

# SCHEDULING STRATEGIES FOR MULTIPROCESSOR REAL-TIME DSP <sup>1</sup>

Edward Ashford Lee  
Soonhoi Ha

U. C. Berkeley  
Berkeley, CA 94720

GLOBECOM, Dallas, Texas, November 1989

## ABSTRACT

Real-time digital signal processing often requires multiple processors. Unfortunately, in most practical situations, partitioning of DSP applications for execution on multiple programmable processors is ad-hoc. Automatic schedulers either (1) add unacceptable cost to the implementation or (2) address only a subset of applications. This paper explores the possibilities for automatic schedulers that result in low implementation cost and can target a broad class of DSP applications. We can define four classes of scheduling strategies, (1) fully dynamic, (2) static assignment, (3) self-timed, and (4) fully static. Moving from (1) to (4), more scheduling activity is performed at compile time and less at run time. This paper argues that for most DSP applications, self-timed scheduling is the most attractive.

## 1. MOTIVATION

Use of multiple programmable processors has become an attractive alternative to custom VLSI for many real-time digital signal processing applications. Widespread use of this alternative, however, will depend on the development of effective software and hardware development environments. Fortunately, such software is being actively pursued in both research and industrial settings. Interesting systems are currently commercially available for implementing DSP algorithms on *single* processors, while some experimental systems emphasize real-time computation on *multiple* processors. An example is Gabriel, which translates block-diagram algorithm descriptions into real-time code for multiple programmable DSPs [Lee89].

One of the key problems behind such systems is scheduling for parallel computation. Compile-time scheduling promises near-optimal performance at low cost for the final system, but is only suitable for a subset of applications. Run-time scheduling can address a wider variety of applications, at greater system cost. Digital signal processing provides unique opportunities and constraints that have not been

completely addressed, either in the deterministic scheduling literature or in the multiprocessor systems literature. This paper will attempt to define these.

We assume the description of the program to be scheduled is a dataflow graph, signal flow graph, or block diagram. Language classes that are efficiently translated into dataflow graphs include functional, applicative, and single-assignment. In a dataflow graph the nodes, or *actors*, are functions that operate on data passed through the arcs. The model is *data-driven*, in that actors fire (or perform their computation), when sufficient data is available on their input arcs. The role of the scheduler is simply to determine when to fire actors and on which processor. The actors may have arbitrary granularity, meaning that they may represent atomic operations, such as addition and multiplication, or much more elaborate operations, such as transforms or digital filters. We assume that no attempt will be made to exploit concurrency within an actor, so that the scheduler only needs to operate on the dataflow graph.

## 2. A SCHEDULING TAXONOMY

Scheduling consists of assigning actors to processors, specifying the order in which actors fire on each processor, and specifying the time at which they fire. each of which can be done either at compile time or at run time. Depending on which operations are done when, we define four classes of scheduling, depicted in figure 1. The first type is *fully dynamic*, where actors are scheduled at run time only. When all input operands for a given actor are available, the actor is assigned to an idle processor and fired. The second type is *static allocation*, where an actor is assigned to a processor at compile time and a local run-time scheduler invokes actors assigned to the processor based on data availability. In the third type of scheduling, the compiler determines the order in which actors fire as well as assigning them to the processors. At run-time, the processor waits for data to be available for the next actor in its ordered list, and then fires that actor. We call this *self-timed* scheduling because of its similarity to self-timed circuits. The fourth type of scheduling is *fully static*; here the compiler determines the exact firing time of actors, as well as their assignment and ordering. This is analogous to synchronous circuits. As with any taxonomy, the boundary between these categories is not rigid.

We can give familiar examples of each of the four strategies applied in practice. Systolic arrays, SIMD (single instruction, multiple

<sup>1</sup> The authors gratefully acknowledge the support of Motorola, in cooperation with the state of California Micro program, and Darpa, under grant no. N00039-87-C-0182.

	assignment	ordering	timing
fully dynamic	run	run	run
static-assignment	compile	run	run
self-timed	compile	compile	run
fully static	compile	compile	compile

Figure 1. The time which the scheduling activities "assignment", "ordering", and "timing" are performed is shown for four classes of schedulers. The scheduling activities are listed on top and the strategies on the left.

