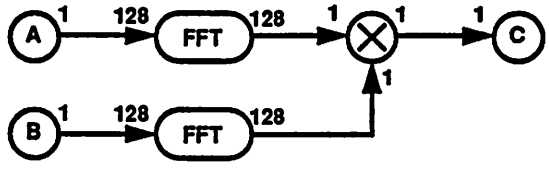

Figure 5. Application of SDF to vector operations.

data parallelism are evident in the precedence graph that can be automatically constructed from this description. That precedence graph would show that the FFTs can proceed in parallel, and that all 128 invocations of the multiplication can be invoked in parallel. Furthermore, the FFT might be internally specified as a dataflow graph (see below), permitting exploitation of parallelism within each FFT as well. The Ptolemy system can use this model to implement overlap-and-add or overlap-and-save convolution, for example.

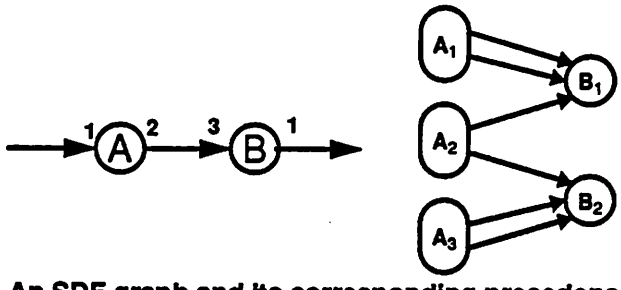

Figure 6. An SDF graph and its corresponding precedence graph.

More interesting control flow can be specified using SDF. Figure 6 shows two actors with a 2/3 producer/consumer relationship. The precedence graph is shown on the right. From this, we can construct the sequential schedule (A_1, A_2, B_1, A_3, B_2), among many possibilities. This sched-

ule is not a simple nested loop, although schedules with simple nested loop structure can be constructed systematically [4]. Notice that unlike the "synchronous" languages Lustre and Signal, we do not need the notion of clocks to establish a relationship between the stream into actor A and the stream out of actor B.

2.4 The Token-Flow Model

Loosely speaking, the balance equations require that in the long run, the number of tokens produced on an arc must equal the number of tokens consumed. This concept has been extended to handle actors that are not SDF, or actors where the number of tokens produced or consumed is not fixed [8][19]. Consider the SWITCH and SELECT actors in figure 7. These route tokens conditional on a Boolean

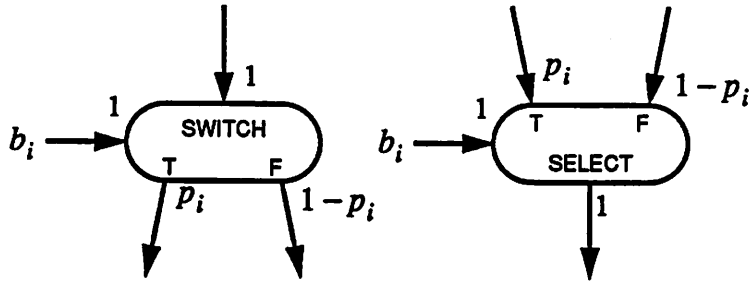

Figure 7. Dynamic dataflow actors annotated with the expected number of tokens produced or consumed per firing as a function of p_i, the probability that a token from the stream b_i is TRUE.

input. The number of tokens produced by the SWITCH or consumed by the SELECT is not known, so in the token flow model that number is replaced with a symbolic placeholder. The balance equations now have symbolic unknowns, and the solution is found in terms of these unknowns. The conceptually simplest interpretation for these unknowns is a probabilistic one, as explained in the caption to figure 7. However, other interpretations are more useful, as explained in [8].

3.0 Relationship with Prior Work

A minor contribution of this paper is to show how for some algorithms, the SDF model can expose data parallelism. Below we will show how recursive graphical descriptions can be used to define some algorithms in a scalable way.

The principal contribution of this paper is to extend SDF to a multidimensional model in order to exploit data parallelism. As such, the work is related to the large body of literature on synthesizing systolic arrays from regular iterative algorithms [15]. The multidimensional model is related to reduced dependence graphs commonly used in this field, but differs in that (1) the model is a dataflow model rather than a direct specification of precedence relationships, (2) there is no need for a homogeneous index space, and (3) the emphasis is on a programming methodology with concise, scalable graphical syntax.

Programming for data parallelism has been accomplished in the past using single-assignment, functional, or dataflow languages. The most relevant of these to this paper is Lucid [26], the language with the best developed support for multidimensional streams. Like our model in this paper, Lucid is designed to have clear semantics that a compiler can analyze, but it does not have our emphasis on compile-time scheduling.

