To appear: Third International Workshop on Hardware/Software Codesign
Grenoble, France; September 1994

A Dynamic Dataflow Model Suitable for Efficient Mixed Hardware and Software Implementations of DSP Applications

Joseph T. Buck

Synopsys, Inc., 700 E. Middlefield Rd., Mountain View, California 94043

Abstract

This paper presents an analytical model for the behavior of dataflow graphs with data-dependent control flow and discusses its suitability to the generation of efficient software and hardware implementations of digital signal processing (DSP) applications. In the model, the number of tokens produced or consumed by each actor is given as a symbolic function of the Boolean values in the system; in addition, it may vary cyclically to permit more memory-efficient multirate implementations. The model can be used to extend the ability of block-diagram-oriented systems for DSP design, such as Ptolemy [1], to produce efficient hardware and software implementations; this permits the hardware-software codesign techniques of [2] to be efficiently targeted at a wider class of problems, those involving some asynchronous behavior, for example.

1. The role of models in the codesign problem

The design problem addressed in this paper concerns special-purpose systems for digital signal processing applications that may be composed of one or more general-purpose processors, programmable digital signal processors, application-specific integrated circuits (ASICs), or a mixture. Thus when these techniques are applied to mixed hardware-software systems, we are doing codesign in the sense of [3].

Kalavade and Lee [2] present two opposing approaches to system-level design: a unified approach, which seeks a consistent semantics for specifying the whole system, and a heterogeneous approach, such as that of Ptolemy [1], which seeks to systematically combine disjoint semantics, and they favor the heterogeneous approach. Nevertheless, most approaches to hardware-software codesign, including those based on Ptolemy, start by finding a single model for expressing the functionality of hardware and software. There are a variety of such models available; for example, Lee and Messerschmitt's synchronous dataflow (SDF) model, used in [2]; the communicating codesign finite state machine (CFSM) model of [3], and languages that specify timing and state relationships such as Esterel [4] and Statecharts [5]. Related to Esterel are the stream-based languages Signal [6] and Lustre [7], which specify abstract timing relationships on streams.

It is apparent, though, that the applicability of a model is not dependent on whether the implementation target is hardware or software, but rather on the functionality to be represented. Dataflow models are effective for representing most types of DSP applications whether the implementation target is software or hardware, and likewise hierarchical finite state machines (FSMs) are effective for control-dominated systems. System designers frequently find themselves using both types of models, for example, some industrial systems designers use dataflow-based tools and FSM-based tools together on different parts of the same problem. Therefore dataflow-based models should not be seen to be rivals of FSM-based models, but rather as complementary. Just as the CFSM model introduces some features of dataflow to a pure communicating FSM model, this work introduces some data-dependent decision-making to a purely synchronous dataflow model.

DSP designers have found it convenient to work directly with dataflow graphs, and not merely use them as an underlying representation (as is done in Silage [8] and other dataflow languages); hence the popularity of commercial tools like COSSAP, SPW, Hyperception, and so forth. Accordingly, the model described here permits control structures to be built up from dataflow graphs rather than the other way around.

2. Synchronous dataflow

Synchronous dataflow [9] is a restricted form of the dataflow model of computation [10]. In the dataflow model, a program is represented as a directed graph. The nodes of the graph, also called *actors*, represent computations and the arcs represent data paths between computations. In SDF, each node consumes a fixed number of data items, called *tokens* or *samples*, per invocation and produces a fixed number of output samples per invocation. Figure 1 shows an SDF graph that has five actors. Each arc is annotated with the number of samples produced by its source actor and the number of samples consumed by its sink actor. SDF graphs may represent manifest iteration. In the example, actor 2 clearly must execute ten times as often as actor 1, for example.

When an SDF graph is to be executed repeatedly, the compiler should construct just one cycle of a periodic schedule. The first step is to determine how many invocations of each actor should be included in each cycle. This can be determined using information about the number of samples consumed and produced. Consider the connection of three actors shown in figure 2. Let I_i denote the number of tokens consumed by the i^{th} actor, and O_i denote the number of tokens produced by the i^{th} actor, as shown in figure 2. Let r_i denote the number of times the i^{th} actor is repeated in the each cycle of the iterated schedule. Then it must be true that

$$r_1 O_1 = r_2 I_2 \;\; ; \;\; r_2 O_2 = r_3 I_3 \tag{1}$$

These two equations ensure that the number of tokens produced on each arc is equal to the number consumed on that arc in each cycle of the iterated schedule. Indeed, the first step in finding a schedule for an SDF graph is to solve a set of such equations, one for each arc in the graph, for the unknowns r_i.

