Presented at 28th Asilomar Conference on Signals, Systems, and Computers
November 1, 1994

Static Scheduling and Code Generation from Dynamic Dataflow Graphs With Integer-Valued Control Streams

Joseph T. Buck

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

Abstract

This paper extends the token flow model of Buck and Lee ([1],[2]), an analytical model for the behavior of dataflow graphs with data-dependent control flow, in two ways: dataflow actor execution may depend on integer, rather than Boolean, control tokens, and multiphase implementations of actors are permitted. These extensions permit data-dependent iteration to be modelled more naturally, reduce the memory required for implementations, and result in bounded-memory solutions in more cases than before. A method for generating efficient single-processor programs from the graphs is also described.

1. Introduction and motivation

Dataflow graphs have proven to be an effective representation for problems in digital signal processing, because the representation is natural to researchers and implementers (algorithms in DSP and digital communications are often expressed as block diagrams with dataflow semantics). When the actors in the dataflow graph are restricted to be synchronous, meaning that the number of data values produced by each output, or consumed by each input, of each actor are constrained to be constant and known at "compile time," it is not difficult to determine the consistency of the graph, determine its memory requirements, and schedule the execution of the graph on one or more processors. These techniques are explained in detail in [3]. Algorithms whose control flow is completely deterministic can be effectively represented using this synchronous dataflow (SDF) paradigm. Since many digital signal processing algorithms have little to no data-dependent decision-making, SDF-based tools, and tools based on dataflow languages with SDF-like characteristics such as Silage [4], have proven effective in producing software implementations ([5],[6],[7]) as well as in the synthesis of custom hardware ([4],[8]).

It is typically found, however, that some data-dependent decision-making is required in many DSP algorithms, but not much: timing recovery in a modem, for example, is one example of a portion of a problem that requires data-dependent, asynchronous sampling. Despite its limitations, SDF has some very desirable properties. It is therefore desirable to extend SDF while retaining these properties as much as possible.

1.1. The token flow model: Boolean-controlled dataflow

Actors with at least one conditional input or output port are called *dynamic actors*. The canonical dynamic actors, whose history goes back at least to [9], are SWITCH and SELECT, shown in figure 1.

In previous work ([1], and originally in [2]), SDF was extended to permit the use of a restricted class of dynamic dataflow actors, actors that fall into the category of *Boolean-controlled dataflow* (BDF) actors. Such actors are a superset of SDF actors (that includes SWITCH and SELECT) and have the following properties:

- The number of values (also called *tokens*) consumed by an input port, or produced by an output 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. One of the two values of the function is zero.
- For output ports we also permit the control port to be an output: in this case, the control token's value announces whether there are data on the conditional port.
- Control ports are never conditional ports, and always transfer exactly one token per execution.

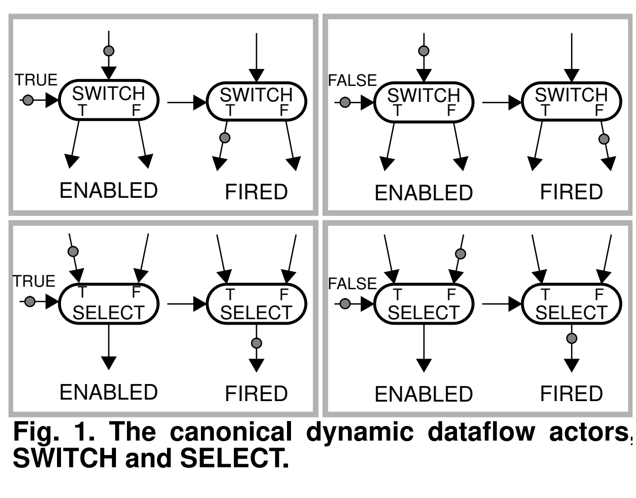

Fig. 1. The canonical dynamic dataflow actors, SWITCH and SELECT.