Skillcorn [26] argues that streams and functions on them are a natural way to model reactive and distributed systems. Reactive systems include signal processing systems, but also include servers and operating systems. They operate continuously and produce and consume unbounded message sequences. Hence, languages designed for operating on such sequences, languages such as Id [1], Lucid [2][26], Sisal [20], Lustre [9], and Signal [3] support streams. In Sisal and Id, streams are lists fashioned after lists in Lisp, but with non-strict semantics. This means simply that a function operating on the stream can begin operating on it before the entire stream has been computed.

4.0 Recursive Iterators[1]

The SDF graph in figure 4 is scalable in that the graphical structure does not depend on the amount of iteration. The "10" entries could easily be replaced with system parameters. However, the structure of the iteration is rather simple. More interesting examples are, of course, more difficult to express. Sometimes, a graphical representation we call a "recursive iterator" is ideal. It is scalable, concise, and elegant. We will illustrate it by giving a scalable graphical representation of an analysis/synthesis multirate filter bank and the decimation-in-time FFT, both using the SDF model. These two examples illustrate most of the features of interest.

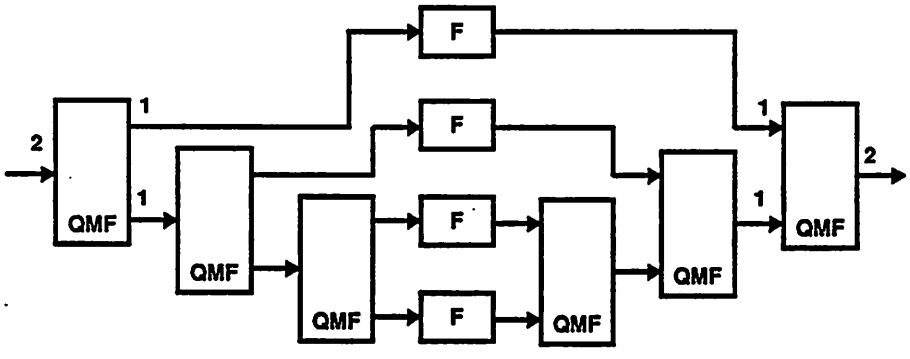

Figure 8. An analysis/synthesis filter bank under the SDF model. The depth of the filter bank, however, is hard-wired into the representation.

Consider the system shown in figure 8. It shows a multirate signal processing application: an analysis/synthesis filter bank with harmonically spaced subbands. The signal coming in at the left is split by matching highpass and lowpass filters (labeled "QMF"). These are decimating polyphase FIR filters, so for every two tokens consumed on the input, one token is produced on the output. The left-most QMF only is labeled with its SDF parameters, but the others behave the same way. The output of the lowpass side is further split by a second QMF, and the lowpass output of that by a third QMF. The boxes labeled "F" represent some function performed on the decimated signal (such as quantization). The QMF boxes to the right of these reconstruct the signal using matching polyphase interpolating FIR filters. There are four distinct sample rates in figure 8

1. The author extends thanks to Jeff Robinson and Keith Rouse of Star Semiconductor for helpful discussions pertaining to this section.

with a ratio of 8 between the largest and the smallest. A scalable parallelizable representation for the decimating and interpolating FIR filters is given below, but for now we will just worry about the representation of the system at this level of abstraction, where the filters are considered an atomic unit.

The graphical representation in figure 8 is useful for developing intuition, and exposes exploitable parallelism, but it is not so useful for programming. The depth of the filter bank is hard-wired into the graphical representation, so it cannot be conveniently made into a parameter of a filter-bank module. To solve this problem, we propose the representation in figure 9. A hierar-

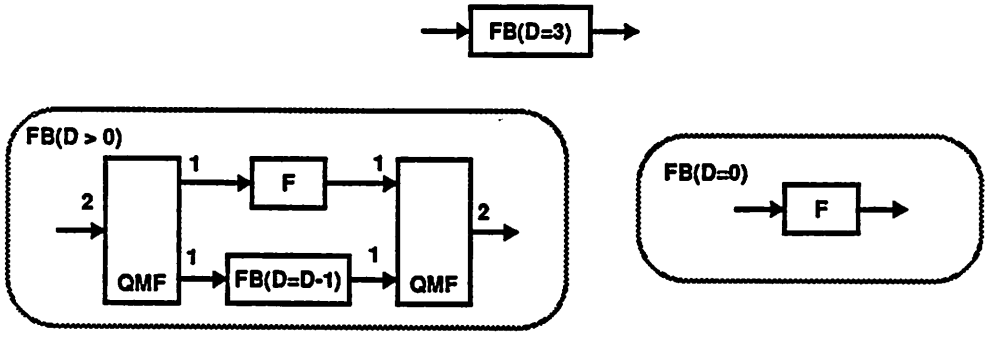

Figure 9. A "recursive iterator" representation of the filter bank application. This representation is scalable.

