[8] 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.

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

[10] 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.

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

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

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

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

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

6.0 Caveats

A 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 18 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 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. 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] E. A. Ashcroft and R. Jagannathan, “Operator Nets,” in *Proc. IFIP TC-10 Working Conf. on Fifth-Generation Computer Architectures,* North-Holland, The Netherlands, 1985.

[4] 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.

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

[6] 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.

[7] 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.

A data-driven or demand-driven style of computation can ameliorate such problems. We will now show, however, that neither data-driven nor demand driven execution is a panacea. Consider the two SDF graphs in figure 21. *Source* actors, such as A and B, have no inputs, and hence are always enabled. They model internally generated token streams, or external inputs, which have side effects. Modeling such external inputs was a principal motivation for the development of stream data types in Id [1]. *Sink* actors, such as E and F can similarly model program outputs. Suppose that $M \neq N$. If a data-driven style of execution is used at run time, ignoring the information about the number of tokens produced and consumed, then some additional control is required to keep A or B from producing an unbounded number of tokens that will accumulate in memory. That additional control is not provided by the data-driven model. A demand-driven style of execution solves the problem for the A-B-C graph, but encounters the same problem with the D-E-F graph. Again, additional control is needed. A typical approach in data-driven execution is to throttle a producer when its tokens are not being consumed, but this approach incurs run-time overhead.

SDF, MD-SDF, and the dynamic extensions based on the token flow model solve all of these problems. The relative firing rates for the graphs in figure 21 are determined at compile time, so the model of execution is neither data driven nor demand driven. Moreover, the execution model is complete, requiring no additional control, and there is no loss of concurrency. Similar results are obtained using the token flow for graphs that do not fit the SDF model [9]. For multidimensional streams, permissible firing patterns are again determined at compile time. For example, suppose the index space for the self-loop of actor C in figure 20 is doubly infinite (infinite in both dimensions). The permissible wavefront pattern of execution and all the parallelism it implies can be determined at compile time, and appropriate run-time control flow can be synthesized.

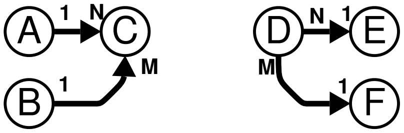

Figure 21. Two SDF graphs that illustrate problems with both demand and data-driven execution.

giving the number of samples produced and consumed by an actor. This is necessary to use multi-dimensional dataflow over non-rectangular index spaces.

5.0 Neither Demand nor Data-Driven Execution

Since streams are data structures like any other, streams of streams can be formed, although functional operations on such composite objects can be ambiguous. Consider for example a stream s of streams p_i of infinite length. Consider a function $f(s)$ applied to the stream. This constitutes the nested infinite iteration

forall i in (0 ... ∞) { forall j in (0 ... ∞) { $f(p_{i,j})$ }},

where $p_{i,j}$ represents the $j-th$ element of the stream p_i.We assume non-strict semantics (necessary for such operators on infinite streams to make sense) and functional semantics (so that we can use the “forall” construct, which implies no ordering constraint). For functional operators on finite sets of data elements, the order of execution does not matter. However, the order of execution of the above iteration, although unspecified, is clearly important. Consider the difference between

for i in (0 ... ∞) { forall j in (0 ... ∞) { $f(p_{i,j})$ }}.

and

forall i in (0 ... ∞) { for j in (0 ... ∞) { $f(p_{i,j})$ }},

where the “for” construct implies sequential execution.

For practical applications, we can restrict streams of streams so that they are infinite in one dimension only, in which case there is no ambiguity of specification. There remains, however, some ambiguity of execution. Consider the nested iteration

forall i in (0 ... M) { forall j in (0 ... ∞) { $f(p_{i,j})$ }}

If this is executed as

for i in (0 ... M) { for j in (0 ... ∞) { $f(p_{i,j})$ }},

then only the infinite stream for $i = 0$ will be processed.

These are identified in figure 19. 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.

4.6 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 20. Assume, as is common, that dimension 1 indexes the row in the index space, and dimension 2 the column, as shown in figure 9. 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.