These equations can be written concisely by constructing a *topology matrix* Γ that contains the integer O_i in position (j, i) if the i^{th} actor produces O_i tokens on the j^{th} arc. It also contains the integer $-I_i$ in position (j, i) if the i^{th} actor consumes I_i tokens from the j^{th} arc. For example, the nested iteration shown in figure 1 has the following topology matrix:

$$\Gamma = \begin{bmatrix} 10 & -1 & 0 & 0 & 0 \\ 0 & 10 & -1 & 0 & 0 \\ 0 & 0 & 1 & -10 & 0 \\ 0 & 0 & 0 & 1 & -10 \end{bmatrix} . \tag{2}$$

Then the system of equations to be solved is

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

where $\vec{o}$ is a vector full of zeros, and $\vec{r}$ is the *repetition vector* containing the r_i for each actor. Printz calls (3) the “balance equations” [11]. For the topology matrix in (2), one solution is

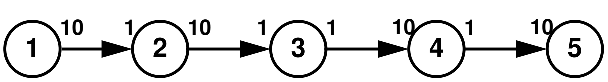


Fig. 1. Nested iteration expressed as an SDF graph.

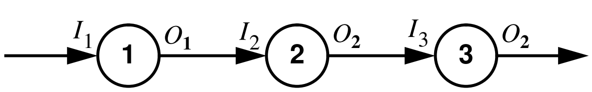


Fig. 2. : Three actors annotated with the number of tokens transferred by each actor on each arc.

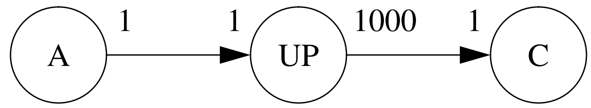


Fig. 3. An SDF graph with a large sample rate change. C’s input requires excessive memory.

$$\vec{r} = \begin{bmatrix} 1 & 10 & 100 & 10 & 1 \end{bmatrix}^T . \tag{4}$$

In fact, this solution is the smallest one with integer entries. For a connected SDF graph, it is shown in [9] that a necessary condition to be able to construct an admissible periodic schedule for a connected SDF graph is that null space of Γ has dimension one. From (3) we see that $\vec{r}$ must lie in the null space of Γ. It is also shown in [9] that when this condition is met, there always exists a vector that contains only integers and lies in this null space.

SDF and related models have been studied extensively in the context of synthesizing assembly code for signal processing applications, for example [11][12][13], as well as for hardware synthesis [14][15], particularly from the Silage language [8], which has SDF semantics. Because of the restrictions imposed by the SDF model, powerful methods exist for determining whether the graph is consistent and deadlock-free, for scheduling resources such as registers and memories, and for scheduling the execution of the graph on one or more processors. For hardware applications, the problem of scheduling the execution of the graph on a limited number of resources, such as adders and multipliers, can be treated in a similar way.

3. Limitations of the SDF model

There are two main problems with the SDF model. The first problem is that many DSP problems require some non-synchronous and data-dependent behavior; for example, timing recovery in a modem. This problem can be addressed by extending the model to permit some actors with data-dependent behavior. The second problem is that even for algorithms that can be expressed in the SDF model, there are circumstances, generally involving large sample rate changes, where using SDF constrains admissible solutions in such a way that more memory is required than is truly necessary. For example, consider the graph in figure 3. Here, implementing the upsample actor as an SDF actor requires a large memory to hold all of the output tokens from a single firing, while an alternate dynamic implementation with internal state that transfers tokens one at a time would require far less memory. These problems could be dealt with by switching to a fully dynamic model of execution, but in many circumstances the costs for doing so would be prohibitive. Instead, we wish to extend the model to deal with the above problems without abandoning static scheduling, so that implementations with the least possible overhead can be produced.

The first problem, extending the SDF model to support some dynamic actors while preserving static scheduling as much as possible, has been dealt with in [16] and [17] (and earlier in [18]) by extending the class of admissible actors to those whose dataflow behavior is affected by Boolean-valued control tokens, and demonstrating how to extend the analysis and scheduling techniques of SDF to handle the more general case. Figure 4 shows the behavior of two such actors that are widely used (dating back at least to [10]) called SWITCH and SELECT. This Boolean-controlled dataflow (BDF) model is the model we will extend further to address the remaining problems described.