chical block called "FB" (for filter bank) is defined, and given a parameter D, the depth. For D>0 the definition of the block is at the left. It contains a self-reference, with the parameter of the inside reference changed to D-1. When D=0, the definition at the right is used. The system at the top, consisting of just one block, labeled "FB(D=3)", is exactly equivalent to the representation in figure 8, except that the graphical representation does not now depend on the depth. The graphical

recursion in figure 9 can be expanded completely at compile time, exposing all exploitable parallelism, and incurring no unnecessary run-time overhead.

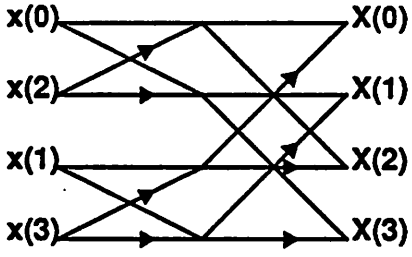

Figure 10. A fourth-order decimation-in-time FFT shown graphically. The order of the FFT, however, is hard-wired into the representation.

A fourth-order decimation-in-time FFT is shown in figure 10. Again, as a graphical program, this representation is extremely inconvenient. A scalable representation using recursive iterators is shown in figure 11. When the FFT order N is a power of two greater than zero, the def-

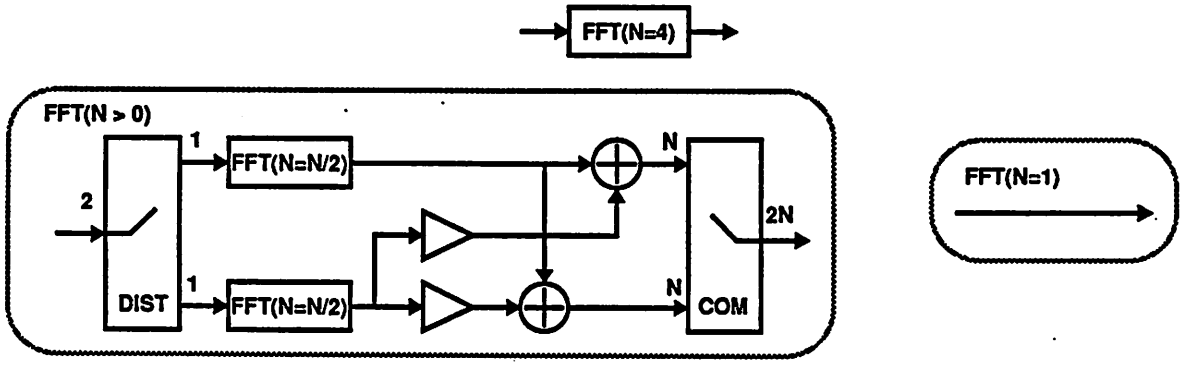

Figure 11. A recursive-iterator representation of the decimation-in-time FFT. This representation is scalable.

inition of the FFT block is shown on the left. The first block is a "distributor", which collects two input tokens and send the first one to the top output and the second one to the bottom output. The resulting decimated-in-time sequences are sent to recursive instances of the FFT block with order N replaced by N/2. These recursive references will be expanded at compile-time until the order is one, at which time the trivial definition on the right will be used. The tokens returned by recursive FFT instances are fed into a butterfly network. The triangles represent fixed gains (the "twiddle factors"). Note that the value of these will depend on the parameter N at a given depth of the

recursion as well as the position of a particular firing in the firing sequence. Expressing these dependencies requires considerable functionality in the expression language used. The "COM" actor (for "commutator") collects N tokens on each input, then outputs them sequential, the top inputs first. The resulting outputs will be in the same order as in figure 10. The DIST and COM actors are used regularly in Ptolemy [7].

The representation in figure 11 satisfies all our objectives, and has the side benefit that it is structured recursively, much like the derivation of the FFT algorithm itself. But there still may be improvements. Instead of the butterfly operations explicitly specified as shown in figure 11, we could use an FFT of some small order, set by a parameter. This would allow us to control the granularity through a parameter instead of being constrained to the finest available granularity.

