Presented at the Asilomar Conference on Signals, Systems and Computers — October 1995

A Comparison of Synchronous and Cyclo-Static Dataflow

Thomas M. Parks, José Luis Pino and Edward A. Lee
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{parks,pino,eal}@EECS.Berkeley.EDU

Abstract

We compare synchronous dataflow (SDF) and cyclo-static dataflow (CSDF), which are each special cases of a model of computation we call dataflow process networks. In SDF, actors have static firing rules: they consume and produce a fixed number of data tokens in each firing. This model is well suited to multirate signal processing applications and lends itself to efficient, static scheduling, avoiding the run-time scheduling overhead incurred by general implementations of process networks. In CSDF, which is a generalization of SDF, actors have cyclicly changing firing rules. In some situations, the added generality of CSDF can unnecessarily complicate scheduling. We show how higher-order functions can be used to transform a CSDF graph into a SDF graph, simplifying the scheduling problem. In other situations, CSDF has a genuine advantage over SDF: simpler precedence constraints. We show how this makes it possible to eliminate unnecessary computations and expose additional parallelism. We use digital sample rate conversion as an example to illustrate these advantages of CSDF.

1 Dataflow process networks

In the process network model of computation [4, 5], concurrent processes communicate through unidirectional FIFO channels. Communication channels are represented mathematically by streams (sequences of data elements or tokens), possibly infinite in length, and processes are functional mappings from a set of input streams to a set of output streams. This is a convenient model for describing audio and video signal processing systems which must operate on infinite streams of data samples.

Dataflow actors have firing rules that determine when enough data tokens are available to enable the actor. When the firing rules are satisfied the actor fires; it consumes a finite number of input tokens and produces a finite number of output tokens. For example, when applied to an infinite input stream, a firing function f may consume just one token and produce one output token:

$$\mathrm{f}([x_1, x_2, x_3 \ldots]) = \mathrm{f}(x_1)$$

To produce an infinite output stream, the actor must be fired repeatedly. A processes formed from repeated firings of a dataflow actor is called a dataflow process [7]. The higher-order function map converts an actor firing function into a process:

$$\mathrm{map}(\mathrm{f})[x_1, x_2, x_3 \ldots] = [\mathrm{f}(x_1), \mathrm{f}(x_2), \mathrm{f}(x_3) \ldots]$$

A higher-order function takes a function as an argument and returns another function. When the function returned by $\mathrm{map}(\mathrm{f})$ is applied to the input stream $[x_1, x_2, x_3 \ldots]$, the result is a stream in which the firing function f is applied pointwise to each element of the input stream. The map function can also be described recursively using the stream-building function cons, which inserts an element at the head of a stream:

$$\mathrm{map}(\mathrm{f})[x_1, x_2, x_3 \ldots] = \mathrm{cons}(\mathrm{f}(x_1), \mathrm{map}(\mathrm{f})[x_2, x_3 \ldots])$$

The use of map can be generalized so that f can consume and produce multiple tokens on multiple streams [7].

Breaking a process down into smaller units of execution, such as dataflow actor firings, makes efficient implementations of process networks possible. Restricting the type of dataflow actors to those that have predictable token consumption and production patterns makes it possible to perform static, off-line scheduling and to bound the memory required to implement the communication channels.

In synchronous dataflow (SDF) [6] the number of tokens consumed and produced by an actor is constant for each firing. This property makes it possible to statically construct a finite schedule that can be periodically repeated to implement a process network that operates on infinite streams of data tokens. Cyclo-static dataflow (CSDF) [3, 1] generalizes SDF by allowing the number of tokens consumed and produced by an actor to vary from one firing to the next in

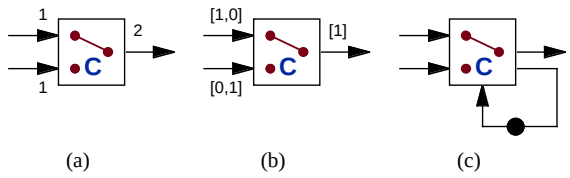


Figure 1: The commutator in synchronous, cyclo-static and state-dependent forms.

