ARCHITECTURES FOR STATICALLY SCHEDULED DATAFLOW

by

Edward Ashford Lee

Memorandum No. UCB/ERL M89/129

7 December 1989

ARCHITECTURES FOR STATICALLY SCHEDULED DATAFLOW

by

Edward Ashford Lee

Memorandum No. UCB/ERL M89/129

7 December 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ARCHITECTURES FOR STATICALLY SCHEDULED DATAFLOW

by

Edward Ashford Lee

Memorandum No. UCB/ERL M89/129

7 December 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

1. INTRODUCTION

Digital signal processing (DSP) applications differ from general purpose computation both in the nature of the algorithms and in the target hardware. The algorithms tend to have less decision making and use mostly simple data structures (arrays and streams). The target hardware is often dedicated to an application or a small class of applications, rather than being general purpose, and often has to have low cost together with very high computation rates (to meet hard real-time constraints). These differences create both a hindrance and an opportunity. The hindrance is that mainstream computer science techniques do not apply very well to DSP. This accounts for the fact that the DSP community designs its own microprocessors, computer languages, multiprocessor architectures, and software. The opportunity is that the structural simplicity of the algorithms and data structures makes some traditionally very difficult problems much easier.

Dataflow techniques have been applied to DSP in the guise of "block-diagram languages" since its very earliest days. Whereas most of the computer user community resists the introduction of new programming paradigms, the DSP community has embraced experimentation of this type. Dataflow representation of algorithms, in fact, is very natural in DSP, appealing *even without the motivation of concurrency.* Of course, automatically exploiting concurrency can only increase the appeal. However, most attempts to do so through the use of dataflow architectures have not succeeded commercially. I propose in this paper that the principal reason is that the dataflow techniques of general purpose computing are too expensive for DSP and more powerful than what is required. The focus of this discussion is on scheduling, the heart of concurrency in dataflow.

In the process of developing a general scheduling strategy suitable for DSP, we have to be realistic in our assertions about the algorithms; specifically, almost any generalization has counterexamples. Although we can rely on *relatively little* decision making in the algorithms,

we cannot rely on *no* decision making without sacrificing a great number of applications. Consequently, the proposed scheduling strategies tolerate decision making, but the performance may degrade as the amount of decision making increases. This tolerance, however, means that elements of the strategy may be applicable in general purpose computing. For example, we will suggest that when only the assignment of actors to processors is done at compile time, it could be done by constructing a static schedule, and discarding all information in the schedule except the assignment.

1.1. A Scheduling Taxonomy

In this paper, we only consider non-preemptive scheduling, and the emphasis will be on practical solutions rather than unrealistic abstract models. For the purposes of this paper, I will define "scheduling" to include three tasks: (1) assigning actors to processors, (2) ordering the actors on each processor, and (3) specifying their firing time. *Every* dataflow implementation must perform all three tasks, but implementations can differ by performing them at compile time or at run-time, or by using complex or simple scheduling strategies. Depending on which tasks are done when, we define four classes of scheduling. The first 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. 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. In the third type of scheduling, the compiler determines the order in which actors fire on each processor. At run-time, each 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 most taxonomies, the boundary between these categories is not rigid.

1.2. Examples

We can give familiar examples of each of the four strategies applied in practice. 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], and the DDM1 [Dav78]. It has also been applied in a digital signal processing context for coding vector processors, where the parallelism is of a fundamentally different nature than that in dataflow machines [Kun87]. A machine that has a mixture of fully-dynamic and static-assignment scheduling is the Manchester dataflow machine [Wat82]. Here, 15 processing elements are collected in a ring. Actors are assigned to a ring at compile time, but to a PE within the ring at run time. Thus, assignment is dynamic within rings, but static across rings.

Examples of static-assignment scheduling include many dataflow machines. It is a commonly adopted practical compromise in these machines is to allocate the actors to processors at compile time. Many implementations are based on the tagged-token concept [Arv82]; for example TI's data-driven processor (DDP) executes Fortran programs that are translated into dataflow graphs by a compiler [Cor79] using static-assignment. Another example (targeted at digital signal processing) is the NEC uPD7281 [Cha84]. The cost of implementing tagged-token architectures has recently been reduced significantly using an "explicit token store" [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].

When there is no hardware support for scheduling (except perhaps synchronization primitives), then 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. A more automated class

of self-timed schedulers targets wavefront arrays [Kun88]. Taking a broad view of the meaning of parallel computation, asynchronous digital circuits can also be said to use self-timed scheduling.

Systolic arrays, SIMD (single instruction, multiple data), and VLIW (very large instruction word) computations [Fis84] are fully-statically scheduled. Again taking a broad view of the meaning of parallel computation, synchronous digital circuits can also be said to be fully-statically scheduled.