The second problem, that of rate conversion actors that can be implemented more efficiently if permitted to execute in multiple phases, has been addressed by extending SDF to support so-called "cyclostatic" actors [19]. A cyclostatic or multiphase actor can be considered to cycle though a series of phases. The number of phases is fixed, and the number of tokens produced or consumed on a given port is fixed, but different phases may have different I/O behavior. Thus, for example, a cyclostatic upsample actor would consume an input only on its first phase and produce an output on every phase.

For cyclostatic dataflow (CSDF) a conceptual shift has been introduced. We can consider a dataflow actor to represent a mapping from one or more input streams to one or more output streams, rather than simply as a device, possibly with internal state, that consumes and produces tokens according to a certain rule. It is the streams that are computed that are what we are interested in, and replacing an SDF actor with a finer-grain object that performs the same stream mapping does not change the streams.

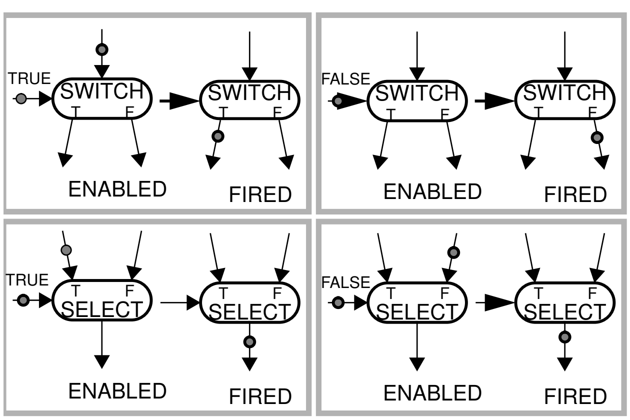


Fig. 4. SWITCH and SELECT: these actors consume and produce tokens depending on the value of Boolean control inputs.

4. The token flow model extended to support multiphase actors

An important relationship between the BDF and cyclostatic models is that both consider the behavior of minimal cyclic schedules: minimal sequences of actor executions that restore the dataflow graph to its initial state (which in the case of the cyclostatic model includes the phases of all cyclostatic actors). We will present a generalized version of the token flow model of [17] that subsumes the cyclostatic actors of [19] into a unified framework. We will treat cyclostatic actors simply as alternate implementation choices for ordinary synchronous dataflow actors. Consider the two implementations of the DOWNSAMPLE actor in figure 5. Both perform the same functional mapping from the input stream to the output stream, but the multiphase implementation may have advantages: lower latency (since output is produced after receiving only one input token instead of four) and smaller buffer size (if the adjacent actors transfer one token per execution). Multiphase implementations of dynamic actors (actors with data-dependent behavior) will also be supported.

The analysis techniques to be presented provide:

- Techniques for finding minimal cyclic schedules, the shortest execution sequences that return the dataflow graph to its original state, by solving generalized balance equations.
- Clustering techniques for finding control structures in the graphs and mapping them onto memory-efficient implementations.

We will use the term *port* to refer collectively to inputs and outputs of actors. Each arc of the dataflow graph is connected to one output port and one input port. We will use the verb *transfers* rather than "produces or consumes" when referring to both input and output ports.

In the SDF model, each port of each actor transfers a constant number of tokens on each actor execution. The generalized Boolean-controlled dataflow (BDF) model permits the following additional I/O behavior:

- The number of tokens consumed by an input port may be a two-valued function of the value of a Boolean token that is received by another input port (the control port) of the same actor. Such a port is a conditional port.
- The number of tokens produced by an output port may be a two-valued function of the value of a Boolean token that may *either* be received by another input port,

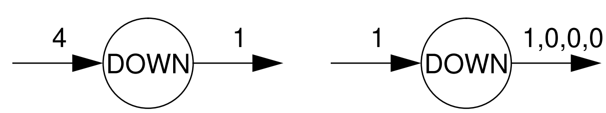


Fig. 5. Two implementations of a downsample-by-4 actor: SDF and multiphase.

or produced by another output port, of the same actor. Such a port is also a conditional port.

- Control ports are never conditional ports, and always transfer exactly one token per execution.

An actor that has at least one conditional port is called a dynamic actor. We now add the possibility of multiphase implementations of actors, with the following rules:

- A multiphase actor with *N* phases, when all *N* phases are executed in sequence, always has exactly the same I/O behavior as an ordinary BDF actor. The number of phases is always known at "compile time."
- The number of tokens transferred by a port may be specified as a vector (a list of integers), indicating the number of tokens to be transferred in successive phases of execution (the port is then a multiphase port). The length of the vector determines the number of phases. The number of phases of the actor is the least common multiple of the number of phases of each port.
- Dynamic multiphase actors are simply multiphase implementations of conventional BDF actors. Control ports for such actors transfer only one token even as *N* phases are stepped through. Control ports must transfer a control token in the first phase and in no other phases (the vector must look like 1,0,0,...).