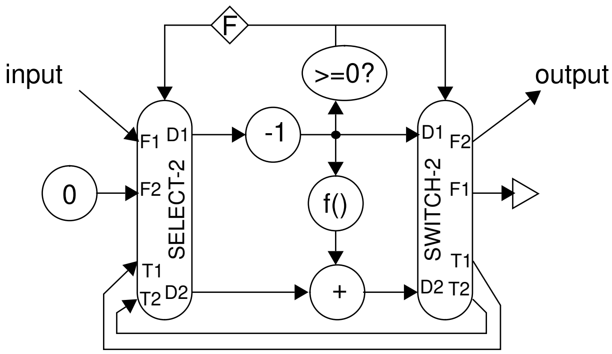

Fig. 2. This graph implements the function $g(n)$ using BDF actors. The actors SWITCH-2 and SELECT-2 switch two data streams with one control token, e.g. SWITCH-2 copies D1 to either T1 or F1 and copies D2 to either T2 or F2.

The BDF rules are designed so that the execution of actors can be scheduled by considering only the values of the Boolean tokens and the numbers of other tokens in the graph; we call this information the *state* of the graph.

1.2. Why extend the BDF model?

It is shown in [1] that the set of dataflow actors consisting of SWITCH, SELECT, and a small number of SDF actors that perform arithmetic on integers are Turing-equivalent, and therefore in a sense complete. Nevertheless, the Boolean-controlled dataflow model does not directly express certain actors that have been found to be useful. Most of these actors have the property that the control token is an integer rather than a Boolean token, which might be used in two ways:

- Specification of the number of tokens produced or consumed on some arc (e.g. a REPEAT actor, where the number of repetitions is read from an input port), or
- Enabling or disabling the arc depending on whether the token has a specific value or belongs to some set of values (as in a multi-way CASE construct).

It is possible to synthesize either a REPEAT actor or a multi-way CASE from the SWITCH, SELECT, and SDF actors. In some cases, however, the constructs that naturally arise for iterations have shortcomings. Consider the design of a subgraph that, given an integer token with value n, computes a token with value

$$g(n) = \sum_{i=0}^{n-1} f(i) \quad \textbf{(1)}$$

assuming that the function $f(n)$ is computed by an atomic actor. Let us assume that the function f is relatively expensive to evaluate, and we wish to leave open the possibility that the f evaluations be computed in parallel. We could produce a subgraph that implements this function using the BDF model by constructing a DO-WHILE loop

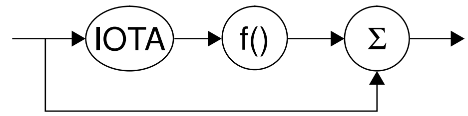

Fig. 3. This graph computes the same function as figure 2 using coarser-grained actors with integer-valued control tokens.

(see figure 2), but this graph implies a serial execution of the f actors, and the data dependency between the iterations is difficult to analyze away. The parallelism is more naturally expressed with actors that have integer control tokens. Consider two such actors: one that, given an integer value n, produces n output tokens with values ranging from 0 to $n-1$, and one that, given an integer value n on its control port, reads n tokens from its input data port and outputs their sum. Let us call the former actor IOTA (after the operation from the APL language that it resembles) and the latter actor SUM or Σ. Then the simple system in figure 3 naturally models the solution. While it is true that we could produce BDF systems corresponding to the actors IOTA and SUM, it would be desirable to have a model that could represent such actors directly, rather than as composite systems of simpler actors.

When we consider the solution in figure 3, we are confronted with a problem. In the graph in figure 2, all dataflow arcs can be implemented with only one storage element for each arc. However, unless a limit is placed on the size of the input, n, unbounded memory will be required to produce the implementation in figure 3. Even with a limit, large amounts of memory will be required. We therefore permit *multiphase* implementations of dataflow actors. Consider IOTA. A multiphase implementation would read the input value on its first phase, then output one value on each subsequent phase. If the SUM actor has a similar implementation, then the graph in figure 3 can be implemented with only one token per arc. This type of multiphase implementation is related to the cyclo-static synchronous dataflow model of [10], extended to permit the number of phases to be variable.

We will find it convenient to treat multiphase and cyclostatic actors simply as alternate implementation possibilities for actors that compute an entire operation in one phase, and it is possible to conceive of an implementation environment that freely substitute single-phase and multiphase implementations of the same actor as is convenient, especially for cyclostatic versions of SDF actors such as those in the Grape-II system [10].