5.0 Multidimensional Dataflow

The standard dataflow model suffers from the limitation that its streams are one dimensional. Although a multidimensional stream can be embedded within a one dimensional stream, it may be awkward to do so. In particular, compile-time information about the flow of control may not be immediately evident. The multidimensional SDF model is a straightforward extension of one-dimensional SDF. Figure 12 shows a trivially simple two-dimensional SDF graph. The number of tokens produced and consumed are now given as M-tuples. Instead of one balance equation for each arc, there are now M. The balance equations for figure 12 are

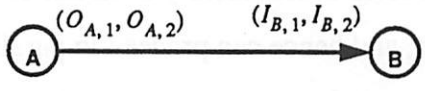

Figure 12. A simple MD-SDF graph.

$$r_{A,1}O_{A,1} = r_{B,1}I_{B,1} \tag{6}$$

$$r_{A,2}O_{A,2} = r_{B,2}I_{B,2} \tag{7}$$

These equations should be solved for the smallest integers $r_{X,i}$, which then give the number of repetitions of actor X in dimension i.

5.1 Application to Multidimensional Signal Processing

As a simple application of MD-SDF, consider a portion of an image coding system that takes a 40x48 pixel image and divides it into 8x8 blocks on which it computes a DCT. At the top level of the hierarchy, the dataflow graph is shown in figure 13. The solution to the balance equa-

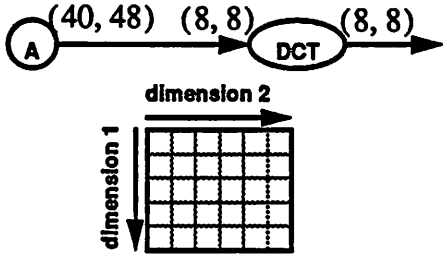

Figure 13. An image processing application in MD-SDF.

tions is given by

$$r_{A,1} = r_{A,2} = 1, r_{DCT,1} = 5, r_{DCT,2} = 6. \quad (8)$$

A segment of the index space for the stream on the arc connecting actor A to the DCT is shown in the figure. The segment corresponds to one firing of actor A. The space is divided into regions of tokens that are consumed on each of the five vertical firings of each of the 6 horizontal firings. The precedence graph constructed automatically from this shows that the 30 firings of the DCT are independent of one another, and hence can proceed in parallel. Distribution of data to these independent firings can be automated.

5.2 Flexible Data Exchange

Application of MD-SDF to multidimensional signal processing is obvious. There are, however, many less obvious applications. Consider the graph in figure 6 above. Note that the first firing of A produces two samples consumed by the first firing of B. Suppose instead that we wish for firing A_1 to produce the first sample for each of B_1 and B_2. This can be obtained using MD-SDF as shown in figure 14. Here, each firing of A produces data consumed by each firing of B, resulting in a pattern of data exchange quite different from that in figure 6. The precedence graph in figure 14 shows this. Also shown is the index space of the tokens transferred along the arc, with

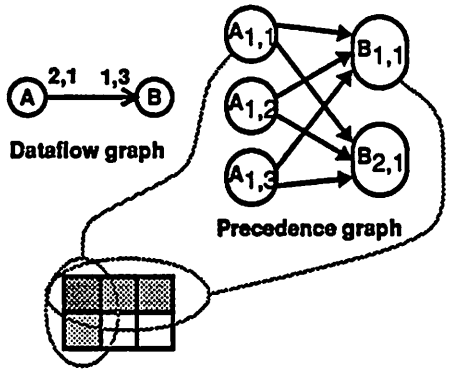

Figure 14. :Data exchange in an MD-SDF graph.

the shaded regions indicating the tokens produced by the first firing of A and consumed by the first firing of B.

A DSP application of this more flexible data exchange is shown in figure 15. Here, ten

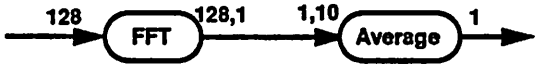

Figure 15. Averaging successive FFTs using MD-SDF.

successive FFTs are averaged. Averaging in each frequency bin is independent and hence may proceed in parallel. The ten successive FFTs are also independent, so if all input samples are available, they too may proceed in parallel.