As for the token flow model of [17], dataflow graphs that obey the model described above could be dynamically scheduled by a process that "looks" only at the Boolean control streams. The restrictions on control ports ensure that Boolean values that affect execution are always visible before they are needed to decide what actor to execute next. The change is that multiphase actors (which are no longer called "cyclostatic" as they may now have data-dependent behavior) have been added. The addition of the multiphase actors does not complicate the analysis of dynamic dataflow graphs appreciably, and the primary benefit, as expected, is that implementations that require less latency and memory can be produced.

The first step in the analysis will be a demonstration of how to determine the properties of minimal cyclic schedules of the system by solving the balance equations and determining the constraints on such schedules. Since a minimal cyclic schedule must step through all phases of execution of each multiphase actor, the existence of multiple phases does not affect this part of the analysis.

The balance equations require that, if a cyclic schedule exists, the number of tokens produced on an arc must equal the number of tokens consumed. This can be checked by relating the number of tokens consumed and produced to the number of TRUEs produced in the Boolean streams in the system. This is shown for the SWITCH and SELECT actors in figure 6. Here, p_i is the proportion of TRUEs in the Boolean stream b_i, which supplies the

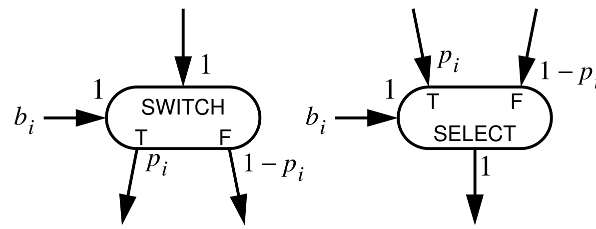


Fig. 6. : Dynamic dataflow actors annotated with the average rate of tokens produced or consumed per firing as a function of p_i, the proportion of tokens from the stream b_i that are TRUE.

control inputs. Since an integer number of Boolean tokens will be produced on each stream, it follows that

$$p_i = \frac{t_i}{n_i} \tag{5}$$

where n_i is the number of Boolean tokens produced (and consumed) on the Boolean stream b_i during the cycle, and t_i is the number of the tokens that are true. Furthermore, we will see that we never need to know or estimate the numerical value of p_i. All manipulations that use it can use it symbolically.

Consider the if-then-else program shown in figure 7. The SWITCH directs the incoming token to one of two subsystems, 3 or 4, depending on the Boolean supplied by actor 7. Since the Boolean is supplied to two actors, it is implicitly forked into two Boolean streams labeled b_1 and b_2. It will be important to recognize these as two Boolean streams. In this figure, the numbers adjacent to the arcs will be used only to identify them.

The topology matrix $\Gamma(\vec{p})$ for this system is given by:

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & (1-p_1) & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & (p_2-1) & 0 & 0 \\ 0 & p_1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -p_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

Here, $\vec{p}$ is a vector with all Boolean probabilities. Analogous to (3), we need to find a vector $\vec{r}(\vec{p})$ such that

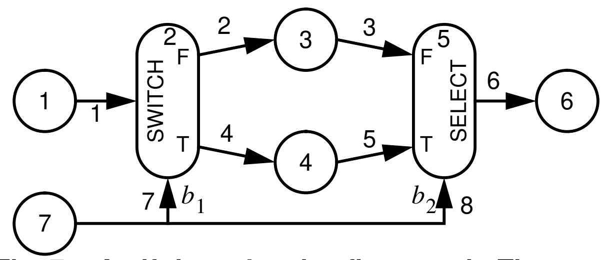


Fig. 7. : An if-then-else dataflow graph. The numbers adjacent to the arcs merely identify them.

$$\Gamma(\vec{p})\,\vec{r}(\vec{p}) = \vec{o} \tag{7}$$

$\Gamma(\vec{p})$ must have a null space of rank 1. It is easy to verify that this is true if and only if

$$p_1 = p_2, \tag{8}$$

which fortunately is true trivially, since b_1 and b_2 emanate from the same actor. The solution space is given by

$$\vec{r}(\vec{p}) = k\begin{bmatrix}1 & 1 & (1-p_1) & p_1 & 1 & 1 & 1\end{bmatrix}^T. \tag{9}$$

Note that, regardless of the exact value of $\vec{p}$, there are nonzero solutions for (9). In [18], graphs with this property are called *strongly consistent* while graphs where nonzero solutions exist only for some values of $\vec{p}$ are called *weakly consistent*. In strongly consistent graphs, there is a balance of long-term flow rates, however this says nothing about whether bounded memory implementations are possible (see [20] for an example of a strongly consistent system that requires unbounded memory and [17] for a thorough analysis of conditions for bounded memory).