4.7 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

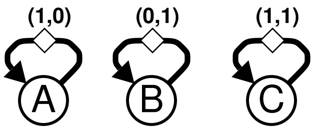

Figure 20. Three macro actors with state represented as a self-loop.

needed in a single-assignment specification to carry a variable forward in the index space [15]. 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 18. The key actors used for this are:

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

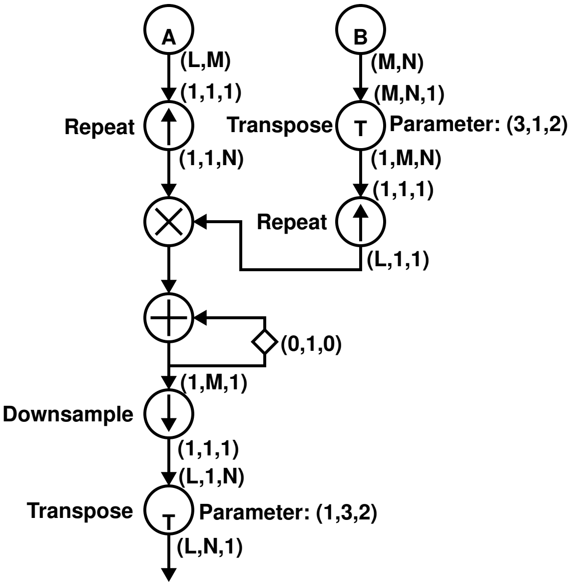

Figure 18. Matrix multiplication in MD-SDF.

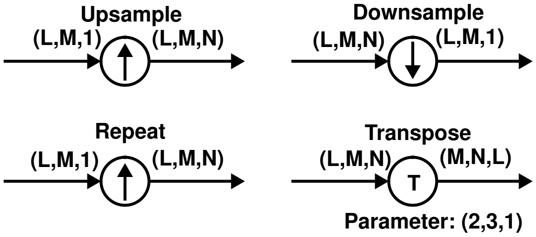

Figure 19. : Some key MD-SDF actors that affect the flow of control.

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 15 are equivalent.

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 16 are equivalent.

4.5 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 be accomplished as shown in figure 17. A 3D index space is used. 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

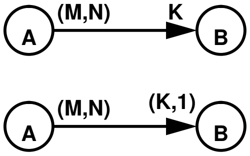

Figure 15. Rule for augmenting the dimensionality of a producer or consumer.

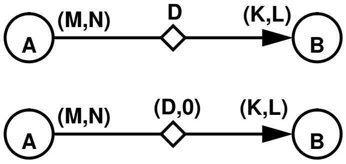

Figure 16. Rule for augmenting the dimensionality of a delay.

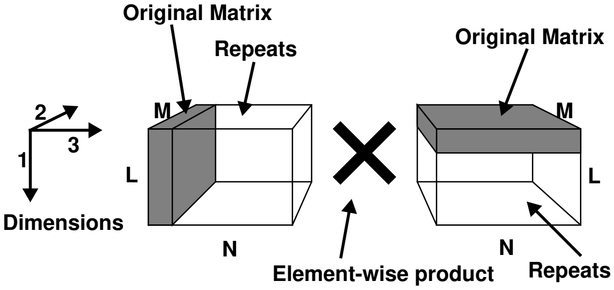

Figure 17. Matrix multiplication represented schematically.

A delay in MD-SDF in associated with a tuple as shown in figure 13. It can be interpreted 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 14. The only significant difference between this and figure 13 is the multidimensional delay. Its effect is illustrated schematically in figure 14, where the index space for the output of the delay is shown. The shaded area is the initial condition specified by the delay. Note that the index space for each arc in the system can be (and usually will be) different, unlike reduced dependence graphs [15].

4.4 Mixing Dimensionality

Note that in figure 14, 2D and 1D-SDF are mixed. We use the following rule to avoid any ambiguity:

- 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

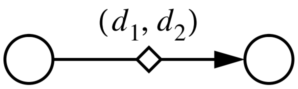

Figure 13. A delay in MD-SDF is multidimensional.