1.3. Generality

As we move from strategy number one to strategy number four, the compiler requires increasing information about the actors in order to construct good schedules. However, assuming that information is available, the ability to construct deterministically optimal schedules increases. To construct an optimal fully-static schedule, the execution time of each actor has to be known; This requires that a program have only deterministic and data-independent behavior. Constructs such as conditionals, data-dependent iteration, and some recursion make this impossible and realistic I/O behavior makes it impractical.

Self-timed scheduling in its pure form is effective for only the subclass of applications where there is no data-dependent firing of actors, and the execution times of actors do not vary greatly. However, unlike fully-static scheduling, some variation is tolerable. Signal processing algorithms and scientific computation often fit this model. The run-time overhead is very low, consisting only of simple handshaking mechanisms, and requiring no sophisticated hardware capability such as indivisible "fetch-and-add" or "fetch-and-set" primitives. Furthermore, provably optimal (or close to optimal) schedules are viable. As with fully-static scheduling, conditionals, data-dependent iteration, and recursion are excluded if the resulting schedule is to be optimal.

Static-assignment scheduling is a compromise that admits more data dependencies than either fully-static or self-timed, but all hope of optimality must be abandoned in most cases. Although static-assignment scheduling is commonly used, compiler strategies for accomplishing the assignment are not satisfactory. 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 a dynamic load balancing scheme at run-time [Kel84][Bur81][Iqb86]. Unfortunately, the cost can be nearly as high as fully-dynamic scheduling. Others have attempted with limited success to incorporate precedence information in heuristic scheduling strategies. For instance, Chu and Lan use very simple stochastic computation models to derive some principles that can guide heuristic assignment for more general computations [Chu87]. However, only very simple stochastic models yield to analysis, so we should not expect too much from the resulting principles.

Fully-dynamic scheduling is most able to utilize the resources and to fully exploit the concurrency of a dataflow representation of an algorithm, regardless of the amount of data dependencies. However it requires too much hardware and/or software run-time overhead. For instance, the MIT static dataflow machine [Den80] proposes an expensive broadband packet switch for instruction delivery and scheduling. Furthermore, it is not usually practical to make globally optimal scheduling decisions at run-time, so practical implementations fall short of the theoretical ability to exploit parallelism. One attempt to overcome this by using compile-time information to assign priorities to actors to assist a dynamic scheduler was rejected by Granski et. al., who conclude that there is usually not enough performance improvement to justify the cost of the technique [Gra87]. However, in the special case of algorithms with "regular, static structure" (such as a DFT), there are significant performance

improvements. But it is precisely such algorithms that require *only* static scheduling, and can therefore be efficiently implemented on much less expensive machines that include no run-time mechanism for scheduling actors. A cost effective solution for a general-purpose computer might be an architecture that can revert to imperative control flow (or something resembling it) when executing algorithms that are statically scheduled. Perhaps some of the recently proposed hybrid von Neumann/dataflow architectures could take advantage of this observation (see for example [Nik89][Ian88]).

In view of the high cost of fully-dynamic scheduling, static-assignment and self-timed are attractive alternatives. This is true even though both will suffer in performance, compared to fully-dynamic scheduling, as the amount of data dependency increases. Self-timed is more attractive for scientific computation and digital signal processing, because it is more static, while static-assignment may be more attractive when there is more data dependency. The performance of both techniques depends heavily on good compile-time decisions, so it is appropriate to concentrate on finding good compiler algorithms.

1.4. Strategy

For any scheduling strategy that requires compile-time decisions, for example assignment or ordering, these decisions can be made by constructing a fully-static schedule and discarding the information that is not required. At run time, the execution is forced to exactly match the "retained" information. For example, in static-assignment, only the assignment information is retained. In self-timed, the assignment and ordering information is retained. In fully-static scheduling, all information is retained.

We will explore a model lying between fully-static and self-timed, and develop an approach to architecture design well matched to this model. In this new model, we retain not only the ordering of actors on each processor, but also the ordering of accesses to shared

resources, such as shared memory or shared data structures. Architectures supporting this model are only slightly more complex than architectures supporting fully-static scheduling, but the model is much more robust. Specifically, some of the flexibility of self-timed scheduling persists, because timing information in the fully-static schedule is discarded. Hence the execution time of actors can vary at run time without affecting the correctness of the execution. On the other hand, the run-time execution is more constrained than for self-timed, because the order in which processors access shared resources is forced at run time to exactly match that of the fully-static schedule.

2. RUN-TIME ENVIRONMENT

For fully-static implementations the target architecture need not have any special hardware for run-time scheduling. For self-timed scheduling, the only additional requirement is efficient handshaking. In both cases, Von Neumann processing elements are adequate; there is no need to resort to dataflow machines. This section explores these advantages by discussing the run-time cost of two architectures, one designed for self-timed scheduling, and one designed for a new model lying between self-timed and fully-static scheduling.