a cyclic pattern. Because these patterns are periodic and predictable, it is still possible to statically construct periodic schedules using techniques based on those developed for SDF [1]. CSDF has been extended to more general dynamic dataflow actors by allowing a data-dependent number of firings in a cycle [2]. In this more general case, it is not always possible to construct a finite schedule that can be repeated, nor is it always possible to put bounds on the memory required to implement the communication channels. In this paper we restrict our discussion to a comparison of SDF and CSDF, both of which can be used to implement process networks with periodic schedules and bounded memory.

1.1 Firing rules

Firing rules define the consumption of data from input streams when a dataflow process is constructed with map. For example, the SDF form of the commutator actor, shown in figure 1(a), has one firing rule: $f([x],[y]) = [x,y]$. It consumes a single token from each of two input streams, and produces a two-element sequence on the output. Both input streams must have at least one token available before f can fire.

The CSDF form of the commutator, shown in figure 1(b), has two firing rules: $f_1([x],[\,]) = [x]$ and $f_2([\,],[y]) = [y]$. In the first firing it consumes a single token x from one input stream and copies it to the output. In the second firing it copies a token y from the other input to the output. This firing sequence then repeats cyclicly. In general, CSDF actors have one rule for each firing of a cyclicly repeated sequence.

An internal state variable could serve as an index to enforce the proper firing sequence of a CSDF actor. If instead we follow a purely functional dataflow model in which actors are not allowed to have internal state, we must modify the firing rules so that the sequence index is shown explicitly as a function argument. The modified actor, shown in figure 1(c), has the firing rules $f([1],[x],[\,]) = ([2],[x])$ and $f([2],[\,],[y]) = ([1],[y])$. Each firing rule is enabled only when the proper value is available on the index stream, and produces the appropriate index value to enable the next firing in the sequence.

The self-loop used to keep track of the sequence index is a form of state feedback. SDF actors are a trivial example of such state-dependent firing rules — there is only one state,

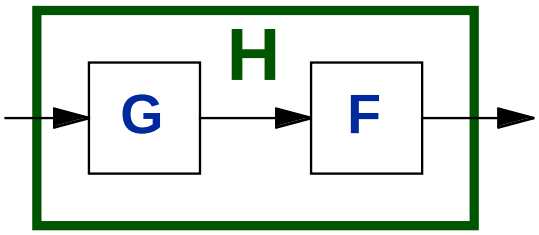


Figure 2: The composition $h = f \circ g$.

so the history of the index stream is a sequence of tokens all with the same value. In CSDF the history of the index stream is a cyclicly repeating pattern. Because this pattern can be predicted completely, it is possible to compute a static schedule and optimize away the state feedback.

1.2 Composition

Functional processes in a process network can be composed just like conventional functions. Two processes that are connected by a communication channel can be composed to form a functionally equivalent process, as shown in figure 2.

$$h(x) = f(g(x)) \iff h = f \circ g$$

Dataflow actors can be composed in a similar manner, but it is necessary to define a firing of the new composite actor. Assuming that the actors f and g shown in figure 2 each consume and produce a single token, then a natural definition for one firing of the composite actor h would be a firing of g followed by a firing of f. The graph in this figure is "well-ordered" because there is only one topological sort — one natural execution order. The graph in figure 3, however, is not well-ordered because once actor A has fired, both actors B and C are enabled and could fire in any order or even in

