1995 Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers

Design and Implementation of a Multidimensional Synchronous Dataflow Environment*

Michael J. Chen

ViaSat, Inc.
2290 Cosmos Court
Carlsbad, CA 92009, USA
Email: mchen@viasat.com

Edward A. Lee

Dept. of Electrical Engineering and Computer Science, University of California at Berkeley
Berkeley, CA 94720, USA
Email: eal@EECS.Berkeley.EDU

Abstract

Multidimensional Synchronous Dataflow is a model of computation where functional blocks produce and consume data on a multidimensional index space. The model can express multidimensional signal processing systems and algorithms elegantly, and is able to reveal the data parallelism of such systems in a way that raises possibilities for better multiprocessor scheduling. We have formalized definitions and specifications for a two-dimensional model and have created a single-processor simulation domain in Ptolemy[1]. Our paper discusses the design issues and methodologies we used to efficiently handle the large amounts of two-dimensional data that the systems operate upon. The primary structures used were matrices to act as buffers and submatrices to access subsets of those buffers.

1: Introduction

Multidimensional dataflow is the term used by Lee [2] to describe an extension to the more common graphical dataflow model implemented in Ptolemy [1] and other systems. The concept involves working with multidimensional streams of data instead of a single stream. Unlike other interpretations of multidimensional dataflow [7,8] that focus more on data dependency and linear indexing issues in textual and functional languages, our focus is primarily on the graphical representation of algorithms (such as those used in multidimensional signal processing and image processing), and exposing data parallelism for multiprocessor scheduling. This paper deals specifically with Multidimensional *Synchronous* Dataflow (MDSDF), where the multidimensional specifications of the number of input tokens consumed and output tokens produced for every actor in the system are fixed at compile time. This differs from a more general dataflow model where the number of inputs and outputs can vary dynamically during execution of the system. In addition, we will be discussing only the two-dimensional implementation of MDSDF.

1.1: SDF and Ptolemy terminology

Synchronous Dataflow (SDF) [4,5] is a mature model for one-dimensional dataflow in Ptolemy. Since MDSDF is in many ways a straightforward extension of the capabilities of SDF, we begin with a brief discussion of how we represent SDF systems in Ptolemy. We emphasize that this presentation is intended to be a summary and not an in-depth discussion. Much work has been done to formalize the concepts of SDF, so we strongly suggest that the reader refer to the papers on SDF and Ptolemy, especially [4], for a more complete discussion.

In SDF and other graphical models of one-dimensional dataflow, the data transferred between functional blocks (also called *actors* or *stars*) is of simple form, i.e. a single value such as a floating-point number, an integer, a fixed-point number, or a complex number. In Ptolemy, these values are held in a container structure called a *particle,* and these particles are transmitted between Ptolemy actors. Actors can also transfer more complex data structures, such as vectors and matrices, using a specialized particle called a MessageParticle.

A simple SDF system in Ptolemy is pictured in Figure 1.

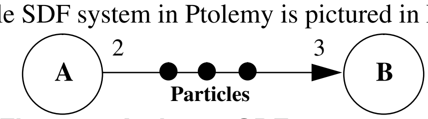


Figure 1. A simple SDF system.

Actors are connected together by arcs that represent FIFO queues. The arcs are attached to an actor at a location called a *porthole*. An actor can have more than one input or output porthole. The numbers along the arc connecting two

* This research is part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), Semiconductor Research Corporation (project 94-DC-008), National Science Foundation (MIP-9201605), Office of Naval Technology (via Naval Research Laboratories), Bell Northern Research, Cadence, Dolby, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, Rockwell, Sony, and Synopsis.

portholes specify the number of particles generated or consumed each time their associated star executes (this execution is called a star *firing* in Ptolemy). In the above example, actor A generates two particles at its output porthole and actor B consumes three particles at its input porthole per firing.

We define SDF to be synchronous because the number of inputs consumed and outputs produced by every actor when it fires is fixed at compile time. This feature gives the scheduler of the SDF domain (note that SDF is just one model of computation supported by Ptolemy, other models of execution are implemented as Ptolemy *domains*) the ability to generate a compile-time schedule for simulation and code generation purposes. This schedule is called a *periodic admissible sequential schedule* (PASS). A PASS is a sequence of actor firings that executes each actor at least once, does not deadlock, and produces no net change in the number of particles on each arc. Thus, a PASS can be repeated any number of times with a finite amount of buffer space for each arc. Moreover, the maximum size of the buffer for each arc is a constant that is determined by the exact sequence of actor firings in the schedule. We call each of these repetitions of the PASS an *iteration*. For the example system above, there are two possible PASSs: AAABB and AABAB.