Now, we express p_1 in terms of t_1 and n_1, and note that $k = n_1$ and $kp_1 = t_1$ in the above. We obtain

$$\begin{bmatrix}n_1 & n_1 & n_1 - t_1 & t_1 & n_1 & n_1 & n_1\end{bmatrix} \tag{10}$$

and note that the minimal integer solution, valid for all outcomes, is $n_1 = 1$, and, as intuition would suggest, actor 3 is fired only if the Boolean token is false, actor 4 only if the Boolean token is true. Thus minimal schedules are bounded in length, which is sufficient (but not necessary) for a bounded memory implementation. Examples of graphs that do not have bounded-length schedules, or that require unbounded memory, are given in [17].

Addition of multiphase actors to the model does not complicate the solution of the balance equations at all. Since a cyclic schedule requires that all phases of any multiphase actor be executed, for purposes of solving the balance equations we can replace each multiphase actor with an equivalent single-phase actor.

5. Clustering of dynamic dataflow graphs

We will now present a algorithm for clustering a dynamic dataflow graph that follows the rules we have described to find control structures such as iteration, if-then-else, and do-while. It is a generalization of the algorithm described in [17]. In that work, clustering is motivated as a way to produce bounded-memory implementations for graphs that contain data-dependent iteration. As a motivating example, consider the graph in figure 7. This graph is similar to the example in [16] but has a multiphase actor.

To solve the balance equations, we replace actor 5, the multiphase actor, with the corresponding SDF implementation, an actor that consumes two tokens and produces one. When the computation described in the previous section is carried out, we obtain a repetition vector of

$$\begin{bmatrix}n & n & n & t & \frac{(n-t)}{2} & \frac{(n-t)}{2}\end{bmatrix}, \tag{11}$$

where n is the number of executions of actor 3 and t is the number of Boolean tokens produced that are true. Note that there is no way we can assure that this vector is integral-valued for any bounded n; from the form or from the graph itself we observe that a complete cyclic schedule must include an even number of false tokens. The dotted curve in the graph suggests a solution: we can produce a bounded-length schedule for the subgraph within the curve. If this subgraph is repeatedly executed until a token emerges (by conceptually surrounding it with a do-while loop), and we treat the looped cluster as a dataflow actor that produces one token, we can produce a schedule for the inner graph and another schedule at the top level.

If the intent were to schedule this graph for a single programmable processor, an implementation like the following would be desirable:

```
for i from 1 to 2
        repeat
                actor 1;
                b = actor 3;
                actor 2;
                if (b) actor 4;
        until (not b);
        actor 5 (phase i);
};
actor 6;
```

Even where the target for implementation is multiple processors, custom hardware or a mixture, we will find that this type of nested control structure is a useful intermediate representation.

We will now describe the clustering algorithm in more detail. There is insufficient space to give a complete description; for more detail see [17]. We say that two actors are *adjacent* if there is an arc that connects them. With respect to this arc, we call the actor that produces tokens on the arc the *source actor* and the actor that con-

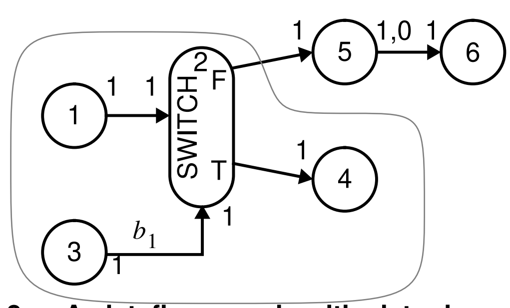


Fig. 8. : A dataflow graph with data-dependent iteration that illustrates the use of clustering.

sumes tokens from the arc the *destination actor*. Two adjacent actors have the *same repetition rate* if the number of tokens the source actor produces on an arc is always equal to the number of tokens the destination actor consumes from the arc (for conditional or multiphase ports, the conditions and phases must match). Finally, we will call an arc a *feedforward arc* if it is not part of a directed cycle of arcs, or equivalently if there is no directed path of arcs from the destination actor to the source actor. An arc that is not a feedforward arc is called a *feedback arc*.