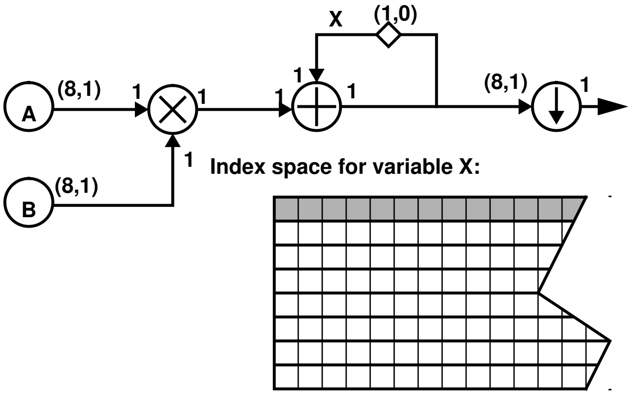

Figure 14. Repeated inner products in MD-SDF.

A practical application of this more flexible data exchange is shown in figure 11. Here, ten successive vectors of length 8 emanating from actor A are averaged. Averaging of each vector element is independent and hence may proceed in parallel. The ten successive firings of actor A may also be independent, so if all input samples are available, they too may proceed in parallel.

4.3 Inner Products

Consider the problem of repeatedly computing an inner product on a stream of vectors. In particular, suppose we wish to express the inner product at its finest level of granularity, exposing all parallelism, 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 12. 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 as above 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, the last, discarding the rest. This configuration will 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. The MD-SDF model provides an elegant solution.

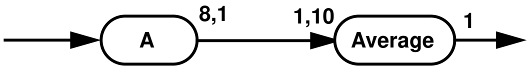

Figure 11. Averaging successive FFTs using MD-SDF.

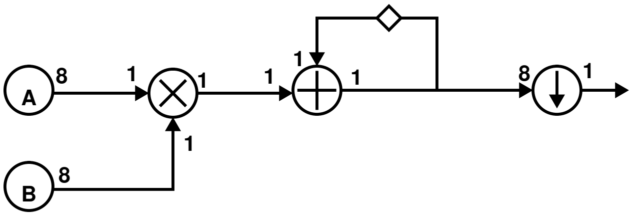

Figure 12. An attempt to use 1D-SDF to repeatedly compute inner products.

level of the hierarchy, the dataflow graph is shown in figure 9. The solution to the balance equations is given by

$$r_{A,1} = r_{A,2} = 1, r_{DCT,1} = 5, r_{DCT,2} = 6. \tag{8}$$

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 six horizontal firings of the DCT. A precedence graph constructed automatically from this would show 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.

4.2 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. 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 10. 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 10 shows this. Also shown is the index space of the tokens transferred along the arc, with the shaded regions indicating the tokens produced by the first firing of A and consumed by the first firing of B.

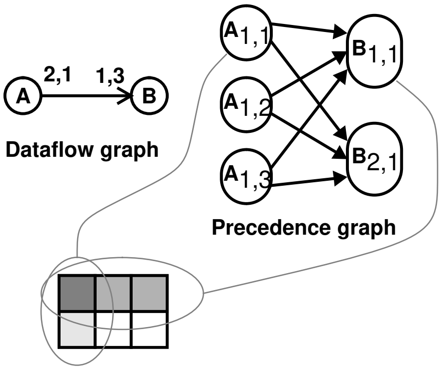

Figure 10. Data exchange in an MD-SDF graph.

4.0 Multidimensional Dataflow

The dataflow streams above suffer from being 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, as it is in the SDF model. The multidimensional SDF model is a straightforward extension of one-dimensional SDF. Figure 8 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 8 are

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

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

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. The techniques given in [16] for automatically constructing a precedence graph are easily adapted to the MD case.

4.1 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

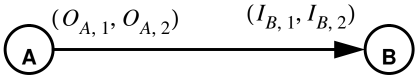

Figure 8. A simple MD-SDF graph.