5.3 Computing Inner Products

Consider the problem of repeatedly computing an inner product on a stream of vectors. This can be easily elaborated into an FIR filter, although for conciseness we will stick to the generic inner product. In particular, suppose we wish to express the inner product at its finest level of granularity, and further that we require the graphical representation to have a structure that is independent of the size of the vectors. To express this using 1D-SDF, we might try the configuration shown in figure 16. Actors A and B each supply vectors of length 8 by producing 8 tokens when they fire. The small white diamond is a "delay", which in a dataflow context is simply an initial, zero-valued token on the arc. The actor with the downward arrow is a "downsample." It simply consumes 8 tokens and outputs one of them, discarding the rest. This configuration will

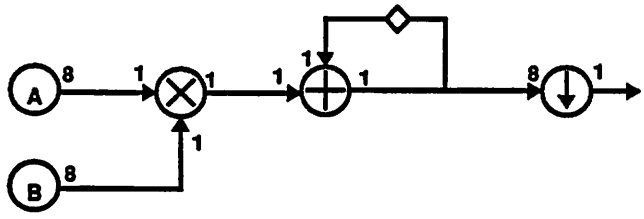

Figure 16. An attempt to use 1D-SDF to repeatedly compute inner products.

correctly compute the first inner product, but when the second set of vectors are generated by repeated firings of A and B, the delay on the feedback path will not be re-initialized. Hence, subsequent inner products will be incorrect.

I have previously proposed a mechanism called "resetting delays" that solve this problem in the context of 1D-SDF [18]. However, the MD-SDF model provides a more elegant solution.

A delay in MD-SDF in associated with a tuple as shown in figure 17. It can be interpreted

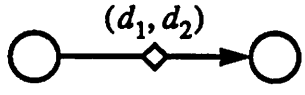

Figure 17. A delay in MD-SDF is multidimensional.

as specifying boundary conditions on the index space. Thus, for 2D-SDF, as shown in the figure, it specifies the number of initial rows and columns. It can also be interpreted as specifying the direction in the index space of a dependence between two single assignment variables, much as done in reduced dependence graphs [15].

Using MD-SDF delays, the repeated inner product can be specified as shown in figure 18. The only significant difference between this and figure 17 is the multidimensional delay. Its effect is illustrated schematically in figure 18, where the index space for the output of the delay is shown. The shaded area is the initial condition specified by the delay.

5.4 Mixing Dimensionality

Note that in figure 18, 2D and 1D-SDF are mixed. We use the following rule to avoid any ambiguity:

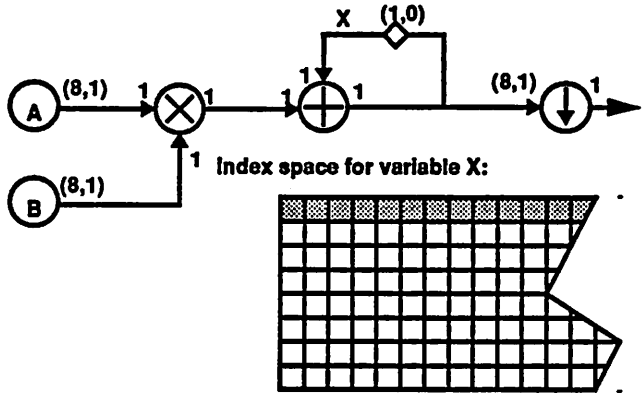

Figure 18. Repeated inner products in MD-SDF.

- The dimensionality of the index space for an arc is the maximum of the dimensionality of the producer and consumer. If the producer or the consumer specifies fewer dimensions than those of the arc, the specified dimensions are assumed to be the lower ones (lower number, earlier in the M-tuple). Hence, the two graphs in figure 19 are equivalent.

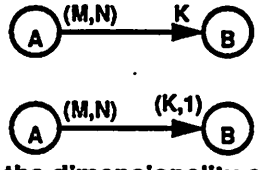

Figure 19. Rule for augmenting the dimensionality of a producer or consumer.

We can specify a comparable rule for delays:

- If the dimensionality specified for a delay is lower than the dimensionality of an arc, then the specified delay values correspond to the lower dimensions. The unspecified delay values are zero. Hence, the graphs in figure 20 are equivalent.

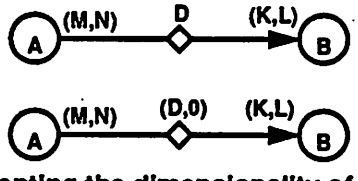

Figure 20. Rule for augmenting the dimensionality of a delay.