SDF systems also support the concept of delays. A delay is depicted by a diamond on an arc, as shown in Figure 2. The delay is specified by an integer N that is interpreted as a sample offset between the input and the output streams. The offset is implemented by inserting N initial particles on the arc between the two actors, so that the first particle consumed by actor B when it fires is the initial particle. Most often the values of the initial particles are zero, but Ptolemy allows the user to give these initial particles non-zero values.

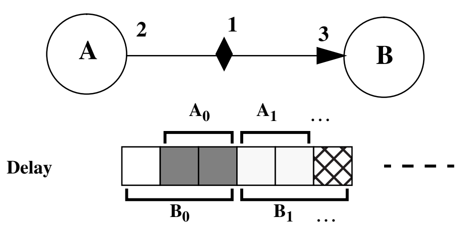


Figure 2. A SDF system with a delay.

1.2: MDSDF Graphical Notation

Although the graphical notation of MDSDF is closely related to that of SDF and in many ways just a simple extension, the added freedom of the multidimensional system allows several semantic interpretations. Such flexibility can lead to confusion by both the user of the system and the person implementing it. Examples of such possible areas of confusion are how to interpret two-dimensional delays and how to define an actor that needs access to data in the "past" or in the "future" along either dimension. This section summarizes our definitions of MDSDF semantics and gives a few examples, but we refer the reader to [2, 3] for more thorough presentations of MDSDF semantics, uses, and implementation details.

In MDSDF, the graphical notation of SDF is extended by adding an extra dimension to the input/output specifications of each porthole of a star. A MDSDF star in our current two-dimensional implementation has input and output portholes that have two numbers to specify the dimensions of the data they consume or produce, respectively. These specifications are given as a (*row*, *column*) pair. For example, Figure 3 shows the system of Figure 1 extended into a MDSDF system.

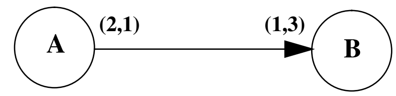


Figure 3. MDSDF extension of the system in Figure 1.

Unlike the SDF case, which can support two-dimensional data objects such as vectors and matrices by passing them inside the MatrixParticle (a specialized MessageParticle), the data generated by a MDSDF star is not a self-contained monolithic structure but is considered part of a underlying two-dimensional indexed data space. The vectors and matrices of SDF are of fixed size and the actors that operate on a stream of such data structures can only manipulate each of them individually. On the other hand, the input/output specifications of a MDSDF star simply give us directions on how to arrange the data consumed or produced by the star. The idea is that the specifications define a window into the data space, the exact location of which is determined by the index of the particular firing of the actor and the dimensions of its data. An actor producing two-dimensional data onto an arc is not required to have data dimension that match those of another actor consuming data from the same arc. It is up to the MDSDF domain to schedule, collect, and distribute the data so that the system operates correctly to consume all data that is produced. This is best illustrated by an example.

Figure 4 shows the underlying data space for the system of Figure 3. The two data values produced by each firing of actor A are placed in a two row by one column block in the data space. The first such block, corresponding to the first firing of actor A, which we designate as $A_{[0,0]}$, is placed into the data space such that it occupies the data locations $\mathbf{d}_{[0,0]}$ and $\mathbf{d}_{[1,0]}$. Similarly, each firing of actor B consumes three data values, which we regard as a one row by three column block of the data space. Firing $B_{[0,0]}$ will thus consume data from locations $\mathbf{d}_{[0,0]}$, $\mathbf{d}_{[0,1]}$, and $\mathbf{d}_{[0,2]}$. It is clear from the diagram that a PASS for this system will require actor A to fire three times along the column dimension (i.e. three column firings), allowing actor B to fire twice along the row dimension (i.e. two row firings). A shorthand for such a schedule is $A_{[0,0]}A_{[0,1]}A_{[0,2]}B_{[0,0]}B_{[1,0]}$. This schedule is infinitely repeatable and requires a buffer of six data locations between actors A and B. Thus, MDSDF retains the feature of SDF where the fixed porthole specifications allow us to compute at compile time a schedule with fixed buffer size requirements (or *static buffering* [6]).