## 35.2.1.

data), and VLIW (very large instruction word) computations [Fis84] are fully statically scheduled. Similarly, wavefront arrays [Kun88] and asynchronous circuits use self-timed scheduling. In general purpose parallel processors, when there is no hardware support for scheduling (except synchronization primitives), self-timed scheduling is usually used. Hence, most applications of today's general purpose multiprocessor systems use some form of self-timed scheduling, using for example CSP principles [Hoa78] for synchronization. In these cases, it is often up to the programmer, with meager help from a compiler, to perform the scheduling.

Examples of static-assignment scheduling include many dataflow machines [Sri86]. Dataflow machines evaluate dataflow graphs at run time, but a commonly adopted practical compromise is to allocate the actors to processors at compile time. To use an example targeted at DSP, the NEC uPD7281 [Cha84] uses static-assignment scheduling based on the tagged-token concept [Arv82]. Another example is TI's data-driven processor (DDP), designed for executing Fortran programs that are translated into dataflow graphs by a compiler [Cor79]. The cost of implementing tagged-token architectures has recently been dramatically reduced using the "explicit token store" concept [Pap88]. Another example of an architecture that assumes static-assignment is the proposed "argument-fetching dataflow architecture" [Gao88], which is based on the argument-fetching data-driven principle of Dennis and Gao [Den88]. A machine that has a mixture of fully dynamic and static assignment scheduling is the Manchester dataflow machine [Wat82]. Here, a number (15) of processing elements are collected in a ring. Actors are assigned to PEs at run time within the ring, but there are many such rings. To communicate between rings, tokens are transmitted over a communication network. Thus, assignment is dynamic within rings, but static across rings. Fully dynamic scheduling has been applied in the MIT static dataflow architecture [Den80], the LAU system, from the Department of Computer Science, ONERA/CERT, France [Pla76], COSSAP [Kun87], and the DDM1 [Dav78].

Although obviously attractive in principle, automatic schedulers today either (1) add unacceptable cost to the implementation by operating near the fully-dynamic end of the spectrum or (2) work well with only a subset of algorithms. The basic premise of this paper, elaborated in the following sections, has three parts.

- (1) In order to reduce implementation costs and make it possible to reliably meet real-time constraints, the more that is done at compile time the better.
- (2) Schedulers that work well with only a subset of applications can be acceptable, but the subset should be large enough that reasonable applications can be implemented in their entirety.
- (3) Automatic scheduling is better than manual.

### 3. DOMAIN AND RANGE

The *domain* of a scheduling strategy can be loosely defined as the set of algorithms for which the scheduling strategy does well. The *range* is the set of architectures that the strategy can target well. Most practical scheduling strategies have a limited domain or range.

Of the classes we have defined, fully static scheduling has the narrowest domain. The subclass of dataflow graphs for which fully static scheduling works best is *synchronous data flow* [Lee87]. SDF graphs consist only of actors where the number of tokens (units of data) produced or consumed when the actor fires is fixed and known at compile time. With the further restriction that execution times of the actors are exactly known, optimal fully static scheduling is possible. Unfortunately, even in this restricted domain, algorithms that accomplish such optimal scheduling have combinatorial complexity, except in certain trivial cases [Cof76][Cap84]. Fortunately, good heuristic methods have been developed over the years, many being critical path methods [Cof76]. The *range* depends on the sophistication of these methods, although most straightforward implementations target homogeneous tightly coupled multiprocessors with full interconnectivity. The

requirement for full interconnectivity limits the range to machines with modest parallelism.

A subclass of fully static scheduling is the set of techniques based on projecting dependence graphs for *regular iterative algorithms* onto systolic arrays [Kun88] [Rao85]. These techniques have a very limited domain (RIA's) and range (systolic arrays), but do extremely well within these constraints.