5.5 Polyphase Decimating FIR Filters

The inner product of figure 18 can be used to concisely describe a polyphase decimating FIR filter while exposing all the parallelism in the algorithm. A 1D-SDF specification of such a filter might look like figure 21. The input signal is divided into three phases, each of which is dis-

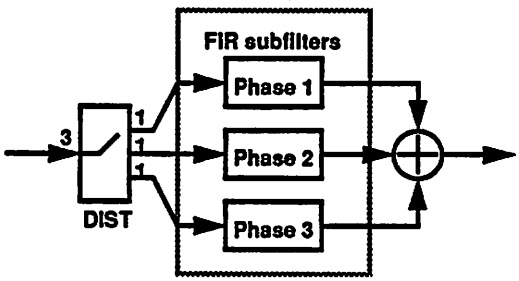

Figure 21. A 1D-SDF specification of a polyphase decimating FIR filter. This description does not expose all the parallelism and has the decimation ratio graphically hard-wired into it.

tributed to a subfilter, and the outputs of the subfilters are added [12]. As a graphical description of the algorithm, however, there are two major deficiencies. If we are restricted to 1D-SDF, then the FIR subfilters must be implemented as atomic actors, in which case the description does not expose all the parallelism in the algorithm. Moreover, the representation has the decimation ratio of three graphically hard-wired into it. This means it cannot easily be made a parameter of the filter.

MD-SDF can be used to solve both these problems, as shown in figure 22. The inner product from figure 18 is used as a module. The "Last N" actor consumes one token and produces a 1xN array composed of the token consumed and the previous N-1 tokens consumed on previous firings. The "matrix constant" actor has obvious functionality, and need not involve any run-time activity. It supplies the filter coefficients with one phase per row. The transpose actor transposes the input array. We will elaborate below on the run-time implications of these actors.

5.6 Matrix Multiplication

As another example, consider a fine-grain specification of matrix multiplication. Suppose we are to multiply an LxM matrix by an MxN matrix. In a three dimensional index space, this can

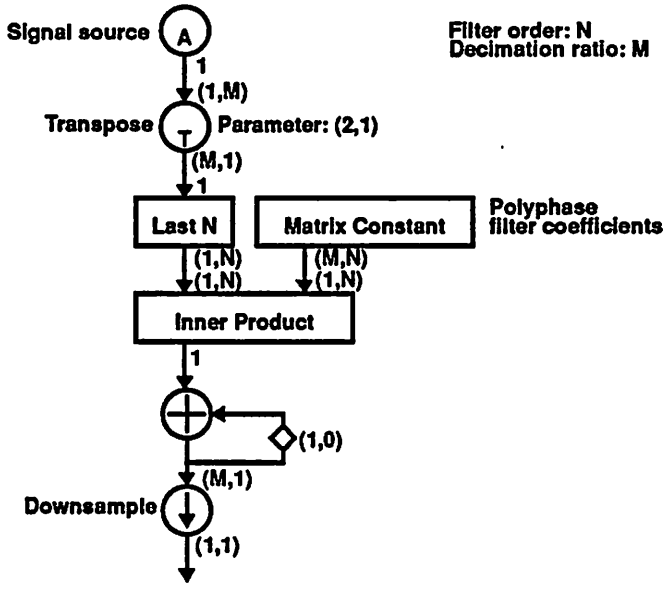

Figure 22. A polyphase decimating FIR filter expressed using MD-SDF. This representation exposes all the parallelism in the algorithm.

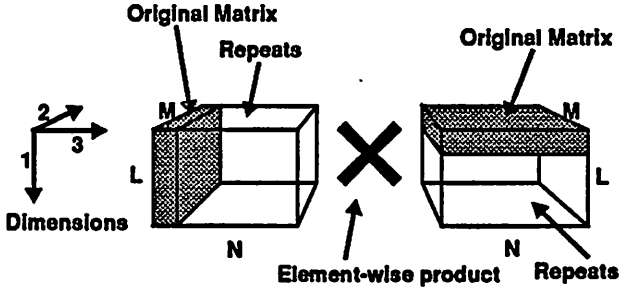

Figure 23. Matrix multiplication represented schematically.

be accomplished as shown in figure 23. The original matrices are embedded in that index space as shown by the shaded areas. The remainder of the index space is filled with repetitions of the matrices. These repetitions are analogous to assignments often needed in a single-assignment specification to carry a variable forward in the index space. An intelligent compiler need not actually copy the matrices to fill an area in memory. The data in the two cubes is then multiplied element-wise, and the resulting products are summed along dimension 2. The resulting sums give the LxN matrix product. The MD-SDF graph implementing this is shown in figure 24. The key actors used for this are:

Upsample: In specified dimension(s), consumes 1 and produces N, inserting zero values.