We assume a host carries out the compilation, mapping an application program onto parallel processors that run under control of the host. The parallel processors are designed for signal processing and scientific computing, so we will not be so ambitious as to try to map the compiler and operating system onto the same set of parallel processors by the same techniques. However, with the dropping cost of hardware, a heterogeneous multiprocessor system of this type is attractive. Different parts of the system are specialized to different functions, and hence can do a much better job than a completely "general purpose" solution.

2.1. Architectures for Static Scheduling

At Berkeley, we have implemented a limited dataflow programming environment (Gabriel) for digital signal processing that targets multiprocessor machines made with programmable DSP microprocessors [Lee89b]. In this case, the compiler and scheduler produce assembly code for each Von Neumann processor in the system. Self-timed scheduling is used, so there are no dataflow principles invoked at run-time, except that semaphore-based synchronization is used when tokens pass between processors.

One of the target architectures in our lab, donated by Dolby Labs of San Francisco, has four Motorola DSP56000's, each with a private memory, plus a single shared memory. Accessing the shared memory requires first requesting the bus, then reading the memory, checking a semaphore, and resetting the semaphore. An ideal transaction is illustrated in figure 1 (ideal means that the bus is free when requested, and the semaphores are in their proper state when checked). It is not necessary to have indivisible test-and-set primitives

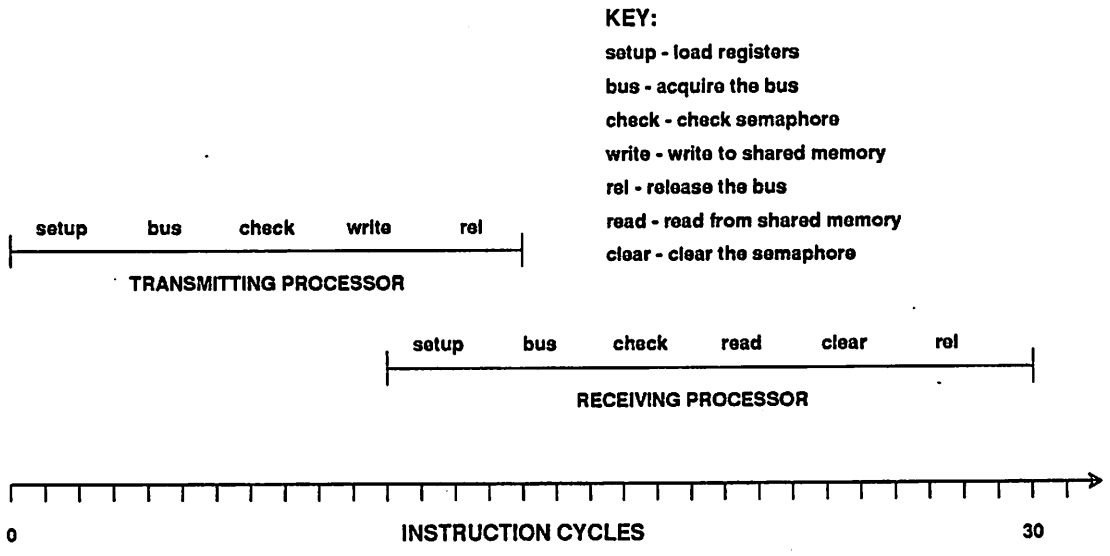

Figure 1. An ideal transaction through shared memory in a typical shared memory multiprocessor when scheduling is self-timed. The number of instruction cycles shown are measured on a prototype multiprocessor system in our Lab.

because the static buffering strategy used in Gabriel ensures that no more than one processor will be writing data to any given memory location, and no more than one processor will be reading data from that location [Lee87b]. The bus contention/resolution and semaphore handling are the only scheduling overhead incurred at runtime. Nonetheless, in the Dolby architecture these require about 30 instruction cycles for a single transaction, as shown in figure 1. This relatively high cost implies that only large-grain dataflow can be supported efficiently. Furthermore, each token should ideally contain more than a single data value, because the overhead is incurred only once for each token. We deem these restrictions excessive, and attribute them to the fact that the architecture was not designed with self-timed scheduling of dataflow graphs in mind. It is much more general than we need.

In the Dolby architecture, if the bus is not available when requested, the requesting processor halts until the bus becomes available. Hence contention for the bus can extend the duration of a transaction well beyond the 30 cycles shown in figure 1. In our software implementation of semaphore handling, if the semaphore read from shared memory is not in the desired state, then the processor releases the bus, and attempts the transaction again some time later. The processor busy-waits in the meantime. It is up to the scheduler to ensure that processors do not spend much time busy-waiting. Of course, these repeated reads further increase the load on the shared bus. We conclude that some modification of the architecture is required to be able to effectively map self-timed dataflow graphs onto it.