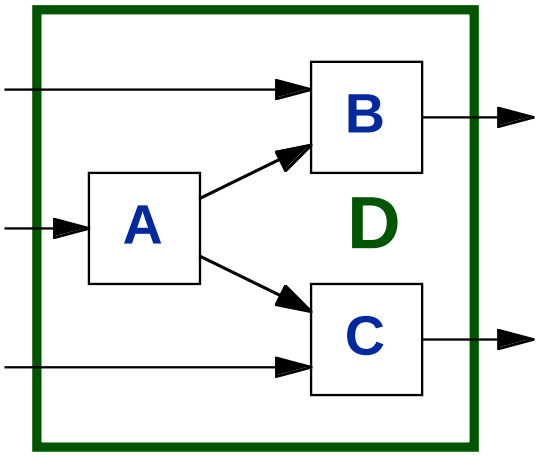


Figure 3: A graph that is not well-ordered.

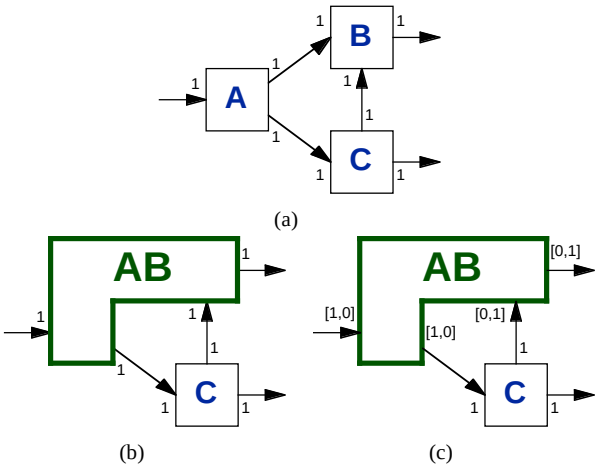


Figure 4: Deadlock introduced by imposing the SDF model on a composite dataflow actor.

parallel.

If we require that tokens be available on all inputs before execution begins, then a composite actor follows the SDF model. This gives the greatest possible flexibility in implementing a composite actor's internal schedule because data is available for any sequential or parallel schedule. However, imposing the SDF model on a composite actor leaves the least flexibility for the rest of the system, which must interact with that actor. All input tokens must be available simultaneously even if the tokens are actually consumed sequentially. This can introduce deadlock as illustrated in figure 4. A new directed cycle is introduced in figure 4(b) by combining actors A and B, and there is an insufficient number of tokens initially on the arcs of this cycle for any of the actors to be enabled. Composition can also introduce deadlock in other similar situations [8].

If instead we allow composite actors to follow the CSDF model, we can strike a balance between flexibility for the internal and external schedules. If the graph is well-ordered and there is only one natural execution order for the internal system, then the cyclo-static model describes the behavior of the composite actor completely — tokens are consumed and produced in the same order as in the original graph. Thus no parallelism is lost and deadlock is not introduced, as in figure 4(c).

2 Synchronous dataflow scheduling

A SDF graph can be described by a topology matrix Γ, where the element Γ_{ij} is defined as the number of tokens produced on the ith arc by the jth actor [6]. A negative value indicates that the actor consumes tokens on that arc. There is one row in this matrix for each arc in the graph, with one positive element for the actor that produces tokens and one

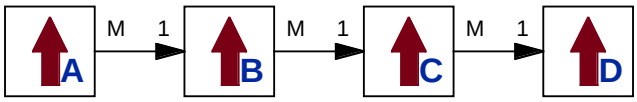


Figure 5: A multirate SDF graph with an exponential number of actor firings in a complete cycle.

negative element for the actor that consumes tokens. All the other elements in the row are zero.