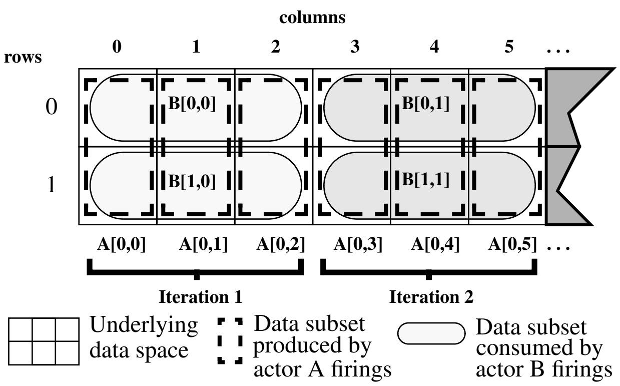


Figure 4. Data space for the MDSDF system of Figure 3.

The data space diagram reflects our design decision to iterate time along the column dimension only. Although we consider the underlying data space to be possibly infinite in the two dimensions, in practice the row dimension will always be finite. This interpretation of time differs from that of other multidimensional dataflow models [7, 8] that treat time as another dimension in the data space.

Note that the firing index of an actor is directly associated with a fixed location in the data space, but they are not exactly equivalent. We need to know the size of the blocks produced or consumed by the porthole to determine the exact mapping between the firing instance of the actor and its corresponding data space.

Additionally, an important feature about the PASS we specified for the above example is that the two sets of firings for actor A and actor B could have been scheduled for parallel execution. The data space diagram clearly shows that the three firings of actor A are data independent and can be executed in parallel. Similarly, once all three firings of A are complete and the data they produce available, the two firings of actor B are also data independent and can be scheduled for parallel execution. Our shorthand for the PASS is thus an imperfect one since it implies an ordering for the various firings that is unnecessary, but since we have implemented only a single-processor simulation domain for MDSDF thus far, it will be sufficient for our discussion. We do want to emphasize that it is this feature of revealing the data parallelism of systems that we hope to exploit in the future to develop better multiprocessor schedulers.

The last two basic features of MDSDF that we must explain deal with dependency of an actor on data that is "before" or "after" in the two-dimensional data space. In SDF, the model of interpreting the arcs as FIFO queues implies an ordering of where particles are in time. Therefore, we could discuss how stars could access data in the "past." In MDSDF, since one of our main goals is to take advantage of parallel execution, we do not impose a time ordering along the two dimensions of the data buffer within one iteration period (note that there is an ordering between the data of successive iterations). Therefore, for lack of better terms and since we still wish to support the ability to reference data offset from the current reference point in the data space, we use "before" or "past" and "after" or "future" in each dimension to refer to data locations with lower or higher index, respectively, in each dimension. So data location $\mathbf{d}_{[0,0]}$ is before $\mathbf{d}_{[0,1]}$ in the column dimension, but they are at the same "time" in the row dimension.

A related concept is the idea of a delay in two dimensions, which can have a number of interpretations. We have chosen to interpret a two-dimensional delay as boundary conditions on the data space, with indexing offsets between the portholes attached to the arc with a delay. For example, Figure 5 shows a MDSDF system with a two-dimensional delay. The delay, just like the portholes of a MDSDF actor, has a (*row*, *column*) specification. The specifications for a two-dimensional delay tell us how many initial rows and columns the input data is offset from the origin of the data space $\mathbf{d}_{[0,0]}$. We see that firing $A_{[0,0]}$ is now mapped to buffer locations $\mathbf{d}_{[1,1]}$ and $\mathbf{d}_{[2,1]}$.

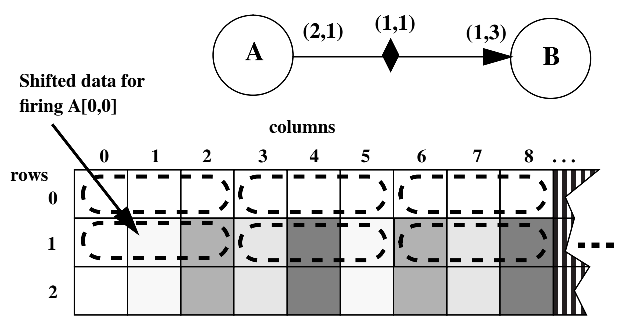
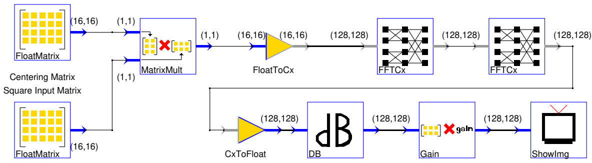


Figure 5. A MDSDF system with a two-dimensional delay.