The domain of static scheduling specifically does not include dataflow graphs with actors that have data-dependent execution times or actors that may or may not fire, depending on the value of some data somewhere in the graph. These restrictions are severe, since they exclude both conditionals and data-dependent iteration within an actor or involving several actors. The restrictions can be relaxed, however, at the expense of optimality in the resulting schedule. For example, an actor with a data-dependent execution time can be padded so that it always executes in worst-case time, an approach that is well suited to real-time computation. As another example, to implement the synchronous dataflow equivalent of if-then-else, both branches of the conditional may be computed, and the desired result may be selected. There are again applications where this option is acceptable, for example when one of the two conditional branches is trivial, but most of the time the cost will be high.

Static scheduling has been extended to handle limited forms of data-dependent firing of actors [Lee88]. For example, consider the graph in figure 2. This is not a synchronous dataflow graph because it is not possible to state at compile time how many tokens the switch and select actors will produce or consume on each input each time they fire. Nonetheless, *quasi-static* scheduling is possible, where a small amount of run-time control is inserted to determine which subgraph,  $f(\cdot)$  or  $g(\cdot)$ , to invoke next. Again, the execution times are padded so that each processor involved in the if-then-else clause finishes at a fixed time, independent of which branch was executed.

Self-timed scheduling has a slightly broader application domain. Although the order of execution of actors is fixed for each processor at compile time, the exact firing times are not. Consequently, the schedule can automatically compensate for certain fluctuations in execution times. For example, if one actor finishes earlier than expected, the following actor can fire immediately, as long as its data is available. Compared to using worst-case execution times, self-timed scheduling will always do at least as well, as long as the overhead for synchronization is negligible. Self-timed scheduling is also more robust, in that minor fluctuations in execution times will not affect the correctness of the execution, and will have little effect on its performance. For example, interrupts are often useful for handling I/O operations, but their introduction introduces uncertainty in the execution of any actor that may be interrupted. The Gabriel system [Lee89] uses self-timed scheduling.

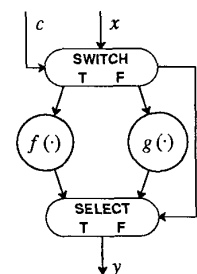


Figure 2. Dataflow representation for an if-then-else clause. A token  $x$  is routed to one of two functions,  $f(\cdot)$  or  $g(\cdot)$ , depending on the value of the condition token  $c$ .

It is clear that self-timed scheduling does well with synchronous dataflow when there is little variability in the execution times of the actors. In the presence of limited amounts of data-dependent firing of actors, such as found in figure 2, it will also do well. We have more recently discovered that it also looks promising for certain types of data-dependent iteration [Ha89]. Loosely, it appears that the domain can be stated simply as the set of algorithms that are "mostly" static. It is not clear just how much dynamic behavior must be present before enough time is spent idling, waiting for data tokens to arrive, that a more dynamic scheduling technique would have been more effective. Nonetheless, this domain seems like a good match to signal processing, for which practical implementations of most well-known algorithms occasionally require data dependencies.

Static assignment scheduling, in principle, has still broader range, because it can adjust the ordering of execution of actors. An example is shown in figure 3. In that example, one of six actors has a data-dependent execution time. Depending on the outcome, it may be better to schedule the actors using the ordering in (b) or in (c). While it is possible to reduce the total execution time by rearranging the order of execution, it is not always easy to determine at run time which actor should be fired next when there is more than one possibility. For example, in figure 3c, after *A* completes on the second processor, either *B* or *E* can be fired. For the implementation to be cost effective, the decision would have to be made on the basis of local or static (compile-time) information.

Fully dynamic scheduling has the broadest application domain, since it can in principle subsume all functions in the previous models, and perform them at run time. Furthermore, it has the flexibility to redirect the computational load in response to changing conditions in the algorithm. Unfortunately, scheduling mechanisms for accomplishing this are not very well understood, except in situation where the scheduling could be equally well done at compile time (section 5). Furthermore, the run time cost of fully dynamic scheduling is too high (section 4).

#### 4. OVERHEAD

Real-time DSP applications typically involve vast amounts of computation at run time, so considerable effort at compile time can be justified. Often a system is designed to perform only a single function in its lifetime. Furthermore, run-time overhead in the form of extra hardware is not acceptable for cost-sensitive applications, and run-time overhead in the form of extra software only makes it more difficult to meet real-time constraints.