Figure 5 shows an example of a multirate SDF graph. The topology matrix for this graph is:

$$\Gamma = \begin{bmatrix} M & -1 & 0 & 0 \\ 0 & M & -1 & 0 \\ 0 & 0 & M & -1 \end{bmatrix}$$

For the system to be balanced, a non-trivial positive repetition vector $\vec{r}$ must be found that satisfies the balance equations:

$$\Gamma \vec{r} = \vec{0}$$

where each element r_j of the repetition vector specifies the number of firings of the jth SDF actor, and $\vec{0}$ is the zero vector. In this example, the minimal integer solution for the balance equations is:

$$\vec{r} = \begin{bmatrix} 1 & M & M^2 & M^3 \end{bmatrix}^T$$

When each actor is fired the number of times specified by $\vec{r}$, the total number of tokens produced on each arc is equal to the total number of tokens consumed. We define this to be a *complete cycle*. In a complete cycle, a balanced system returns to its initial state with the same number of tokens on each arc. Thus the total memory required for the buffers associated with the arcs is bounded. If the balance equations have a non-trivial solution and a complete cycle can be executed (i.e. there is no deadlock), then this firing sequence can be repeated infinitely in bounded memory.

3 Cyclo-static dataflow scheduling

Unlike the scalar token consumption and production parameters Γ_{ij} for SDF, these parameters are vectors $\vec{\gamma}_{ij}$ for

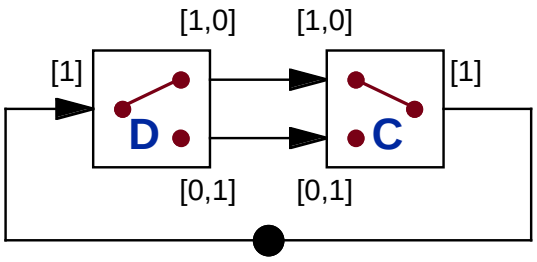


Figure 6: A CSDF system that becomes deadlocked when transformed to SDF.

CSDF [3]. Figure 6 shows an example of a simple CSDF graph using a commutator and a distributor. The distributor is the counterpart to the commutator: it *distributes* tokens from its input stream to several output streams. The first input token goes to the first output, the second input token goes to the second output, and so on. In this example, the token production parameters are: $\vec{\gamma}_{11} = \vec{\gamma}_{12} = [1], \vec{\gamma}_{21} = \vec{\gamma}_{22} = [1,0]$ and $\vec{\gamma}_{31} = \vec{\gamma}_{32} = [0,1]$.

Let $p_{ij} = \dim(\vec{\gamma}_{ij})$ be the length or period of the token production pattern for the ith arc connected to the jth actor. If there is no connection, then $p_{ij} = 1$. The jth actor fires in a cycle with period $P_j = \text{lcm}(p_{ij})$, the least common multiple of the consumption and production periods for all the arcs connected to that actor. In our example, $p_{11} = p_{12} = 1$ and $p_{21} = p_{22} = p_{31} = p_{32} = 2$. The cycle periods for the commutator and distributor in figure 6 are $P_1 = P_2 = 2$.

If we let σ_{ij} be the sum of the elements in $\vec{\gamma}_{ij}$, then the total number of tokens produced on an arc in a cycle of firings is given by:

$$\Gamma_{ij} = P_j \frac{\sigma_{ij}}{p_{ij}}$$

We can now solve the balance equations as described previously for SDF. For our example in figure 6 the topology matrix and repetition vector are:

$$\Gamma = \begin{bmatrix} -2 & 2 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$\vec{r} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$$

In CSDF, however, the repetition vector $\vec{r}$ represents not the number of actor firings, but the number of cycles. The number of actor firings is $r_j P_j$.

4 Transforming CSDF to SDF

The number of actor firings that must be scheduled can be exponential relative to the number of nodes in an SDF graph [8]. Figure 5 is an example of such a graph. If there are N nodes in the graph, then there are more than M^N actor firings that must be scheduled. This exponential explosion in the number of actor firings is only made worse by having a cycle of P_j firings for CSDF actors. Remember that the balance equations determine the number of cycles for a CSDF actor. The number of firings is the repetition count r_j multiplied by the cycle period P_j. If all the periods p_{ij} for the arcs leading into a node are relatively prime, then P_j can be quite large. The problem with this explosion is that the parallelism expressed in the dataflow graph can far exceed the parallelism available in the target hardware. It is counterproductive to expose hundreds or thousands of operations that can execute in parallel when there are many fewer processors available.