2: 2D-FFT Example

In this section, we present an example of a MDSDF system which computes the 2D FFT of a white square on a black background. The screen dump for this system and the output generated by running it in Ptolemy is shown in Figure 6.




```
MDSDF SCHEDULE:
fft_of_square2.FloatMatrix1, firing range: (0,0)
fft_of_square2.FloatMatrix2, firing range: (0,0)
fft_of_square2.Mult1, firing range: (0,0) - (15,15)
fft_of_square2.FloatToCx1, firing range: (0,0)
fft_of_square2.FFTCx2, firing range: (0,0) - (15,0)
fft_of_square2.FFTCx1, firing range: (0,0) - (0,127)
fft_of_square2.CxToFloat1, firing range: (0,0)
fft_of_square2.DB1, firing range: (0,0)
fft_of_square2.Gain1, firing range: (0,0)
fft_of_square2.ShowImg1, firing range: (0,0)
```

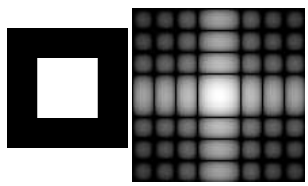


Figure 6. Screen dump of 2D-FFT system, the associated schedule, and outputs.

The top figure shows the system as specified in the MDSDF domain in Ptolemy. We have manually labeled the dimensions for all the arcs. The input is an eight by eight pixel square of magnitude four on a sixteen by sixteen pixel background square (the rest of the pixels have magnitude zero). This source matrix is multiplied by another matrix, entry by entry, in order to shift the center of the output of our 2D-FFT to the center of the display. The output of is then scaled before being processed by the 2D-FFT stars. A 2D-FFT is straightforwardly implemented by computing the 1D-FFTs of the rows and then the columns in series. Conversion stars are used to change the float matrix to a complex matrix and back again (since the FFTs work on complex data). Lastly, the output matrix is fed through a stars which scale the values into decibels and puts them within the range for the display star. The input square and the output 2D-FFT of the square are the last two images at the bottom right of the figure.

The middle window in the figure shows the schedule that is generated for the system. Notice especially how the schedule reveals the parallelism inherent in the system. The input to the first row FFT star has dimensions (1,16). Therefore, 16 1D-FFTs are applied to the rows of the input. Those 1D-FFTs have been set to generate 128 output points each by zero padding the input, thus the accumulated output of the row FFTs will be a (16,128) block of data. Next, 128 column FFTs are computed, with each column having 16 values each, again zero padded to produce 128 points. Thus, the final output of the 2D-FFT galaxy is a block with dimensions (128,128). Each of the sets of 16 row FFTs and 128 column FFTs are independent and could have been schedule in parallel in a multiprocessor system.

3: Ptolemy Implementation Details

This section discusses the details of the implementation of MDSDF in Ptolemy. Specifically, we discuss the problem of handling the large amounts of two-dimensional data that needs to be transferred during execution in an efficient way.

3.1: Two-dimensional data structures - matrices and submatrices

Since MDSDF uses a model in which actors produce data that are part of a two-dimensional data space, the data structure used to represent both the buffers and the subsets of the buffer that the stars can work with is very important. Currently, the data structure used for the buffer is the `PMatrix` (the 'P' is silent) class from Ptolemy's kernel. A subclass of the `PMatrix` class, called the `SubMatrix` class, was developed to act as the primary structure used by stars to access data from the buffer. There are four `SubMatrix` classes: `ComplexSubMatrix`, `FixSubMatrix`, `FloatSubMatrix`, and `IntSubMatrix`, to match the four corresponding types of the `PMatrix` classes supported in Ptolemy. Actors never deal with the buffer directly, but read and modify subsets of the buffer strictly through the submatrices.

The primary function of the `SubMatrix` class is to provide an interface to a window of the memory allocated by the `PMatrix` class. Every submatrix has a pointer to a "parent" or "mother" matrix, and many submatrices can have the same parent. Only the parent matrix has allocated memory to act as the buffer. A submatrix simply accesses a subset of this buffer, using the information it knows about its own dimensions and that of its parents. The relationship between submatrices and parent matrices is depicted in Figure 7.