2.1.1. A Gated-Shared-Memory Architecture

One way to reduce the total transaction time illustrated in figure 1 would be to add hardware that performs the functions we now perform in software, such as semaphore management. An architecture doing this might look like that in figure 2. Shared memory accesses begin by supplying an address in the shared memory space to the *gate keeper*. The gate keeper asserts a wait signal until the memory request can be satisfied, causing the

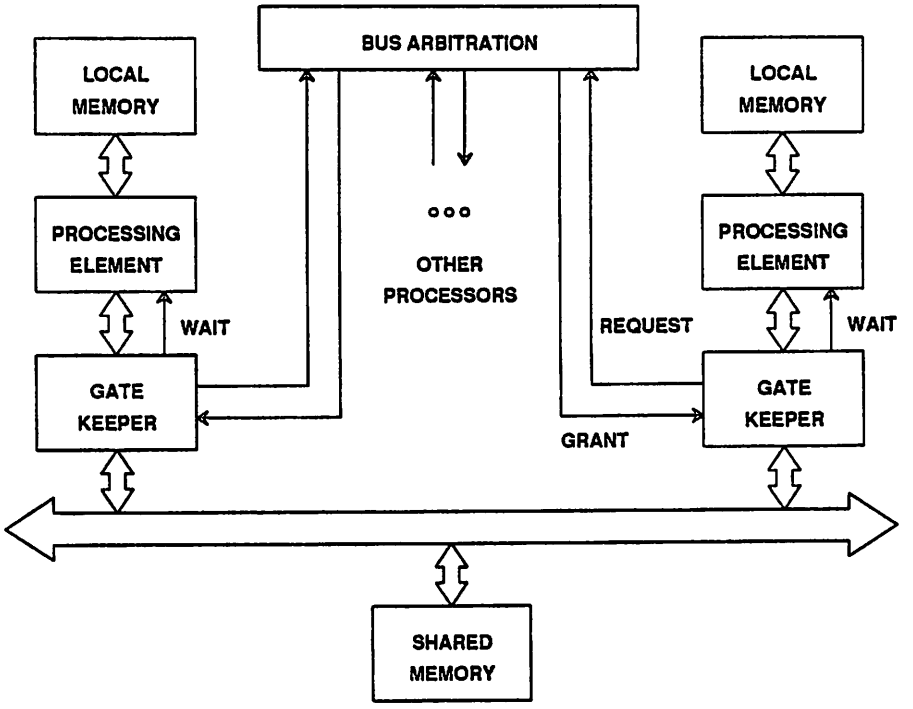

Figure 2. A gated-shared-memory architecture.

processor to halt. Meanwhile, the gate keeper acquires the bus, accesses the shared memory, and checks the semaphore. Just as in the software implementation, if the semaphore is not in the desired state, then the read is repeated some time later, with the processor held in its suspended state in the meantime. As before, it is up to the scheduler to ensure that not much time is wasted this way. Since the schedule is self-timed, there is no danger of introducing deadlock by this mechanism.

On writes to shared memory, the gate-keeper need not halt the processor requesting the write. The processor can proceed with its execution until the next shared-memory transaction is encountered. In the meantime, the gate keeper performs the shared-memory write in parallel.

The main advantage of this architecture is that the functions we now perform in software are performed in hardware, and therefore presumably occur much faster. Furthermore, the

processing elements can access shared-memory locations the same they access local memory. The contention/resolution and synchronization functions are transparent. However, the gate keeper is not trivial, so the time required for shared-memory accesses is still likely to be larger than the time for local memory accesses, even when there is no contention and the semaphores are in the desired state. In addition, a preliminary design indicates that to implement it on a single chip requires a large (albeit manageable) number of pins. This preliminary design assumed the processing elements would be Motorola DSP96002's, which have separate 32-bit address and data busses (two of each). Consequently, it is worth considering an alternative that brings us closer to fully-static scheduling.

2.1.2. An Ordered-Shared-Memory Architecture

Consider an architecture, shown in figure 3, where a shared bus is not requested by the processors, but rather a central controller (labeled MOMA) grants the bus to processors in some prespecified order. Once the bus has been granted to a processor, it is not released until the processor has completed a shared-memory transaction. The key idea is that a fully-static scheduler can determine *a-priori* the order in which shared-memory transactions will occur. Hence, at the same time that a program is loaded into the private memories of each of the processors, a list is loaded into the controller specifying the order of memory transactions. This list is simply a list of processor numbers, and the controller simply asserts the bus-grant line for each processor in turn. The key advantage is that *no explicit hardware or software is required for contention/resolution or semaphore management.* Contention is avoided by granting the bus to only one processor at a time. If any processor reaches a code segment where it tries to access shared memory but has not been granted the bus, it simply halts until the bus is granted. Static scheduling ensures that this will not cause deadlock. Furthermore, semaphore synchronization is no longer required. To see this, suppose processor 1 wishes to read a location written by processor 2. Static ordering of memory accesses ensures that the