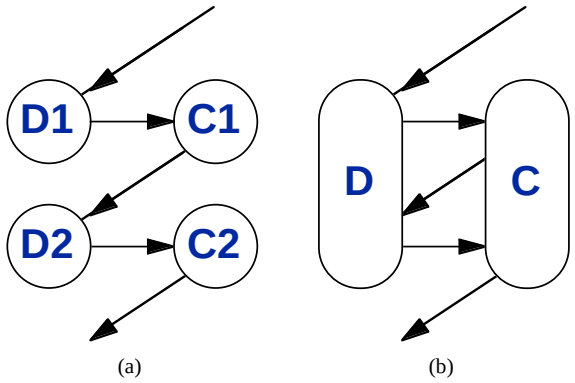


Figure 7: Deadlock is caused by a directed cycle in the precedence graph.

Incremental compilation heuristics have been developed to make parallel SDF scheduling tractable [8]. We would like to simplify CSDF scheduling and take advantage of all the scheduling techniques that already exist for SDF. To do this, we can transform a cycle of CSDF actor firings into a single SDF actor firing with a higher-order function. Instead of using the map function to form a process from an infinite number of actor firings, we use the loop function to define a new actor g that is equivalent to N consecutive firings of the original actor f.

$$\text{loop}(\text{f}, N)[x_1, x_2, x_3 \ldots] = [\text{f}(x_1), \text{f}(x_2), \text{f}(x_3) \ldots, \text{f}(x_N)]$$

By choosing $N = P_j$, we force all firings of a cycle to be scheduled together and transform the CSDF actor into a SDF actor that implements a cycle of firings.

One pitfall of this transformation is that it may introduce deadlock, as in figure 6. The repetition vector for this graph, $\vec{r} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$, specifies that there should be one cycle of each actor, and each actor has two firings in a cycle. The precedence relationships for this CSDF graph are shown in figure 7(a). When the firings of a cycle are combined into a single firing, deadlock is caused by the introduction of a directed cycle in the precedence graph in figure 7(b). We can safely transform CSDF actors that are not in a directed cycle of the dataflow graph. However, when an actor is part of a directed cycle, we might introduce deadlock as just demonstrated. In such cases, we must test the resulting CSDF graph for deadlock using more sophisticated methods [1].

This transformation from CSDF to SDF reduces the number of operations that must be scheduled, and allows us to use the many existing SDF scheduling techniques. But we have seen that transforming a CSDF graph into a SDF graph can introduce deadlock. There are other situations where it is undesirable to perform this transformation, as we shall see in the following examples.

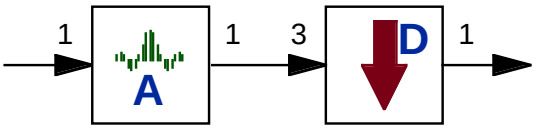


Figure 8: An FIR anti-aliasing filter followed by a 1:3 downsampler.

5 Dead code elimination

Consider the simple 1:3 downsampling operation shown in figure 8. Actor A implements a low-pass finite impulse response (FIR) filter to prevent aliasing. In SDF the downsampling actor D consumes 3 input tokens and produces 1 output token. The precedence graph for this system is shown in figure 9(a): three firings of A precede a single firing of D. Notice that the output is not available until after all firings of A and D.

In CSDF the downsampling actor D has a cycle of 3 firings. It consumes 1 input token and produces 1 output token in the first firing, then it consumes 1 input token and produces no output tokens in the next 2 firings. Thus the single firing of D in figure 9(a) expands to a cycle of 3 firings in figure 9(b). Notice that now the output is available as soon as A and D have each fired once and that the results from the remaining firings of A are unused. Thus, not only can CSDF reduce the critical path by making the result available earlier, it may also uncover unnecessary computations that can be eliminated.