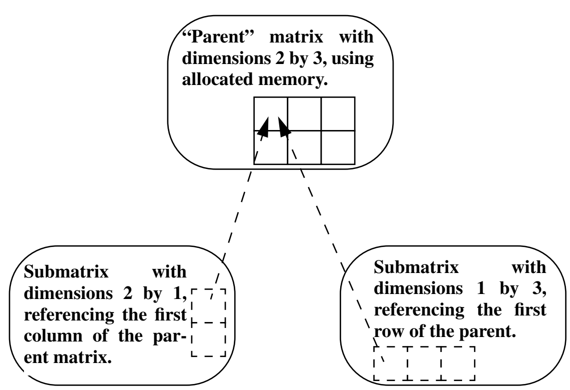


Figure 7. Parent matrix and two submatrices accessing its buffer.

The interface of the `SubMatrix` class for accessing individual entries is the same as that of the `PMatrix` class. Using the function name overloading ability of C++, we have overloaded the `[]` operator and the `entry()` method so that they return the entry from the correct location in the memory of the parent matrix. These interface functions provide a very natural syntax for dealing with two-dimensional data. The various arithmetic operators, such as addition and matrix multiplication, and matrix manipulation operators, such as transpose and inverse, are inherited from the `PMatrix` class and are still functional on the submatrices. An example of `SubMatrix` class usage is shown in Figure 8.

```
// Declare a 4 by 4 parent matrix
IntMatrix *A = new IntMatrix(4,4);

// Make B a 4 by 2 submatrix of A, such that
// its origin is entry (0,2) of matrix A
// (i.e. submatrix B references the last two
//  columns of matrix A)
IntSubMatrix *B = new IntSubMatrix(A,4,2,0,2);

// Create a matrix to act as the output
IntMatrix *C = new IntMatirx(4,2);

// Assign all entries of C the value 0
*C = 0;

// Give entry (3,3) of matrix A the value 1,
// this also results in entry (3,1) of
// submatrix B having the same value
*A[3][3] = 1;

// Give entry (0,0) of submatrix B the value
// 2, this also results in entry (0,2) of
// matrix A having the same value
*B[0][0] = 2;

// Do matrix multiplication of two matrices
*C = (*A) * (*B);
```

Figure 8. Use of the `SubMatrix` class interface.

3.2: Buffering and Flow of Data

In the current implementation of the SDF simulation domain, a great deal of overhead is involved in moving data particles from one actor to another because the design has been implemented to model general dataflow constructs. Each actor is connected to another by a series of structures. First, there is the *porthole*, which acts as a buffer to hold the data particles, either before they are sent by the actor producing them or when they are received by actors consuming the data. The arc connecting the portholes is implemented using the *geodesic* structure, which also has a buffer that acts as a FIFO queue. These structures are depicted in Figure 9

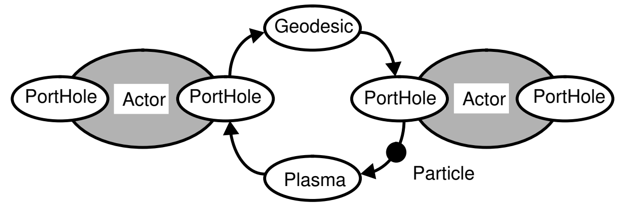


Figure 9. Close-up of connections for data transfer between actors in the SDF simulation domain.

The particles go into the geodesic buffer when the source actor has finished firing to produce the output data. The particles move from the geodesic buffer to the buffer of the destination porthole when the destination actor is ready to fire and consume its input data. After the destination actor has fired, the “empty” particles are returned to the *plasma*, which acts as a repository of empty particles that can be reused by the source porthole.

We felt that this system of having three buffers (one in each porthole and one in the geodesic) per arc would be too inefficient for MDSDF. Many MDSDF systems have large rate changes, which results in a large number of particles flowing through the system if we use the old style of implementation. An example of such a system was given previously with the 2D-FFT system. Computing the 2D-FFT of an image would generate thousands of particles of data if treated at the pixel level. This inefficiency is not inherent to SDF. On the contrary, SDF systems in general have very desirable qualities, such as the ability to make static schedules and perform static buffer allocation for them. These qualities have been exploited in SDF code generation domains, but not for the SDF simulation domain. MDSDF has similar qualities, so we have designed the MDSDF simulation domain to take advantage of these attributes to reduce the amount of buffering overhead in the system.