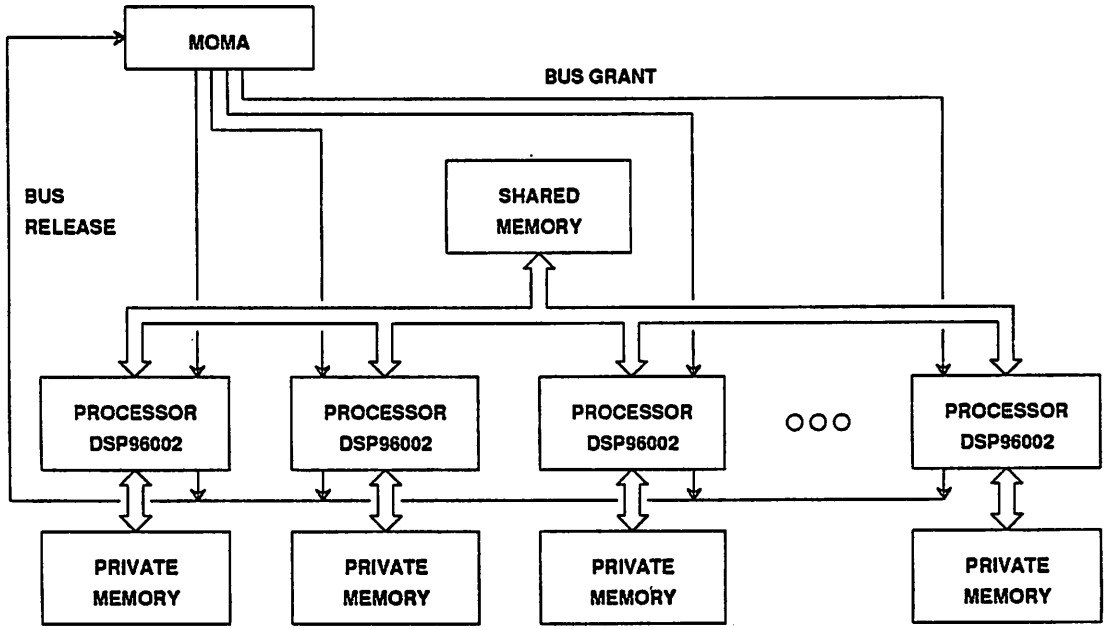

Figure 3. Shared memory accesses can be made extremely efficient when scheduling is static. Here, a controller (MOMA: Maintains Ordered Memory Accesses), grants access to the shared memory in the order predicted by the scheduler. No software or hardware overhead is required for contention/resolution or semaphore handling.

read and write occur in the proper order. Furthermore, if the static scheduler has done a good job, very high bandwidth to shared memory is achievable. The bus is held a minimum amount of time for each transaction. A good hardware design and a good schedule could achieve at least 15 times the shared-memory bandwidth of the implementation in figure 1.

In general, centralized controllers in multiprocessor systems are not a good idea because they limit scalability. However, we believe that the architecture in figure 3 can be used to find the true limits of scalability of shared-memory architectures. The controller will not be the bottleneck. Unlike many centralized controllers, this one is simple, and can probably be implemented easily on a single semi-custom chip. The bus grant lines require one pin per processor, so pin count is not a serious problem until hundreds of processors are used together (unlikely even in an efficient shared-memory system). Even if this proves feasible, the bus grant signal could be encoded, and a small number of decoder chips would have to be added.

The bus release is accomplished with a single wired-or release signal, so only one pin on the controller is required for this function.

Interesting enhancements are immediately evident. for example, the static scheduler not only knows which processor will be accessing shared memory next, but it knows which memory location will be accessed, and whether the location will be read or written. This knowledge can be used to speed shared-memory transactions for a relatively slow memory. As it asserts the bus-grant line for a processor, the controller can also drive the address lines of the shared memory. If the next transaction is a read, and the processor is not already waiting, then the response time of the memory will appear to the processor to be zero! When it gets around to performing the read, the desired data will already be on its data bus.

Another obvious possibility is to apply the same philosophy to architectures with more elaborate (and more scalable) communication networks. The basic idea is that the order in which shared resources are used is determined at compile time and enforced at run time.

The main limitation with the architecture in figure 3 is the requirement that the order of shared memory transactions be known at compile time. This is possible when fully-static scheduling is possible, which, among other limitations, implies that the execution time of every actor must be known. The Gabriel system is self-timed because it is not usually practical to know exactly the execution time of all actors. However, the architecture in figure 3 will function correctly even if *estimates* of the execution time are used. This is because the ordered bus grants provide synchronization, of a sort. If poor estimates of runtimes are used, or a processor is interrupted, then this architecture may yield worse performance than an architecture that permits dynamic re-ordering of shared-memory accesses, like the Dolby architecture. However, the overhead is so much smaller, that intuition indicates that these estimates would have to be poor indeed.