To eliminate unnecessary computations, or dead code, find the nodes in the precedence graph that have no successors — sink nodes. If these nodes have no side effects (such as input/output operations on an external device) then they can be removed from the graph. By removing these nodes, we may create new sink nodes. If these new sink nodes have no side effects, then they can also be removed. This pruning operation can continue following the precedence graph.

Actors with internal state, like actors with side effects, cannot be removed from the precedence graph. Internal

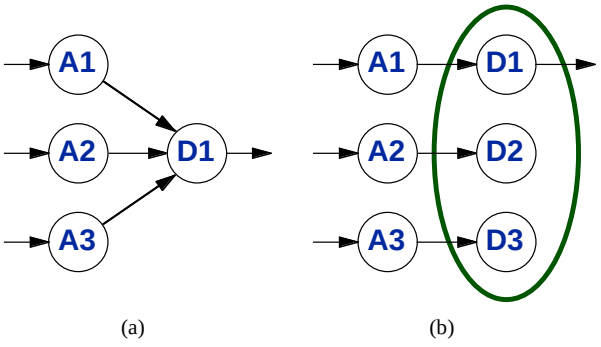


Figure 9: The SDF and CSDF precedence graphs for downsampling.

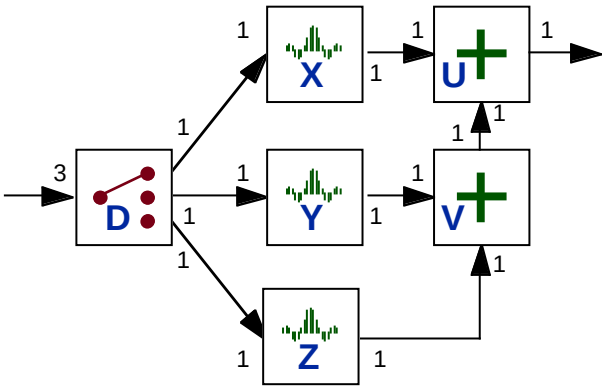


Figure 10: Polyphase 1:3 downsampling.

state can be made explicit with an external self-loop in the dataflow graph, as shown in figure 1(c). This self-loop introduces dependencies between the successive firings of an actor, as we will see later in figure 14. These nodes cannot be eliminated because they always have a successor in the precedence graph and do not appear as sinks. However, the internal state used to enforce the firing sequence of a CSDF actor is an artifact that can be optimized away.

Dead code elimination can also be applied directly to a dataflow graph before constructing its precedence graph — any sink nodes with no side effects and no internal state can be removed from the dataflow graph. Only for CSDF can additional dead code be eliminated by examining the precedence graph. This is another example where the CSDF to SDF transformation may be undesirable.

6 Parallelism

To overcome the limitations that SDF places on dead code elimination, we can use a polyphase filtering algorithm [11] for downsampling as shown in figure 10. This polyphase implementation is more efficient because it avoids the computation of unused results. There is no dead code to eliminate. Upsampling is a different problem and dead code elimination does not apply. In a direct implementation, zero-valued samples are inserted between the original

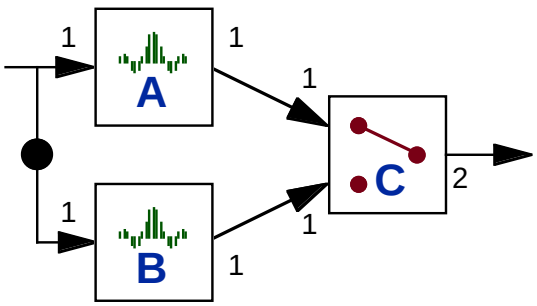


Figure 11: Polyphase 2:1 upsampling.