We will assume that the graph is connected, strongly consistent, and deadlock-free. If this is true, then we can assure that certain problematic situations do not occur — for example, we will never have a pair of adjacent actors that are "the same repetition rate" with respect to one arc that connects them, but not with respect to another connecting arc (this would lead to inconsistency). We would also never have arcs connecting the actors in both directions, with no initial tokens on any arc (this would be a delay-free loop and would cause a deadlock), unless the presence of conditional arcs mean that a true deadlock does not occur. It is not difficult to drop these assumptions and detect these conditions as errors with slight modifications to the algorithm; these modifications insert extra checks before a pair of actors is combined into a single cluster to test for deadlock or inconsistency.

Our algorithm consists of two alternating phases: a merge pass and a loop pass. The method is related to the loop scheduling of [21], but generalized beyond SDF graphs. The merge pass attempts, as much as possible, to combine adjacent actors that have the same repetition rate into clusters. The loop pass may make an actor or cluster conditional, add repetition by a constant factor, or add a do-while loop, as appropriate to enable additional merge operations to take place in the next merge pass.

The merge pass combines two adjacent actors (or clusters) into a single cluster provided that their repetition rates are the same, the merge operation cannot introduce a deadlock (because there is no directed path that begins at the source actor, passes through a third actor not involved in the merge, and ending at the destination actor), and any conditional ports that remain as external to the cluster have control ports that are also external to the cluster. The property of a FORK actor that all output streams are identical to the input stream may be exploited in providing an externally visible control port. Recall that we permitted control ports of conditional output ports to be outputs; such cases frequently arise during clustering. If two actors are connected by an arc that has a multiphase port attached to one end, a merge may not take place unless there is also a multiphase port at the other end with the exact same I/O pattern.

This is a complex set of conditions that may require repeated searching of the entire graph for paths. Fortunately, in most cases it can quickly be determined that two actors may be merged based only on local information. If all outputs of the source actor S connect directly to the destination actor D, or all inputs to D connect directly to S, and at least one of the connecting arcs has no initial tokens, then merging cannot possibly create deadlock. Since this is a very common case, the merge pass is split into a "fast" phase that does as many merges as possible based on local information followed by a "slow" phase that searches for indirect paths and remaps control arcs where needed (after the size of the graph has been reduced by the application of the fast phase). The loop pass introduces repetition by fixed factors, conditionals, and do-while operations. Integral rate changes occur where the number of tokens produced by the source port evenly divides the number consumed by the destination, or vice versa. Provided that there is not a difference in conditionals (indicating that an if-then-else or a do-while should be introduced), the loop pass will add a repetition factor to one cluster to match rates.

The loop pass turns multiphase actors or clusters into "ordinary" clusters by causing the cluster to be repeated *N* times, where *N* is the number of phases.

In cases where a port at one end of an arc transfers tokens unconditionally and a port at the other end of the arc transfers tokens conditionally, and the constant term is the same, the loop pass may create a conditional. Normally, the creation of a conditional requires the addition of a control arc, as we shall see. Under other circumstances, particularly where the constant term is different, a do-while loop is created instead. Finally, the loop pass may do a merge and create a do-while loop in a single step, as we shall see below. The detailed conditions for the creating of conditionals do not change with the addition of multiphase actors (multiphase actors are treated as if they were the equivalent ordinary BDF actors obtained by executing all phases consecutively) and are described in [17].

Consider the graph in figure 8.We can merge actors 1 and 3, since they have the same repetition rate. Actor 4 can be made conditional on the Boolean control stream. The result is shown on the left of figure 9(a) below. The conditional cluster can then be merged with 1 and 3, resulting in the graph on the right side of figure 9. At this stage, it

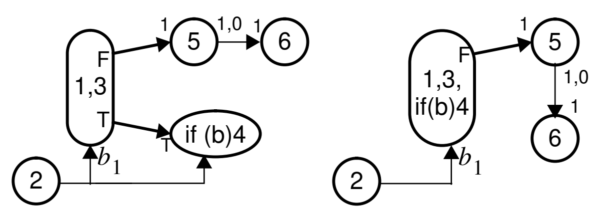
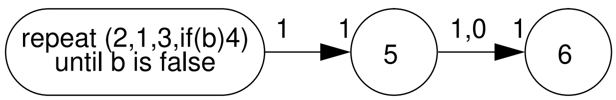
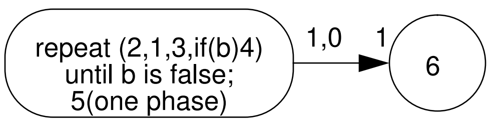


Fig. 9. :Stages in the clustering of the graph in figure 8.