One mechanism that might cause the estimates of actor execution times to be poor is traditional cache management. In many systems, a cache miss can cause a processor to suspend execution of the program for hundreds of cycles. This would probably be unacceptable, since there is a good chance that many of the other processors will be delayed as a consequence. This would not necessarily occur with self-timed scheduling, because shared-memory accesses could be dynamically re-ordered. The same is true of interrupts, and to a lesser degree, data-dependent instruction execution times. Therefore, cache management, interrupt handling, and data-dependent execution-speed optimizations would have to be rethought, or more simply, forbidden. An interesting possible solution is to replace dynamic cache management with static paging. If fully-static scheduling is possible, then in principle, all the requisite information is available. In signal processing, because of hard-real-time constraints, it is common to use software controlled paging rather than dynamic caches [Lee88] [Lee89a].

The scheduling strategy for the architecture in figure 3 lies somewhere between fully static and self-timed. Hence, just like self-timed scheduling, conditionals, data-dependent iteration, and recursion are excluded from the programming model. We will explain these limitations precisely below. They may be acceptable for a subset of signal processing applications, but will not be acceptable in more general applications. A more sophisticated centralized controller, tuned to these programming constructs, may remove this difficulty. Using our new quasi-static scheduling strategies to handle these programming constructs [Ha89], preliminary indication is that the controller can monitor decision paths in the program by monitoring addresses on the shared-memory bus. Consequently, the controller can modify its bus-grant pattern based on decisions made in each of the component processors.

The processors in figure 3 are specified as DSP96002's, the next generation of 32-bit floating-point programmable digital signal processors from Motorola. It is expected that this device will be well suited to general scientific computation, graphics, and signal processing.

More importantly, it is well matched to the proposed architecture because it has two completely independent tri-stated memory busses. It can be wired exactly as shown in figure 3, with no extra logic. The result is a very low cost, very dense multiprocessor system.

There is nothing *architecturally* sophisticated in either figure 3 or figure 2. The sophistication comes with binding the architecture to a software methodology.

2.2. Statically Ordered Data Structure Access

The lack of side-effects in dataflow actors makes it particularly difficult to support large, shared data structures [Gau86]. Arvind, et. al., have therefore extended the dataflow model by introducing *I-structures*, a controlled form of global data structures [Arv87]. I-structures are write-once data structures with non-strict semantics, which in practice means that reads may be issued before data is available in the data structure. Support for I-structures requires an ability to queue read requests until they can be satisfied. This mechanism is the most promising available, but it does not come cheaply. A much simpler mechanism is possible when scheduling is static.

Consider for example an actor that outputs an array. This array might be carried by a single token. Suppose there are two actors that take this array as an argument. A pure dataflow model requires that the array be copied, or at least that an implementation behave as if the array had been copied. Using an I-structure avoids this copying. However, an older technique called *reference counting* [Hud86][Dri86] is more attractive when scheduling is static. In this example, the reference count (RC) associated with the array storage would be initialized to zero. When the array is written by the first function, the reference count is incremented to 2. Each actor taking the array as an argument can only fire if the RC is greater than zero, and when it fires it decrements the RC.

Many variations of this idea immediately come to mind; for example, reference counts could be used for each element of the array, instead of the whole array, thereby getting some of the advantages of the non-strictness of I-structure. Specifically, the array does not have to be completely filled before some of its elements can be read. Also, if the RC is identically one, then an actor using it may also modify it, something not permitted in the write-once I-structures.

The reference count technique has been criticized for a number of reasons [Arv87b], most of which break down when the scheduling is static. For example, for the ordered-shared-memory architecture of figure 3, the overhead of managing the RC is incurred *only at scheduling time*, not at run time. A data structure shared between processors is put in shared memory, and the RC is used at scheduling time only to determine the order of the accesses. Also, on the main disadvantages of I-structures is avoided. The non-strictness of I-structures implies that any number of read requests may be generated for a data structure before the data is available. Queueing these requests may be burdensome. No such queueing is required with RC's and static scheduling.

This static reference count mechanism works well with the ordered-shared-memory architecture. However, if we wish to go to self-timed execution, using for example the gated-shared-memory architecture, then some run-time support for reference counts is required when the data structure is shared across processors. The reference count replaces the simpler full/empty single-bit semaphore that would be required if a token had only one destination. In this case, the RC can be viewed simply as a multi-bit semaphore.