Our extended model, which we will call integer-controlled dataflow or IDF, permits actors to have integer control tokens of the following types:

- Type 1 (CASE): the number of tokens transferred is either a constant, or zero, depending on whether the value of the control token is a member of some set.

- Type 2 (REPEAT): the number of tokens transferred is a constant multiple of the control token.

BDF actors are a special case of IDF actors. If only Type 1 control tokens are considered, there is not much new in the IDF theory: we simply have mapping functions to turn integer tokens into Boolean values, and, with respect to any controlled arc, a control token may still be regarded as "true" or "false." However, relations among Boolean streams may be more easily discovered and represented in some cases given CASE arcs.

1.3. Multiphase actors

Multiphase actors are a generalization of the cyclostatic synchronous dataflow actors that appear in Grape-II [10]. The multiphase actors we will consider implement the interface of an IDF actor by executing in more than one phase. Phases of the same actor need not be executed consecutively. The phases have the following structure:

- An optional starting phase. This phase may perform initialization functions, as well as read tokens from input ports. Specifically, it may read an integer-valued control token that specifies the number of phases. The starting phase never produces outputs.
- Intermediate phases. These intermediate phases may be all the same (example: a REPEAT star that outputs the same value once on each phase) or they may be different. Intermediate phases may read, write, and compute.
- An optional final phase. This phase may write out final results. The final phase never consumes inputs.

The scheduler functions "know" the I/O behavior of the actor during each phase. Some parts of the analysis ignore the existence of phases and treat the actor as the starting phase, intermediate phases, and final phase were always executed in succession. In all cases, this composite behavior corresponds to the general IDF actor described above.

2. Analysis questions for dataflow graphs

For all (SDF, BDF, IDF) dataflow graphs, the analysis conditions that occur when attempting to synthesize an implementation of the graph are as follows:

- Are there sequences of actor executions that return the graph to its original state? Graphs that lack such sequences (*cyclic schedules*) because of differences in flow rates are said to be *inconsistent* (see figure 4). For graphs with dynamic actors, consistency may depend on the values of the control tokens produced [2].

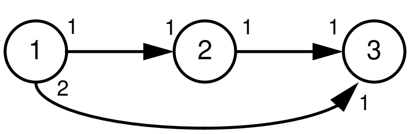

Fig. 4. An inconsistent SDF graph. Numbers adjacent to arcs give the number of tokens produced or consumed per actor execution.

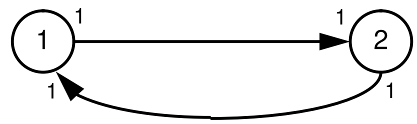

Fig. 5. A deadlocked SDF graph.

- Does the graph deadlock? A graph is deadlocked if it reaches a configuration in which no actor can be executed, as in figure 5.
- Does the graph have a *bounded-length* cyclic schedule? This means that the number of actor executions required to return the graph to its original configuration is bounded. This question is important if the graph is to be scheduled with a hard real-time constraint.
- Can the graph be scheduled to use bounded memory? In general, a bounded-length cyclic schedule implies bounded memory but not vice versa.

3. A brief review of SDF theory

For SDF graphs, algorithms exist to answer all four questions for any graph [3]. Furthermore, questions 2 and 4 become trivial: whenever cyclic schedules and exist and deadlock does not occur, all graphs have bounded length schedules and therefore require bounded memory.

In SDF graphs, the number of tokens produced by an actor on an output port, or consumed by an actor from an input port, is fixed and known at "compile time." Initial tokens on arcs, corresponding to algorithmic delays, are permitted. SDF graphs can represent manifest iteration, as in figure 6. Here 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 7. 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 the figure. Let r_i denote the number of times the i^{th} actor

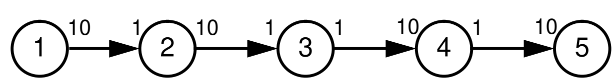

Fig. 6. Nested iteration in an SDF graph.