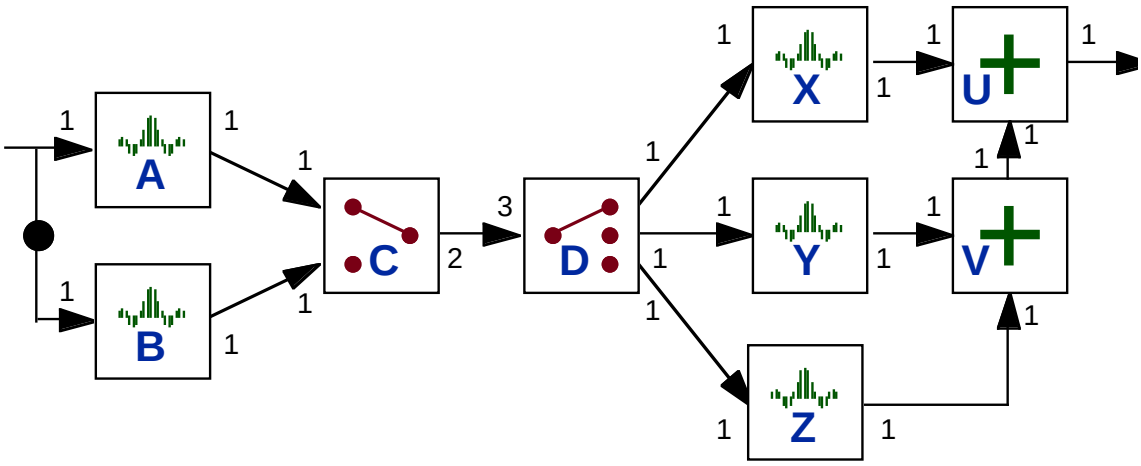
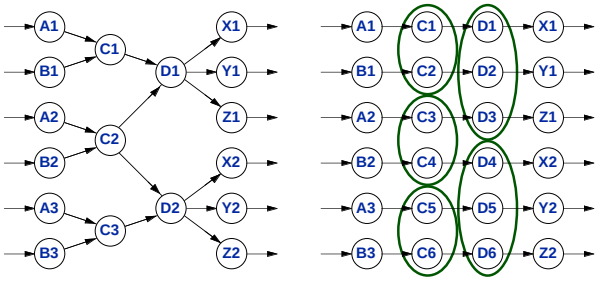


Figure 12: Polyphase 2:3 sample rate conversion.

samples of a stream, and then a filter performs interpolation. This is inefficient because multiplications and additions are being performed on many zero-valued inputs. A polyphase implementation of upsampling is shown in figure 11. Upsampling and downsampling can be combined to provide rational sample rate conversion, as shown in figure 12.

Compare the precedence graphs for SDF and CSDF implementations polyphase filtering in figure 13. Notice that for SDF the each firing of X, Y and Z depends on four firings of A and B while for CSDF the precedence relationships are much simpler: each firing of X, Y and Z depends only on one firing of A or B. There is more exploitable parallelism in the CSDF implementation of this system. If every actor had internal state, the CSDF precedence graph would be as shown in figure 14. Notice that now the second firing of Z depends on all firings of A and B. Knowing which actors do not have internal state is crucial. Actors with internal state must be fired sequentially instead of in parallel, severely limiting the opportunities for optimization.



(a) (b)

Figure 13: The SDF and CSDF precedence graphs for polyphase filtering.

7 Conclusion

We showed how CSDF actors can be transformed into SDF actors through the use of the higher-order function loop. This transformation reduces the number of actor firings that must be scheduled, and allows us to make use of existing SDF scheduling techniques. However, care must be taken when applying this transformation to avoid introducing deadlock .