Fully dynamic scheduling involves considerable run-time overhead, even when simplistic scheduling strategies such as randomized scheduling are used. It is difficult to exploit communication locality in a dataflow graph, since all tokens traverse the communication network. In one of the better known realizations, the MIT static dataflow machine [Den80], Dennis proposes using a high-speed packet switch for instruction delivery. A useful multiprocessor for DSP must have a very high instruction delivery rate, so this packet switch will have to have high bandwidth. Furthermore, the packet switch must perform

scheduling functions and buffer instructions when they cannot be executed immediately. The cost of such a packet switch would have to be very carefully justified, and probably can only be justified when more static scheduling strategies break down due to excessive data-dependency in the algorithms. Most signal processing applications and scientific computation are unlikely to fall in this category.

Static-assignment scheduling is more commonly used in dataflow machines because of the high cost of shipping code around the machine. The run-time overhead reduces to that of scheduling actors within one processor. This involves determining when actors can be fired and arbitrating between actors that simultaneously become ready to fire. Some implementations are based on tagged-token principles [Arv82], for example the NEC uPD7281 [Cha84], the cost of which has recently been reduced through the token store (ETS) scheme [Pap88].

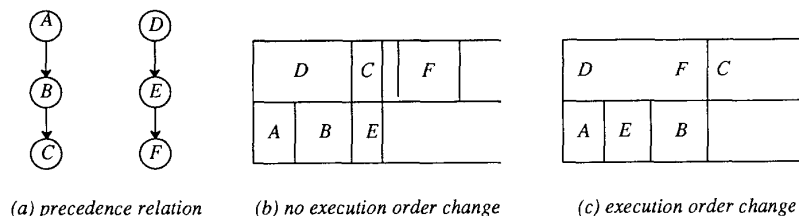
In self-timed scheduling, there is no need to determine at run time which actor to invoke next. The ordering is specified at compile time. The only run-time scheduling function is to determine whether the actor that goes next is ready to be fired by checking its input data. If it is not, then the machine is idled until it is. It is up to the compile-time scheduler to find the ordering that minimizes the idling time. As shown in figure 3, restricting the firing order can result in poor schedules when execution time of actors varies.

Fully static scheduling has the lowest run-time cost: no hardware and no software. Since behavior is completely known at compile time, there is no need to check to see when actors can be fired. The compiler can figure it out, so actors simply fire at the designated time, and are assured that their data is available. The main drawback of this technique is its very narrow domain, since it is completely intolerant of random behavior.

#### 5. AN OVERVIEW OF SCHEDULING STRATEGIES

The goal of scheduling is find an assignment, ordering, and timing for actors that optimizes some performance measure. Typical performance measures are minimum mean flow time, makespan, or iteration period. Mean flow time is the average of the firing times of all actors; minimizing this measure might be appropriate when actors involve interaction with human operators. The makespan is the maximum completion time for all actors; minimizing this measure is often appropriate for finite programs. The iteration period is the time to execute one cycle of a periodic program; this performance measure is most appropriate for real-time DSP. It is common to compromise on the performance measure. For example, instead of minimizing the iteration period, a scheduler might construct a blocked schedule, in which each cycle of a periodic computation must finish on all processors before the next cycle can begin on any. With this constraint, minimizing the iteration period is the same as minimizing the makespan of one cycle. We assume non-preemptive scheduling.

For any finite program with data-independent behavior, the solution space of the scheduling problem is enumerable, so optimal schedules can be found in principle. Unfortunately, most deterministic



**Figure 3.** Two static-assignment schedules for two processors are shown for the precedence graph in (a). The execution time of actor *D* is data-dependent and is longer for the schedule in (b) than for the schedule in (c). Note that the ordering of the firing of actors is determined at run time and is different in (b) and (c).

scheduling problems are NP-hard [Cof76], so optimal solutions can only be found for small problems. A more practical approach is to use heuristic algorithms, based for example on the critical path method [Cof76]. We have applied one such method, Hu-level scheduling [Cof76], to the programming of systems with multiple programmable DSPs [Lee89]. Unfortunately, even for reasonably static DSP applications, the restriction to data-independent behavior is too severe.