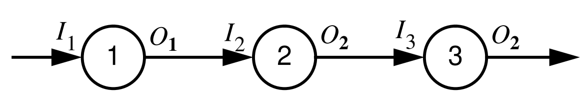

Fig. 7. : SDF actors annotated with the number of tokens consumed and produced by each actor, for use in explaining balance equations.

is repeated in the each cycle of the iterated schedule. Then it must be true that

$$r_1 O_1 = r_2 I_2 \quad ; \quad r_2 O_2 = r_3 I_3 \tag{2}$$

We can construct 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. Then the system of equations to be solved is

$$\Gamma \vec{r} = \vec{0} \tag{3}$$

where $\vec{0}$ 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" [5]. If this procedure were carried out for the graph in figure 6, one solution would be

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

This solution is the smallest one with integer entries.

For a connected SDF graph, it is shown in [3] that a necessary condition to be able to construct an admissible periodic schedule is that null space of Γ has dimension one. From (3) we see that $\vec{r}$ must lie in the null space of Γ. When this condition is met, there always exists a vector that contains only integers and lies in this null space. If there are nonzero solutions to the balance equations and there are enough initial tokens in any cycle of the graph to avoid deadlock, there are always bounded-length schedules (the length is the sum of the elements of the repetition vector) and there are always bounded-memory implementations.

4. Dynamic graphs with integer control

For dynamic dataflow graphs, the analysis questions concerning schedule length and bounded memory become more interesting, because cyclic schedules that are unbounded in length, or that may require unbounded memory on arcs, may occur. The techniques used to extend SDF graph theory to include some dynamic actors were first proposed in [2] and developed more fully in [1]. Of the methods presented in [1], we will principally be concerned with extending the following two:

- Solving the balance equations in symbolic form to determine whether bounded-length schedules exist, and
- Clustering the graph to find control structures.

4.1. Solving the balance equations

We solve the balance equations for dynamic dataflow graphs by using symbolic expressions for the number of tokens produced and consumed on conditional arcs. For BDF, these symbolic expressions were of the form p_i, which expressed the proportion (over some interval) of tokens on the Boolean control stream b_i with value TRUE. When BDF is extended to support IDF actors with

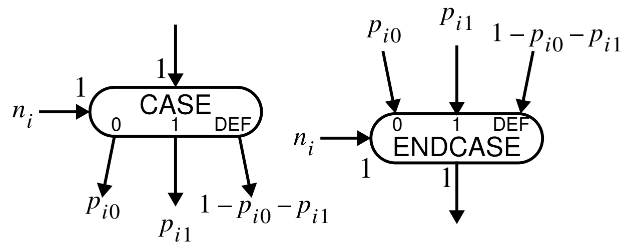

Fig. 8. The CASE and ENDCASE actors, annotated with IDF analysis quantities. The DEF (default) output is used if the control is neither 0 nor 1.

type 1 control arcs, such as the CASE and ENDCASE actors in figure 8, we have quantities of the form p_{ij} instead, designating the proportion of integer control tokens on control stream c_i whose value is j.

Using the CASE and ENDCASE actors, we can produce the three-way branch analog to the canonical if-then-else construct, as shown in figure 9. The topology matrix for this graph can be written down easily by recording the number of tokens produced or consumed by each actor on each arc. If the resulting system is solved, we can determine that the repetition vector for the graph is

$$\vec{r}(\vec{p}) = k \begin{bmatrix} 1 & 1 & p_{10} & p_{11} & (1 - p_{10} - p_{11}) & 1 & 1 & 1 \end{bmatrix}^T \tag{5}$$

Note that there are nonzero solutions for the repetition vector regardless of the value of $\vec{p}$. Graphs that have this property are called *strongly consistent* in [2]. Strong consistency implies a balance of long-term flow rates, but says nothing about bounded-length schedules or bounded-memory implementations: strongly consistent graphs can still require unbounded memory (see [1] for examples). Finding bounded complete cycles will assure bounded memory, so we shall now show how to find them.