would appear that either another conditional or a do-while loop needs to be introduced. Because there is a rate change in the multiphase actor, we prefer the do-while loop. We cannot simply merge actor 2 with the cluster, since that would leave a conditional port with no conditional control signal. However, in this case, adding a do-while loop would make the newly created cluster's external port unconditional, so we permit the merge and the creation of the do-while in one step. We now have



Since the input to actor 5 has a matching rate, we can merge it with the main cluster:



The loop pass will now repeat the main cluster twice to convert it from a multiphase actor to an ordinary SDF actor that outputs one token. The result will be merged with actor 6 to complete the clustering. Because a multiphase implementation of actor 5 is used, no buffer in the system need hold more than one sample; in addition to saving memory, this eliminates the need for complex addressing schemes such as indexes into circular buffers or hardware shift registers to implement queues of tokens.

An important characteristic of the nested structure built up by the clustering algorithm is that, at each level, all subclusters have the same repetition rate, so that a valid sub-schedule can be formed for each level simply by performing a topological sort of the data dependencies and "executing" the sub-clusters in a valid order.

The fast merge pass is $O(v+e)$, where v is the number of nodes and e is the number of edges, while the slow merge pass is $O(e(v+e))$ .The loop pass is $O(v+e)$. In the possible case, the slow merge pass and loop pass might need to be alternated to merge each node, giving cubic behavior $O(ve(v+e))$ for the whole algorithm. This can only happen if the number of required rate changes grows with the graph size. In typical cases, the initial fast merge pass reduces the graph size greatly and the number of rate changes and conditionals is small so that typical cost grows much more slowly and the algorithm has quadratic behavior.

6. Producing implementations from the model

This section will point the way to a variety of implementation paths that can be pursued starting from the data structures presented in this model.

The extended token flow model presented here subdivides dynamic dataflow graphs neatly into three categories

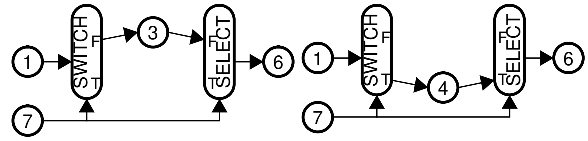


Fig. 10. Acyclic precedence graphs for the if-then-else construct. The left graph corresponds to the production of a FALSE token, the right graph to the production of a TRUE token.

(the same as for the original model of [17]):

- Those that have bounded cycle length (in that the solution to the balance equations yields bounded integer solutions). This set is a superset of SDF graphs; "if-then-else" forms also fall into this set.
- Those that can be completely clustered. For this set, all memory resources and hardware registers can be allocated at "compile time."
- All others. These will require dynamic scheduling, however, any successfully clustered sub-components can be statically scheduled.

For single-processor code generation targets, it is straightforward to generate code from the clustered structure produced by the algorithm outlined above, in a code generation environment such as Ptolemy [1] or DESCARTES [13] simply by emitting the correct control structures.

We will briefly consider code generation for multiprocessor targets. For graphs with bounded cycle length, a structure called an annotated acyclic precedence graph can be constructed giving precedence relations between actor executions and phases of execution as a function of the Boolean control values produced during the execution of a cyclic schedule of the graph. For simplicity, this can be thought of as a concise way of representing the separate, unconditional precedence graphs that would result from each possible outcome on the Boolean control streams (see figure 10). From such structures, parallel scheduling techniques such as those in [22] to obtain hard real-time multiprocessor implementations by using a minimax criterion: the graph must execute in the required time for the most expensive Boolean outcome.

For graphs that contain data-dependent iterations such as do-while loops, the minimax criterion cannot be applied as there is no upper bound to the time required for one iteration of the graph. Given probabilistic information, techniques such as those of Ha [23] can be applied, though the simplifications that are made (independence of distinct control streams) may not be applicable. Hoang's MacDAS system [24] is capable of exploiting the nested structure built up by the clustering algorithm in many cases by scheduling the clusters as units, decomposing them only as necessary for load balancing.

Homogeneous dataflow graphs (those for which every port transfers exactly one token per actor execution) are easily mapped into hardware implementations by mapping actors one-for-one into hardware. This approach corresponds fairly closely to the operation of Dennis's original static dataflow machines [10] and is provided commercially by Comdisco's HDS system as well as by Cadis COSSAP. For all but the most speed-critical applications, such implementations are too expensive, and approaches that synthesize a state machine as well as a datapath, trading off speed for area, are now becoming more common. Synopsys's Behavioral Compiler, for example, can produce such implementations for a wider class of algorithms than simple homogeneous dataflow.