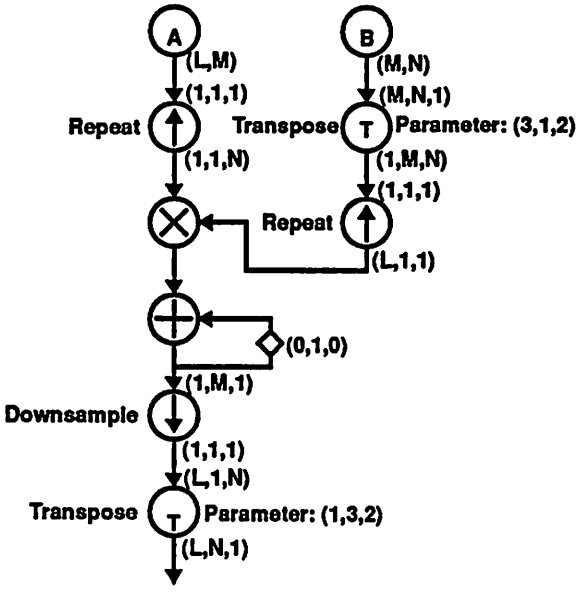

Figure 24. Matrix multiplication in MD-SDF.

Repeat: In specified dimension(s), consumes 1 and produces N, repeating values.

Downsample: In specified dimension(s), consumes N and produces 1, discarding samples.

Transpose: Consumes and M-dimensional block of samples and outputs them with the dimensions rearranged.

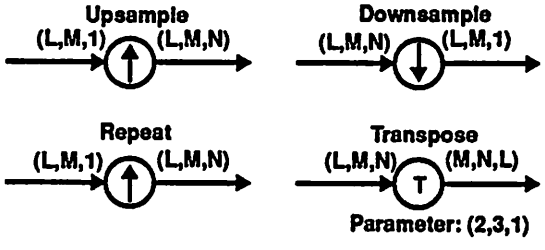

Figure 25. Some key MD-SDF actors that affect the flow of control.

These are identified in figure 25. Note that all of these actors simply control the way tokens are exchanged and need not involve any run-time operations. Of course, a compiler then needs to understand the semantics of these operators.

5.7 Polyphase Interpolating FIR Filters

The matrix multiplication of figure 24 can be used to describe a polyphase interpolating FIR filter [12] as shown in figure 26. This uses the matrix multiply from figure 24 as a module. It

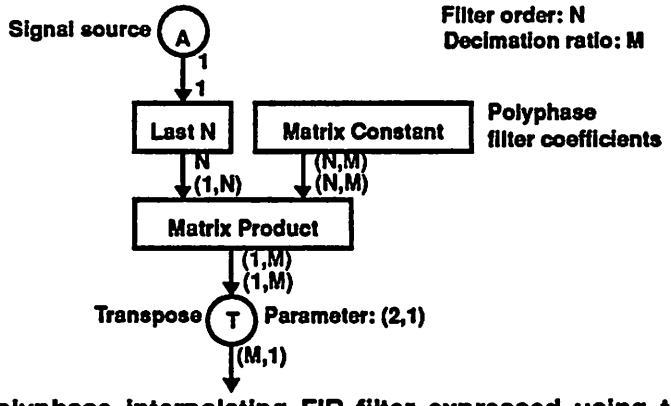

Figure 26. A polyphase interpolating FIR filter expressed using the MD-SDF model. All the parallelism in the algorithm can be automatically exploited.

also uses a new actor, the "matrix constant," which repeatedly supplies the polyphase filter coefficients. Again, a good compiler will hopefully have no run-time activity associated with this actor.

5.8 Run-Time Implications

Several of the actors we have used perform no computation, but instead control the way tokens are passed from one actor to another. In principle, a smart compiler can avoid run-time operations altogether, unless data movement is required to support parallel execution. We set the following objectives for a code generator using this language:

Upsample: Zero-valued samples should not be produced, stored, or processed.

Repeat: Repeated samples should not be produced or stored.

Last-N: A circular buffer should be maintained and made directly available to downstream actors.

Downsample: Discarded samples should not be computed (similar to dead-code elimination in traditional compilers).

Transpose: There should be no run-time operation at all, just compile-time bookkeeping.

It is too soon to tell how completely these objectives can be met.