We mentioned in the previous section that stars in MDSDF access the data space using submatrix structures instead of through particles like SDF stars. These submatrices are not buffered at all, but are created and deleted as needed when the star requests one for input or output purposes (it might be even more efficient to allocate a submatrix plasma to store “empty” submatrices so that we can reuse allocated memory for the structures, but this would involve additional overhead to maintain such plasmas, which we wanted to avoid in our initial implementation). For example, a star that generates data would first request from the output porthole a submatrix to access the matrix acting as the buffer for the output connection. That star could then write to the entries of that submatrix using the standard matrix operations. Similarly, a star that receives input from another star could access the data by requesting a submatrix to access the matrix acting as the buffer for the input connection. This is in contrast to the standard SDF style of working on the particles and the data they contain directly. No buffers of particles or submatrices are used at all in this implementation of the MDSDF simulation domain. The geodesic contains one matrix that acts as the buffer for the arc, and all accesses to that buffer are conducted through submatrices that are created and deleted as needed.

Since submatrices are never buffered, the responsibility to delete them falls on the stars that requested them. This makes the code in MDSDF stars slightly more complex than those of standard SDF stars that rely on the system to recover particles that are no longer being used. Although it is possible to buffer the submatrices used by a star in the portholes for that star, which would give us the advantage of maintaining pointers to all the submatrices used so that the system could recover or reuse them instead of having the stars delete them after each use, this would involve an additional overhead of maintaining a two dimensional buffer. In our initial attempt at implementing a MDSDF simulation domain, we did not think the gains justified the additional complexity.

In summary, this system of having only one matrix allocate memory for the buffer of an arc connection makes our system very efficient. A possible alternative implementation would have the source actor produce matrices with their own allocated storage. The problem with that system is that if a destination actor required inputs that differed in dimensions from those provided by the source actor, the mismatched dimensions would have to be reconciled by copying and/or merging inputs and then creating new output matrices. Our system avoids such excessive copying and shuffling of data. Source and destination actors work directly with subsets of the larger buffer using the submatrix access structures. The buffer itself has a fixed size that can be determined and allocated at compile time from the information provided by the MDSDF graph.

4: Conclusion

We have discussed various issues that arose during our implementation of a MDSDF domain in Ptolemy. There are numerous challenges in efficiently managing the large amounts of data that a typical MDSDF system would generate. We have presented the formal specifications of a workable MDSDF model, and presented some examples of its features. We have also presented a discussion of the complexities involved in implementing a simulation environment for MDSDF and the design decisions we made to simplify the implementation. Currently, a MDSDF single-processor simulation domain has been implemented in Ptolemy. It has been tested on small simple systems such as 2D-FFT computation, and 2D FIR and IIR filters. Areas for future work include implementing a multiprocessor scheduling target and examining possible extensions of the system to greater than two dimensions.

5: References

[1] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, “Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems”, *International Journal of Computer Simulation*, special issue on “Simulation Software Development,” January, 1994.

[2] E.A. Lee, “Multidimensional Streams Rooted in Dataflow,” *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Jan 20-22, 1993, IFIP Transactions A (Computer Science and Technology), 1993, vol.A-23:295-306.*

[3] M.J. Chen, “Developing a MultidimensionalSynchronous Dataflow Domain in Ptolemy,” **MS Report**, ERL Technical Report UCB/ERL No. 94/16, University of California at Berkeley, Berkeley, CA 94720, May 6, 1994.

[4] E.A. Lee and D.G. Messerschmitt, “Synchronous Data Flow,” *Proceedings of the IEEE*, Vol. 75, No. 9, pp. 1235-1245, September, 1987.

[5] E.A. Lee and D.G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing” *IEEE Transactions on Computers,* Vol. C-36, No. 1, pp. 24-35, January, 1987.

[6] S. Bhattacharyya and E.A. Lee, “Memory Management for Synchronous Dataflow Programs,” Memorandum No. UCB/ERL M02/128, Electronics Research Laboratory, U.C. Berkeley, November 18, 1992.

[7] I.M. Verbauwhede, C.J. Scheers, and J.M. Rabaey, “Specifications and Support for Multidimensional DSP in the Silage Language,” ICAASP ‘94.

[8] F.H.M. Franssen, F. Balasa, M.F.X.B. van Swaaij, F.V.M. Catthoor, and H.J. De Man, “Modeling Multidimensional Data and Control Flow,” IEEE Transactions on VLSI Systems, Vol. 1., No. 3, pp. 319-27, September 1993.