For mixed hardware-software implementations, the principal reason for implementing part of the algorithm in hardware is speed. The clustering algorithm naturally separates out the parts of the problem with the highest repetition rate, making it a strong candidate for use in the sort of codesign system described in [2] as an aid to problem partitioning. Different subclusters can be mapped to hardware as speed demands.

7. Conclusions and further work

This paper presents a dataflow model in which as much compile-time scheduling work is done as possible to avoid the need for run-time scheduling in implementations. By incorporating multiphase actors into the model, more memory-efficient (and frequently more time-efficient) implementations become possible. The path to implementation for a single programmable processor is clear; for multiprocessor, hardware, and mixed hardware/software targets, we have only outlined solution paths.

Further work will focus on combining dataflow specifications from high-level tools like Ptolemy and COSSAP to produce efficient hardware implementations.

8. References

[1] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, Vol. 4, pp. 155-182, 1994.

[2] A. Kalavade and E. A. Lee, "A Hardware-Software Codesign Methodology for DSP Applications," *IEEE Design and Test of Computers,* September 1993.

[3] M. Chiodo *et al.*, "A Formal Specification Model for Hardware/Software Codesign," *Proc. International Workshop on Hardware-Software Co-Design*, Cambridge, Mass., October 1993.

[4] F. Boussinot, R. De Simone, "The ESTEREL Language," *Proceedings of the IEEE*, Vol. 79, No. 9, September 1991.

[5] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.,* vol 8, pp. 231-274, 1987.

[6] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Trans. on Automatic Control*, pp. 535-546, May, 1990.

[7] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305-1319.

[8] P. N. Hilfinger, "A High-Level Language and Silicon Compiler for Digital Signal Processing," *Proc. Custom Integrated Circuits Conf.,* IEEE Computer Society Press, pp. 213-216, 1985.

[9] E. A. Lee, D. G. Messerschmitt, "Synchronous Dataflow," *Proceedings of the IEEE*, September 1987.

[10] . B. Dennis, "First Version of a Dataflow Procedure Language," *MIT/LCS/TM-61*, Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge MA 02139, 1975.

[11] 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, May 1991.

[12] W. H. Ho, E. A. Lee, D. G. Messerschmitt, "High Level Dataflow Programming for Digital Signal Processing," *VLSI Signal Processing III*, IEEE Press 1988.

[13] S. Ritz, M. Pankert, H. Meyr, "High Level Software Synthesis for Signal Processing Systems", *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, CA, August, 1992.

[14] H. De Man, J. Rabaey, P. Six, L. Claesen, "CATHEDRAL-II: a silicon compiler for digital signal processing," *IEEE Design and Test Magazine,* pp. 13-25, Dec. 1986.

[15] J. Rabaey *et al.*, "Fast Prototyping of Datapath-Intensive Architectures," *IEEE Design and Test of Computers,* pp. 40-51, June 1991.

[16] J. T. Buck and E. A. Lee, "Scheduling Dynamic Dataflow Graphs With Bounded Memory Using the Token Flow Model," *Proc. of ICASSP '93*, 1993.

[17] J. T. Buck, "Scheduling Dynamic Dataflow Graphs With Bounded Memory Using the Token Flow Model," Memorandum No. UCB/ERL M93/69 (Ph.D. Thesis), EECS Dept., University of California, Berkeley, September 1993.

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

[19] G. Bilsen, M. Engels, R. Lauwereins, J. A. Peperstraete, "Static Scheduling of Multi-rate and Cyclo-static DSP-applications," *Proc. IEEE Workshop on VLSI Signal Processing*, to appear, 1994.

[20] G. R. Gao, R. Govindarajan, P. Panangaden, "Well-Behaved Programs for DSP Computation," *Proc. ICASSP 1992*, San Francisco, California, March 1992.

[21] S. S. Bhattacharyya, J. T. Buck, S. Ha, E. A. Lee, "Generating Compact Code From Dataflow Specifications of Multirate DSP Algorithms," Technical Report, Electronics Research Laboratory, College of Engineering, Berkeley, California 94720, May 1993.

[22] G. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication," Ph.D. Thesis, University of California at Berkeley, 1991.

[23] S. Ha, "Compile-Time Scheduling and Assignment of Dataflow Program Graphs with Dynamic Constructs," Memo. No. UCB/ERL M92/43 (Ph.D. Thesis), University of California, Berkeley, April 1992.

[24] P. D. Hoang and J. M. Rabaey, "Scheduling of DSP Programs Onto Multiprocessors for Maximum Throughput," *IEEE Trans. Signal Processing,* pp. 2225-2235, June 1993.