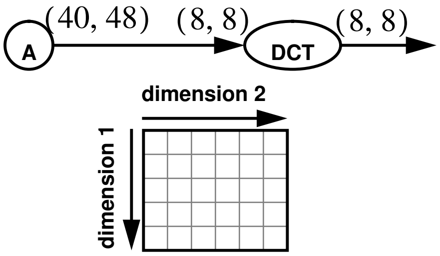

Figure 9. An image processing application in MD-SDF.

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 schedule is not a simple nested loop, although schedules with simple nested loop structure can be constructed systematically [5]. Notice that unlike the “synchronous” languages Lustre and Signal, we do not need a notion of clocks to establish a relationship between the stream into actor A and the stream out of actor B.

3.0 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 [9][18]. Consider the SWITCH and SELECT actors in figure 7. These route tokens conditional on a Boolean 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 [9].

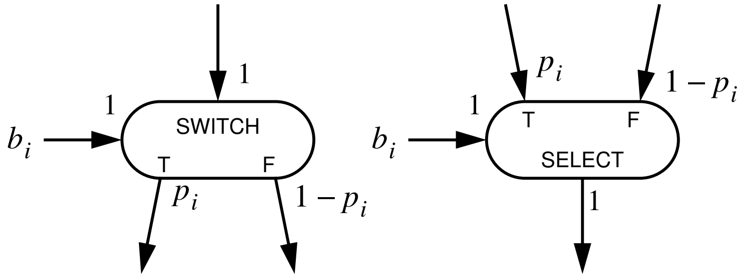

Figure 7. Dynamic dataflow actors annotated with the expected number of tokens produced or consumed per firing as a function of p_i, the probability that a token from the stream b_i is TRUE.

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

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

$$\vec{r}^T = \begin{bmatrix} 1 & 10 & 100 & 10 & 1 \end{bmatrix}, \tag{5}$$

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 [7]. An application to vector operations is shown in figure 5, where two FFTs are multiplied to effect frequency domain convolution. Both function and 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 an SDF graph, permitting exploitation of parallelism within each FFT as well. Data parallelism is therefore expressed in a compact graphical notation.

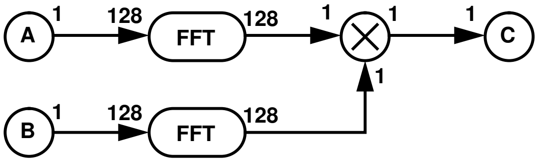

Figure 5. Application of SDF to vector operations.

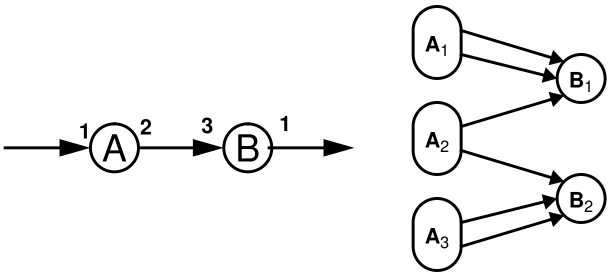

Figure 6. An SDF graph and its corresponding precedence graph.

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. 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. Hence, actor number 1 is enabled by I_1 tokens at its input, and when it fires, it will consume those tokens and produce O_1 tokens on its output. Clearly, the number of firings of actor 1 should be related to the number of firings of actor 2 so that all tokens produced on an arc should also be consumed. This constraint is expressed by 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 using techniques given in [16]. Given this precedence graph, good parallel schedules can be constructed [22][23], and from these, code can be synthesized for multiprocessor systems [6].

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 equations can be collected into matrix form

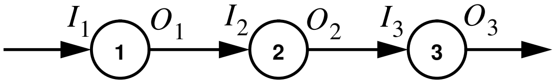

Figure 3. A simple synchronous dataflow graph.

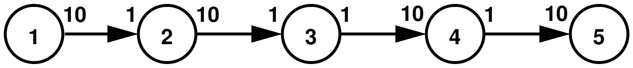

Figure 4. Nested iteration described using SDF.

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. In this paper, we will use a graphical syntax with only occasional pointers to equivalent textual syntax.