When data-dependent behavior is included, for example conditionals and iteration, then the optimal scheduling problem becomes hopeless, except in degenerate cases. In principle, stochastic modeling of the program could be used to make compile-time decisions, but only the most grossly oversimplified stochastic models yield to optimization. Equally lacking are policies that can be applied at run-time to make optimal decisions.

One heuristic technique is to use the *expected* execution time of data-dependent subgraphs to guide the run-time scheduling. Based on the work of Martin and Estrin [Mar67], Granski, et. al. describe an experiment in which a fully dynamic scheduler is guided by priorities computed at compile time [Gra87]. The target machine is the MIT static dataflow machine [Den80]. A modified Hu-level algorithm (using approximate expected execution times) determines the priorities. Since the priorities indicate the preferred ordering of actor firing, this technique could almost be viewed as a fifth category in figure 1, where assignment and timing are done at run-time, but ordering is done at compile-time. However, the priorities are not strictly enforced, so this technique remains a fully dynamic scheduling technique. Unfortunately, practical implementation of this method increases the hardware cost of the dataflow machine, and the authors conclude that the marginal performance improvement over random scheduling does not justify the increased cost. Furthermore, they conclude that it is the presence of data-dependent behavior that precludes better performance. Of course, without the data-dependent behavior, we would be in the domain of fully static or self-timed scheduling, and there would be little reason to use a machine that relies on a fully dynamic scheduler.

Fully dynamic schedulers often take the simpler approach of randomly assign actors that are ready to fire to processors as the processors become idle. This scheme, which can make no pretence to optimality, removes the burden of run-time decision making, but still leaves considerable run-time cost. For example in the packet-switched network for instruction delivery proposed by Dennis for the static dataflow architecture [Den80], the packet processing is simplified by random scheduling, but the network is still costly. Finally, the performance of this architecture will depend somewhat on the assignment of actors to memory, since multiple parallel memory units are used to get the requisite bandwidth. The compiler should strive to minimize conflicts, but we have seen no discussion of this problem. Is random assignment adequate?

Static-assignment schedulers have an easier time at run-time because there is no need to determine how to assign actors to processors. Control becomes localized, because each processor only has to worry about the actors that have been assigned to it. A simple "greedy" scheduling algorithm simply fires an actor immediately when the processor becomes free, assuming there is an actor ready to be fired. This is not optimal because it is sometimes better to leave the processor idle until another actor is ready to fire. However, this compromise is common, even in fully static schedulers. When more than one actor is ready to be fired, the scheduler must determine which one to fire. A typical approach that is far from optimal is to randomly order the list of actors and apply a "fairness" principle, in which no actor will be tried twice before all other actors have been tried [Gao83]. Another alternative would be to use compile-time analysis of the dataflow graph to assign priorities to the actors. It seems to us that this idea should work much better in this static-assignment context than in fully dynamic scheduling as investigated by Granski, et. al. [Gra87], because the run-time implementation cost would be trivial.

The question remains how to accomplish the assignment in

static-assignment scheduling. One simple approach is random assignment. This tends to balance the load. Another approach is clustering, where actors which communicate a great deal are grouped and mapped onto one processor in order to reduce interprocessor communication. Numerous authors have proposed techniques that compromise between interprocessor communication cost and load balance [Muh87][Chu80][Zis87][Ma82][Efe82][Lu86]. But none of these consider precedence relations between actors. To compensate for ignoring the precedence relations, some researchers propose using a dynamic load balancing scheme in which processors exchange actors dynamically as they monitor their computational loads at runtime [Kel84][Bur81][Iqb86]. Unfortunately, the cost can be high, and may be difficult to justify for many applications. In particular, for applications with relatively little data dependency, for instance DSP or scientific computation, the compiler can do a much better job of assignment in the first place if it considers the precedences, so dynamic load balancing becomes unnecessary.