3. SYNCHRONOUS DATAFLOW

The gated-shared-memory architecture and the static reference counts seem to provide a very clean solution to some vexing problems. However, they are only applicable when fully-static or self-timed scheduling is possible. Although this imposes some serious constraints, the constraints may be less serious than it may seem at first.

A subclass of dataflow graphs lacking data dependency is well suited to static scheduling. Precisely, the term "synchronous dataflow" has been coined to describe graphs that have the following property [Lee87a]:

SDF property:

> A *synchronous actor* produces and consumes a fixed number of tokens on each of a fixed number of input and output paths. An *SDF graph* consists only of synchronous actors.

The basic constraint is that the number of tokens produced or consumed cannot depend on the data. An immediate consequence is that SDF graphs cannot have data-dependent firing of actors, as one might find, for example, in an if-then-else construct. In exchange for this limitation, we gain some powerful analytical and practical properties [Lee87a][Lee87b]:

1) For SDF graphs, the number of firings of each actor can be easily determined at compile time. If the program is non-terminating, as for example in real-time DSP, then a periodic schedule is always possible, and the number of firings of actors within each cycle can be determined at compile time. In either case, knowing these numbers makes it possible to construct a deterministic acyclic precedence graph. If the execution time of each actor is deterministic and known, then the acyclic precedence graph can be used to construct optimal or near-optimal schedules.

2) For non-terminating programs, it is important to verify that memory requirements are bounded. This can be done at compile time for SDF graphs.

3) Starvation conditions, in which a program halts due to deadlock, may not be intentional. For any SDF graph, it can be analytically determined whether deadlock conditions exist.

4) If the execution time of each actor is known, then the maximum execution speed of an SDF graph can be determined at compile time. For terminating programs, this means finding the minimum makespan of a schedule. For non-terminating programs, this means finding the minimum period of a periodic schedule.

5) For any non-terminating SDF graph executing according to a periodic schedule, it is possible to buffer data between actors *statically*. Static buffering means loosely that neither FIFO queues nor dynamically allocated memory are required. More specifically, it means that the compiler can statically associate memory locations with actor firings. These memory locations contain the input data and provide a repository for the output data.

These properties are extremely useful for constructing parallelizing compilers, but they only apply to SDF graphs, and optimal schedules can only be constructed when the execution times of the actors are known. For more general dataflow graphs, where there is data-dependent firing, intuition suggests that the graph should be divided into SDF subgraphs which can themselves be scheduled statically. However, it is not obvious how to account for the interaction among such subgraphs. For example, for each SDF subgraph that is scheduled statically, the compiler must decide how many processors to devote the subgraph. We will review some techniques that have been developed.

Optimal compile-time scheduling of precedence graphs derived from SDF graphs is one of the classic NP-complete scheduling problems. Many simple heuristics have been developed over time, with some very effective ones having complexity n^2, where n is the number of actors (see for example [Hu61]). However, even n^2 complexity can bog down a compiler. Our experience [Lee89b] suggests that medium to large-grain dataflow graphs may

be required to avoid excessive compilation delays on today's computers, assuming an entire program is compiled in one step. A much better alternative would be incremental compilation, suggesting a line of inquiry. Fortunately, signal processing programs, particularly for real-time applications, tend to be small, so the problem is not severe in this application domain. The key limitation, therefore, is the inability to express data-dependent computation. This does not, however, imply that there is no program control.

4. STATICALLY SCHEDULED CONTROL

Static scheduling promises low-cost architectures, at the expense of compile-time complexity. For many applications, this is a very attractive tradeoff. However, only some applications can be statically scheduled. The SDF model, which can be statically scheduled, may appear to lack control constructs because it does not permit data-dependent firing of actors. However, this is not entirely true. In this section we explain the control structures that are possible within SDF.

4.1. Recurrences

The dataflow community has recognized the importance of supporting *recursion*, or self-referential function calls. To some extent, this ability has become a litmus test for the utility of a dataflow model. The most common implementation, however, dynamically creates and destroys instances of actors. This is clearly going to be problematic for a static scheduler.

In imperative languages, recursion is used to implement recurrences and iteration, usually in combination. If we avoid the notion of "function calls", at least some recurrences can be simply represented as feedback paths in a dataflow program graph. This section will study the representation of recurrences using feedback. This representation poses no difficulty for static scheduling, although to some it lacks the elegance of recursion. Dataflow models for iteration will be pursued in the next section.

A dataflow graph with a recurrence is represented schematically in figure 4. This graph is assumed to fire repeatedly. Borrowing terminology from the signal processing community, the feedback path has a *delay*, indicated with a diamond, which can be implemented simply as an initial token on the arc. A set of delays in a dataflow graph corresponds to a *marking* in Petri nets [Pet77] or to tag manipulation operators in the U-interpreter [Arv82]. A necessary (but not sufficient) condition for avoiding deadlock is that any directed loop in the graph must have at least one delay.