There are some genuine advantages that CSDF has over SDF. We showed how to eliminate dead code and how to expose additional parallelism. These advantages could be lost if we transformed every CSDF actor into an SDF actor, so this transformation is not always beneficial. Previous work on CSDF [3, 1] has made the restrictive assumption that all actors have internal state, effectively adding a self-loop in the dataflow graph, as in figure 1(c), and additional dependencies in the precedence graph, as in figure 14. In the absence of additional information about which actors do or do

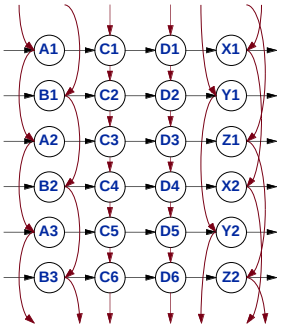


Figure 14: Added constraints in the CSDF precedence graph when every actor is assumed to have internal state.

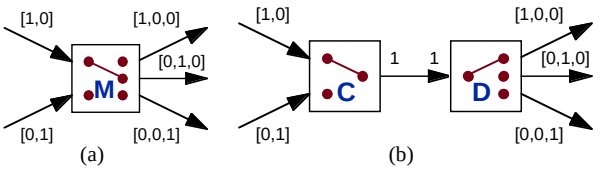


Figure 15: The CSDF mixer and an equivalent combination of the commutator and distributor.

not have internal state or side effects that require sequential execution, this is the safest way to ensure a correct execution order. However, this hides many of the advantages of the simpler precedence relationships of CSDF. None of the optimizations we have discussed are possible without knowing which actors have internal state and/or side effects. In the Ptolemy system[9], our approach is to have the designer of an actor specify attributes for it. Thus the designer can assert whether or not an actor has internal state or side effects.

Our loop transformation allows us to use existing SDF scheduling techniques for CSDF graphs. But it also hides some important advantages of CSDF. Instead of developing schedulers that exploit the full generality of CSDF, we could extend existing SDF schedulers to treat certain multirate actors as special cases. In fact, we need only one multirate actor: the mixer [10], shown in figure 15(a). The mixer is a generalization of the distributor and commutator. It can have any number of inputs and outputs, and is functionally equivalent to a combination of a commutator and distributor, as shown in figure 15(b). Because commutators and distributors are sufficient for building any multirate system [11], we could use a simple dataflow model where the mixer is the only multirate actor. This would give us all the advantages of CSDF without the need to support its full generality.

Acknowledgments

This work 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), the Semiconductor Research Corporation (project 95-DC-324), the National Science Foundation (MIP-9201605), the State of California MICRO program, and the following companies: Bellcore, Bell Northern Research, Dolby Laboratories, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, and Rockwell. José Luis Pino is also supported by AT&T Bell Laboratories as part of the Cooperative Research Fellowship Program.

References

[1] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *IEEE Int. Conf. ASSP*, pages 3255–3258, Detroit, Michigan, May 1995.

[2] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer valued control signals. In *Asilomar Conf. Sig. Sys. and Comp.*, Pacific Grove, California, Oct. 1994. http://ptolemy.eecs.berkeley.edu/papers/IDF_Asilomar.ps.Z.

[3] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow: Model and implementation. In *Asilomar Conf. Sig. Sys. and Comp.*, Pacific Grove, California, Oct. 1994.

[4] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Aug. 1974. International Federation for Information Processing, North-Holland Publishing Company.

[5] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing*, pages 993–998, Toronto, Aug. 1977. International Federation for Information Processing, North-Holland Publishing Company.

[6] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, C-36(1):24–35, Jan. 1987.

[7] E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. IEEE*, 83(5):773–799, May 1995. http://ptolemy.eecs.berkeley.edu/papers/processNets.

[8] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. Technical Report UCB/ERL M95/36, University of California, Berkeley, May 1995. http://ptolemy.eecs.berkeley.edu/papers/erl-95-36.

[9] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for DSP using Ptolemy. *J. VLSI Sig. Proc.*, 9(1):7–21, Jan. 1995.

[10] S.-I. Shih. Code generation for VSP software tool in Ptolemy. Master's thesis, University of California, Berkeley, May 1994.

[11] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, Englewood Cliffs, 1993.