We make the following observation; if the dataflow graph has no data dependencies, then we can do a much better job of assignment (compared to random assignment) by constructing a static schedule and discarding the ordering and timing information, retaining only the assignment. Heuristic techniques can be incorporated to simultaneously perform clustering and optimization of the performance measure, as done for example in [Kim88]. If small data-dependencies are introduced, the effectiveness of this approach will hardly be affected. To be able to construct a static schedule, we could simply use expected Hu-levels, as done in [Gra87]. If larger amounts of data dependency are introduced, at some point, this approach will break down, and will not perform significantly better than random assignment. However, the data dependencies can be taken into account more directly in the static scheduling, using for example the quasi-static scheduling strategies in [Lee88]. Such a scheme is proposed by [Ha89], who also extends the quasi-static scheduling strategies in [Lee88] to include data-dependent iteration. The details of these proposals are beyond the scope of this paper, but we can state their conclusion: For dataflow graphs with some data-dependent behavior, reasonable static schedules can be constructed. The quality of the resulting static schedules degrades as the amount of data-dependency increases, but the *assignment* part of the schedules may still be useful. This assignment may yield significantly better performance than random assignment.

A reasonable self-timed scheduling strategy follows the same principle; construct a fully static schedule using the best available information about execution times, etc., and discard the timing information, retaining both the assignment and the ordering. This is the approach taken in Gabriel [Lee89] as well as other related systems [Zis87].

Fully static scheduling can now be seen to be a useful technique even if it only solves part of the problem for self-timed and static assignment scheduling.

## 6. CONCLUSION

In order to reduce implementation costs and make it possible to reliably meet real-time constraints, the more scheduling that is done at compile time the better. Unfortunately, in order to automatically do more at compile time, it appears to be necessary to restrict the system to a narrower range of applications. However, real-time DSP is a range of applications that appears to be a good match with the self-timed scheduling strategy. In this strategy, the assignment of actors to processors, and the ordering of the firing of actors, is determined by a compiler. Only the timing of the firing is determined at run-time. The run-time cost of this determination is minimal. Furthermore, automatic scheduling techniques that fit this model are growing in generality and efficiency. We conclude research in multiprocessor real-time DSP should focus on solving the remaining compile-time scheduling problems for the self-timed scheduling strategy.