We are interested in determining the properties of minimal complete cyclic schedules of the graph. To do this we note that quantities like p_{10} must be the ratio of two integers: the number of tokens on control stream 1 with value 0 during a complete cycle divided by the number of tokens on stream 1 in a complete cycle. Let n be the number of

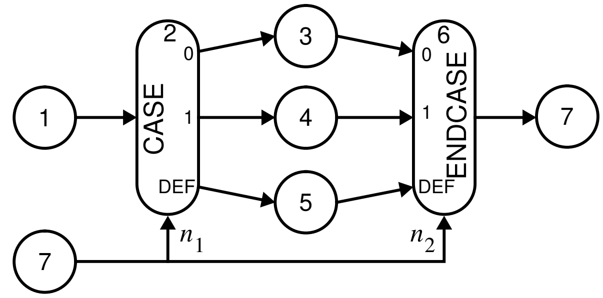

Fig. 9. A CASE construct. Here n_1 and n_2 are the control streams for the CASE and ENDCASE actors, respectively.

integer control tokens produced, and let n_i be the number of control tokens whose value is i. Then in the above equation, $k = n$ and we have

$$\vec{r}(\vec{p}) = \begin{bmatrix} n & n & n_0 & n_1 & (1-n_0-n_1) & n & n & n \end{bmatrix}^T \qquad (6)$$

We now find the smallest integer solution to determine what a minimal complete cycle consists of. It is

$$\vec{r}(\vec{p}) = \begin{bmatrix} 1 & 1 & n_0 & n_1 & (1-n_0-n_1) & 1 & 1 & 1 \end{bmatrix}^T \qquad (7)$$

where n_0 is 1 if the control token is 0 and 0 otherwise, and n_1 is 1 if the control token is 1 and 0 otherwise.

For type 2 arcs, we will find that quantities like e_i, the average (arithmetic mean) value of the integer control token on control stream c_i over some interval, appear; for example, as the label on the output of the IOTA actor in figure 3. Except in rare circumstances, any graph with a type 2 actor will have unbounded cycle length, unless some restriction is placed on how large the values of the control tokens can be. Consider figure 10. After solving the balance equations, finding the smallest integer solution, and noting that the average value of the control stream over one control token is just the value of that token, we obtain

$$\vec{r} = \begin{bmatrix} 1 & 1 & c & 1 & 1 \end{bmatrix}^T \qquad (8)$$

as the minimal repetition vector for a cyclic schedule, where c is the value of the control token produced.

Type 2 arcs, in which the number of tokens transferred on an arc is proportional to the value of an integer control token, introduce a new complication into IDF theory. If we have even a single type 2 arc in the system, we immediately have unbounded schedule length, because there is no limit on how large an integer control token's value might be. Unbounded memory can be avoided in many cases by using multiphase implementations. But there are distinct differences between a case like the IDF graph of figure 3 and a BDF graph with data-dependent iteration. The BDF graph may represent a system that never returns to its initial state; however, we are assured that the IDF system always terminates. In the IDF case, while the cycle length is not absolutely bounded, it is bounded if we possess an upper bound on the value of the computed tokens, and furthermore it is guaranteed to be finite even without such a bound. Thus for IDF we have an important distinction between "bounded length schedule" and "finite length schedule" and we can speak of bounds that are functions of the maximum values of certain control tokens.

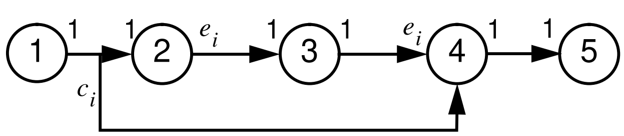

Fig. 10. In this graph, actors 2 and 4 have type 2 arcs. These are annotated by quantities e_i, the average value over the interval considered of control tokens on the stream c_i.

4.2. Clustering to find control structure

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 (over a fixed or variable range), if-then-else, and do-while. It is a generalization of the algorithm described in [1] (which, in turn, generalizes techniques in [11]). In that work, clustering is motivated as a way to produce bounded-memory implementations for graphs that contain data-dependent iteration, such as figure 2. Because of space limitations, the algorithm will only be described qualitatively here.