5.9 State

For large-grain dataflow languages, it is desirable to permit actors to maintain state information. From the perspective of their dataflow model, an actor with state information simply has a self-loop with a delay. Consider the three actors with self loops shown in figure 27. Assume, as is

Figure 27. Three macro actors with state represented as a self-loop.

common, that dimension 1 indexes the row in the index space, and dimension 2 the column, as shown in figure 13. Then each firing of actor A requires state information from the previous row of the index space for the state variable. Hence, each firing of A depends on the previous firing in the vertical direction, but there is no dependence in the horizontal direction. The first row in the state index space must be provided by the delay initial value specification. Actor B, by contrast, requires state information from the previous column in the index space. Hence there is horizontal, but not vertical dependence among firings. Actor C has both vertical and horizontal dependence, implying that both an initial row and an initial column must be specified. Note that this does imply that there is no parallelism, since computations along a diagonal wavefront can still proceed in parallel. Moreover, this property is easy to detect automatically in a compiler. Indeed, all modern parallel scheduling methods based on projections of an index space [15] can be applied to programs defined using this model.

5.10 Asynchronous Actors

The token flow model, which permits SWITCH and SELECT actors, can be easily extended to multiple dimensions by simply allowing symbolic placeholders inside the M-tuples giving the number of samples produced and consumed by an actor. This is necessary to use multi-dimensional dataflow over non-rectangular index spaces. However, we have a great deal of work to do yet before a practical programming language making use of this can be devised.

6.0 Caveats

A graphical programming model based on dataflow that supports multidimensional streams has been outlined. However, we have only defined a language in sufficient detail to illustrate some simple examples. It is not clear at this point that a language based on these principles will be easy to use. Certainly the matrix multiplication program in figure 24 is not very readable. Algorithms with less regular structure will only be more obtuse. This difficulty will be exacerbated when a multidimensional DF language based on the token flow model is developed. However, the analytical properties of programs expressed this way are compelling. Parallelizing compilers and hardware synthesis tools should be able to do extremely well with these programs without relying on runtime overhead for task allocation and scheduling. We conclude, therefore, that further investigation is certainly warranted. At the very least, the method looks promising to supplement large-grain dataflow languages, much like the GLU "coordination language" makes the multidimensional streams of Lucid available in large-grain environment [14]. It may lead to special purpose languages, but could also ultimately form a basis for a language that, like Lucid, supports multidimensional streams, but is easier to analyze, partition, and schedule at compile time.

7.0 REFERENCES

[1] Arvind and J. D. Brock, "Resource Managers in Functional Programming," *J. of Parallel and Distributed Computing*, Vol. 1, No. 5-21, 1984

[2] E. A. Ashcroft, "Proving Assertions about Parallel Programs," *J. of Computer and Systems Science*, Vol. 10, No. 1, pp. 110-135, 1975.

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

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

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

[6] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Multirate Signal Processing in Ptolemy", *Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, Toronto, Canada, April, 1991.

[7] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, to appear, 1993.

[8] J. Buck and E. A. Lee, "The Token Flow Model," presented at *Data Flow Workshop*, Hamilton Island, Australia, May, 1992.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A Declarative Language for Programming Synchronous Systems," *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, Germany, January, 1987.

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

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

[12] F. J. Harris, "Multirate FIR Filters for Interpolating and Desampling," in *Handbook of Digital Signal Processing*, Academic Press, 1987.

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

[14] R. Jagannathan and A. A. Faustini, "The GLU Programming Language," Tech. Report SRI-CSL-90-11, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, November 1990.

[15] S. Y. Kung, *VLSI Array Processors*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

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

[17] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow" *IEEE Proceedings*, September, 1987.

[18] E. A. Lee, "Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages", in *VLSI Signal Processing III*, Ed. R. W. Brodersen and H. S. Moscovitz, IEEE Press, New York, 1988.

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

[20] J. McGraw, "Sisal: Streams and Iteration in a Single Assignment Language", *Language Reference Manual*, Lawrence Livermore National Laboratory, Livermore, CA 94550.

[21] D. G. Powell, E. A. Lee, W. C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *Proceedings of ICASSP*, San Francisco, March, 1992.

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

[23] S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," in *Proc. of the Int. Conf. on Application Specific Array Processors*, IEEE Computer Society Press, August 1992.

[24] G.C. Sih, E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures", to appear, *IEEE Trans. on Parallel and Distributed Systems*, 1992.

[25] G. C. Sih and E. A. Lee, "Declustering: A New Multiprocessor Scheduling Technique," to appear in *IEEE Trans. on Parallel and Distributed Systems*, 1992.

[26] D. B. Skillcorn, "Stream Languages and Data-Flow," in *Advanced Topics in Dataflow Computing*, ed. L. Bic and J.-L. Gaudiot.

[27] P. A. Suhler, J. Biswas, K. M. Korner, J. C. Browne, "TDFL: A Task-Level Dataflow Language", *J. on Parallel and Distributed Systems*, 9(2), June 1990.