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 [10], Signal [4], 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 [4].

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, 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.

2.0 Synchronous Dataflow — The Background

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 [6] and Ptolemy [8] programs use this model. It has also been used in Aachen [21] and at Carnegie Mellon [20]. 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

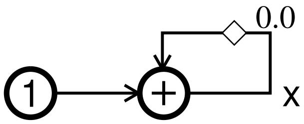

Figure 2. A graphical equivalent to the Silage program given in the text.

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 [6][8]). 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.

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. 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 and the "->" operator in Lustre. 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

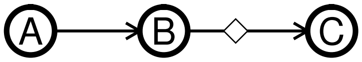

Figure 1. A graphical dataflow program.

Lucid, streams are multidimensional sequences. In Lustre and Signal, streams are sequences with “clocks” that align tokens in different streams relative to one another. In this paper, we develop an alternative representation for streams based on early conceptions of dataflow languages [11][12], more recent large-grain dataflow languages [25], and countless signal processing languages rooted in dataflow (see for example [6][8][13]). These latter models are extended to cleanly support streams of streams, as done in Id, Lucid, and Sisal, thus promoting streams to first-class data structures. Unlike Id, Lucid, and Sisal, however, streams are developed using a producer-consumer relationship between actors, extending the so-called “synchronous dataflow” model [16][17] and the “token flow” model [18][9] to multiple dimensions. We argue in this paper that this interpretation leads both elegant representations of some algorithms and clear semantics that are easily managed by a parallelizing compiler.

Streams are different from most data structures in that they are infinite in size. They should not be viewed as merely of indeterminate size, as is common with dynamically constructed data structures, but in reactive and signal processing systems rather are so large that they must be considered actually infinite. Streams can subsume finite data structures such as arrays as special cases, but any interesting language and execution model must assume that they are infinite. For this reason, we argue, streams cannot be strict. Any language with strict streams has simply defined a different syntax for operating on lists or arrays. Furthermore, Lisp-style lists are not the appropriate basis for streams. In graphical languages based on dataflow [6][11][25] and some textual stream-oriented languages [4][10][13], Lisp lists have had no influence. Streams are instead sequences of tokens managed through a producer-consumer relationship. Broadly speaking, these languages divide into two classes, those with dataflow semantics, in which tokens that are produced live until they are consumed [6][11][13][25], and those with clock-based semantics [4][10], in which tokens are produced or consumed on compatible clock ticks. The model in this paper has dataflow semantics.

In a graphical syntax, the program is a graph with nodes representing actors and arcs representing streams, as shown in figure 1. Each token produced by 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

MULTIDIMENSIONAL STREAMS ROOTED IN DATAFLOW

Edward A. Lee

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720

To appear in the Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, January 20-22, 1993, Orlando, Florida, USA, published by North-Holland.

ABSTRACT

A programming model rooted in dataflow principles and supporting multidimensional streams is developed and compared with streams in Lucid, Lustre, Signal, Silage, Sisal, and Id. The model is based on production and consumption of tokens and requires neither the “clock” synchronization of Lustre and Signal nor the Lisp-like list model of Sisal and Id. Multidimensional streams are similar to those in Lucid, but augmented with a producer/consumer model derived from our earlier work on one-dimensional dataflow models. Analytical properties of programs are easily derived, and compile-time predictability is maximized so that run-time overhead costs can be minimized. The scheme is illustrated by building scalable fine-grain programs for some simple examples and showing how data and function parallelism can be automatically exploited given these descriptions.

1.0 Motivation

Skillcorn [24] argues that streams and functions on them are a natural way to model reactive and distributed systems. Reactive systems, such as servers, operating systems, and signal processing systems, operate continuously and produce and consume unbounded message sequences. For operating on such sequences, languages such as Id [1], Lucid [2][24], Sisal [19], Lustre [10], and Signal [4] support streams. In Sisal and Id, streams are lists fashioned after lists in Lisp. In

---

The author gratefully acknowledges the support of The National Science Foundation, AT&T Bell Labs, Philips, and Hitachi.