The goal of the clustering algorithm is to map the graph into its traditional control structures such as iteration, if-then-else and do-while, whenever possible. The substructures are treated as atomic actors from their exterior. If the interior of each control structure has a bounded cyclic schedule, the graph can be scheduled in bounded memory.

For the purposes of this discussion, 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 consumes 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).

The algorithm clusters the graph to find control structure by repeatedly applying two transformations:

- The *merge pass* combines adjacent actors with the same repetition rate into clusters. This transformation is allowed whereever deadlock is not created and required control signals are not buried inside clusters.
- The *loop pass* may make an actor or cluster conditional, add repetition by a constant factor, add a do-while loop, or (new in IDF as compared to BDF) execute a cluster n times where n is read from some arc, as appropriate to enable additional merge operations to take place in the next merge pass.
- Each transformation results in clusters that obey the rules for IDF actors; furthermore, an valid sub-schedule for each cluster can be found by a topological sort of data dependencies, since all rates within clusters match.

For BDF actors, the clustering algorithm is described in detail in [1]. There are two new features that must be supported: multiphase ports and repetition based on a run-time token. When a cluster containing a multiphase port is

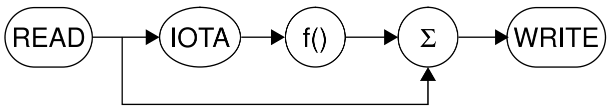

Fig. 11. This graph uses the actors from figure 3. We assume multiphase implementations of IOTA and Σ; the former writes one token per phase, the latter reads one token per phase.

modified by the loop pass, the effect is to "sum" the phases together, producing a non-multiphase port. Similarly, the loop pass can convert an actor that reads one token into an actor that reads *n* (where *n* is variable).

As an example, consider the graph in figure 11, and assume that we have multiphase implementations for IOTA and Σ. The initial phase of IOTA reads the number of phases to be executed from the input and sets an internal state variable to zero. Each intermediate phase of IOTA outputs the value of the internal state variable and increments it. The initial phase of Σ sets an internal state variable, the sum, to zero, and reads in the number of phases. The intermediate phase adds the input token to the sum. The final phase outputs the sum.

The clustering algorithm merges IOTA and f() into one cluster because their repetition rates match. The resulting cluster can be merged with Σ because the streams that provide the number of phases of the actor are identical. A *repeat_x* loop, where *x* is the input token, is then added around the cluster consisting of IOTA, f(), and Σ. The intermediate phases form the body of the loop, the starting phases come just before the loop, and the ending phases come just after. The resulting cluster reads one token and writes one, so we have a rate match. The pseudocode generated for a single-processor implementation is

```
READ x;
iota_state := 0;
sum_out := 0;
repeat x times
        iota_out := iota_state;
        iota_state := iota_state + 1;
        f_out := f(iota_out);
        sum_out := sum_out + f_out;
end repeat;
WRITE sum_out;
```

In this manner, memory-efficient single-processor implementations can be produced.

5. Further work

Parallel implementations from IDF graphs have not been addressed here. The hierarchical clustering produced by this model may be suitable for exploitation by frameworks such as those by Ha [12]. Ha's system, by incorporating probabilistic assumptions about the control constructs (if-then-else, do-while, repeat_x), generates parallel schedules for those constructs. The assumptions of independence of control streams made in that model may be unrealistic in many cases, however.

For parallel implementations, multiphase actors would supply a set of precedence relationships for the phases to aid in parallel scheduling. In some actors, intermediate phases are independent, in others, sequential dependencies will occur. In still other cases, successive phases may be partially overlapped, as in the systolic actors of the MacDAS system of [7]. In that system, hierarchical actors are scheduled as units and are expanded only as needed to accomplish load balancing. It seems reasonable to expect that the clustered data structures described here could be used instead as the units of hierarchy.

6. References

[1] J. 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.

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] J. 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.

[10] 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*, 1994.

[11] S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, "A Scheduling Framework for Minimizing Memory Requirements of Multirate DSP Systems Represented as Dataflow Graphs," in *VLSI Signal Processing VI*, IEEE Special Publications, New York, 1993.

[12] 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.