## 35.2.4.

## REFERENCES

- [Ada74] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems" *Comm. ACM*, **17** (12) pp. 685-690, Dec., 1974.
- [Arv82] Arvind and K. P. Gostelow, "The U-Interpreter", *Computer* **15**(2), February 1982.
- [Bur81] F. W. Burton and M. R. Sleep, "Executing Functional Programs on a Virtual Tree of Processors", *Proc. ACM Conf. Functional Programming Lang. Comput. Arch.*, pp. 187-194, 1981.
- [Cap84] P. R. Cappello and K. Steiglitz, "Some Complexity Issues in Digital Signal Processing" *IEEE Trans. on ASSP* **32** (5), October 1984.
- [Cha84] M. Chase, "A Pipelined Data Flow Architecture for Signal Processing: the NEC uPD7281" *VLSI Signal Processing*, IEEE Press, New York (1984)
- [Chu80] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing", *IEEE Computer*, pp. 57-69, November, 1980.
- [Cof76] E. G. Coffman, Jr., *Computer and Job Scheduling Theory* Wiley, New York (1976)
- [Cor79] M. Cornish, D. W. Hogan, and J. C. Jensen, "The Texas Instruments Distributed Data Processor", *Proc. Louisiana Computer Exposition*, Lafayette, La., March 1979, pp. 189-193.
- [Dav78] A. L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine", *Proc. Fifth Ann. Symp. Computer Architecture*, April, 1978, pp. 210-215.
- [Den80] J. B. Dennis, "Data Flow Supercomputers" *Computer*, **13** (11), November 1980.
- [Den88] J. B. Dennis and G. R. Gao "An Efficient Pipelined Dataflow Processor Architecture" To appear in *Proceedings of the IEEE*, also in the *Proc. ACM SIGARCH Conf. on Supercomputing*, Florida, Nov., 1988.
- [Efe82] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems", *IEEE Computer*, pp. 50-56, June, 1982.
- [Fis84] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", *Computer*, July, 1984, 17(7).
- [Gao83] A Pipelined Code Mapping Scheme for Static Dataflow Computers, PhD dissertation, Laboratory for Computer Science, MIT, Cambridge, MA (1983).
- [Gao88] G. R. Gao, R. Tio, and H. H. J. Hum, "Design of an Efficient Dataflow Architecture without Data Flow", *Proc. Int. Conf. on Fifth Generation Computer Systems*, 1988.
- [Gon77] M. J. Gonzalez, *Deterministic Processor Scheduling*, *Computing Surveys*, **9**(3), September, 1977.
- [Gra87] M. Granski, I. Korn, and G. M. Silberman, "The Effect of Operation Scheduling on the Performance of a Data Flow Computer", *IEEE Trans. on Computers*, **C-36** (9), September, 1987.
- [Ha89] S. Ha and E. A. Lee, "Compile-time Scheduling of Data-Dependent Iteration", paper in preparation.
- [Hoa78] C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, August 1978, **21**(8)
- [Iqb86] M. A. Iqbal, J. H. Saltz, and S. H. Bokhari, "A Comparative Analysis of Static and Dynamic Load Balancing Strategies", *Int. Conf. on Parallel Processing*, pp. 1040-1045, 1986.
- [Kel84] R. M. Keller, F. C. H. Lin, and J. Tanaka, "Rediflow Multiprocessing", *Proc. IEEE COMPCON*, pp. 410-417, February, 1984.
- [Kim88] S. J. Kim and J. C. Browne, "A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures", *Proc. Int. Conf. on Distributed Computing Systems*, 1988.
- [Kun87] J. Kunkel, "Parallelism in COSSAP", *Internal Memorandum*, Aachen University of Technology, Fed. Rep. of Germany, 1987.
- [Kun88] S. Y. Kung, *VLSI Array Processors* Prentice-Hall, Englewood Cliffs, NJ (1988).
- [Lee87] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Trans. on Computers*, January 1987, **C-36**(2)
- [Lee88] E. A. Lee, "Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages", in *VLSI Signal Processing III*, Ed. R. W. Brodersen and H. S. Moscovitz, IEEE Press, New York, 1988.
- [Lee89] E. A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP" *IEEE Trans. on ASSP*, To Appear (1989).
- [Lu86] H. Lu and M. J. Carey, "Load-Balanced Task Allocation in Locally Distributed Computer Systems", *Int. Conf. on Parallel Processing*, pp. 1037-1039, 1986.
- [Ma82] P. R. Ma, E. Y. S. Lee and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems", *IEEE Trans. on Computers*, Vol. **C-31**, No. **1**, pp. 41-47, January, 1982.
- [Mar67] D. F. Martin and G. Estrin, "Models of Computations and Systems — Cyclic to Acyclic Graph Transformations" *IEEE Trans. on Electron. Comput.*, **EC-16** pp. 70-79, February 1967.
- [Muh87] H. Muhlenbeim, M. Gorges-Schleuter, and O. Kramer, "New Solutions to the Mapping Problem of Parallel Systems : The Evolution Approach", *Parallel Computing*, **4**, pp. 269-279, 1987.
- [Pap88] G. M. Papadopoulos, *Implementation of a General Purpose Dataflow Multiprocessor*, Dept. of Electrical Engineering and Computer Science, MIT, PhD Thesis, August, 1988.
- [Pla76] A. Plas, et. al., "LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment", *Proc. 1976 Int. Conf. Parallel Processing*, pp. 293-302.
- [Rao85] S. K. Rao *Regular Iterative Algorithms and their Implementations on Processor Arrays*, Information Systems Laboratory, Stanford University, October, 1985, PhD Dissertation.
- [Sri86] V. P. Srin, "An Architectural Comparison of Dataflow Systems", *Computer*, **19**(3) March 1986.
- [Wat82] I. Watson and J. Gurd, "A Practical Data Flow Computer", *Computer* **15** (2), February 1982.
- [Zis87] M. A. Zissman and G. C. O'Leary, "A Block Diagram Compiler for a Digital Signal Processing MIMD Computer", *IEEE Int. Conf. on ASSP*, pp. 1867-1870, 1987.