A *delay* does not correspond to unit time delay, but rather to a single token offset. Such delays are sometimes called *logical delays* or *separators* to distinguish them from time delays [Jag86]. For SDF graphs, a logical delay need not be a run-time operation. Consider for example the feedback arc in figure 4, which has a unit delay. The numbers adjacent to the arcs indicate the number of tokens produced or consumed when the corresponding actor fires. The initial token on the arc means that the corresponding input of actor A has sufficient data, so when a token arrives on its other input, it can fire. The *second* time it fires, it will consume data from the feedback arc that is produced by the *first* firing of actor B. In steady-state, the n^{th} firing of actor B will produce a token that will be consumed by actor A on its $(n + 1)^{th}$ firing; hence the arc has unit token offset. The *value* of the initial token can be set by the programmer, so a delay can be used to initialize a recurrence. When the initial value is other than zero, we will indicate it using the notation D(*value*). Since delays are simply initial

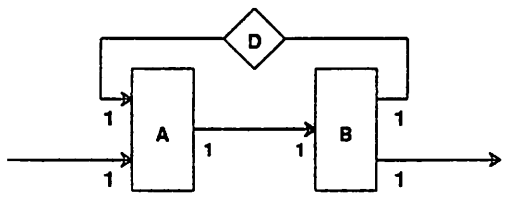

Figure 4. A dataflow graph with a recurrence. Recurrences are expressed using directed loops and delays.

conditions on the buffers, they require no run-time overhead.

Consider non-terminating algorithms, or algorithms that operate on a large data set. For these, directed loops are the only fundamental limitation on the parallelizability of the algorithm. This is intuitive because any algorithm without recurrences can be pipelined. A special case of SDF, called homogeneous SDF, is where every actor produces and consumes a single token on each input and output. For homogeneous SDF graphs, it is easy to compute the minimum period at which an actor can be fired. This is called the *iteration period bound*, and is the reciprocal of the maximum computation rate. Let $R(L)$ be the sum of the execution times of the actors in a directed loop L. The iteration period bound is the maximum over all directed loops L of $R(L)/D(L)$, where $D(L)$ is the number of delays in L [Ren81][Coh85]. The directed loop L that yields this maximum is called the *critical loop*. General SDF graphs can be systematically converted to homogeneous SDF graphs for the purpose of computing the iteration period bound [Lee86]. If there are no directed loops in the graph, then we define the iteration period bound to be zero, since in principle all firings of each node could occur simultaneously. It is important to realize that there is nothing fundamental in the following discussion that prevents this. Implementation considerations may make it impractical, however.

Although in an SDF graph dataflow actors cannot be created at runtime, SDF is not the same as static dataflow [Den80]. For instance, in SDF, there is no impediment to having multiple instances of an actor fire simultaneously. A particular implementation, however, may impose such a constraint. Consider for example an implementation that permits no more than one memory location to be associated with each arc. This can be modeled with the recurrence in figure 4. The feedback arc begins with an initial token. This token represents a "space" on the output buffer of actor A. After A fires, and consumes that token, it cannot fire again until after B has fired. Any memory limitation on any arc in an SDF graph can be modeled as a

feedback path with a fixed number of delays. To avoid unnecessarily sacrificing concurrency, enough memory should be allocated to each arc that the corresponding feedback path does not become the critical loop.

Another impediment to multiple simultaneous invocations of an actor is the notion of state. Particularly in large or medium grain dataflow graphs, it is very convenient to permit an actor to remember data from one invocation to the next. This is simply modeled as a self-loop with a unit delay. Such a self-loop precludes multiple simultaneous invocations of the actor.

4.2. Manifest Iteration

Manifest iteration is where the number of repetitions of a computation is known at compile time, and hence is independent of the data. Manifest iteration can be expressed in data flow graphs by specifying the number of tokens produced and consumed each time an actor fires, and can be statically scheduled. For example, actor B in figure 5 will fire ten times for every firing of actor A. In conventional programming languages, this would be expressed with a *for* loop. Nested *for* loops are easily conceived as shown in figure 5. If actors A and E fire once each, then B and D will fire ten times, and C will fire 100 times. Techniques for automatically constructing static parallel schedules for such graphs are given in [Lee87a].

Although there is no fundamental limitation on the parallelism in figure 5 (there are no directed loops), there may be practical limitations. In figure 6, we model a buffer of length 10 between actors B & C. Again, the tokens on the feedback path represent empty locations in the buffer. Actor B must have ten tokens on the feedback path (i.e. ten empty locations in the

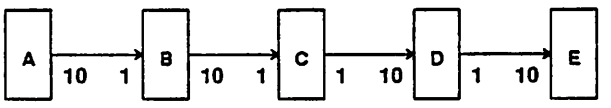

Figure 5. An SDF graph that contains nested iteration.

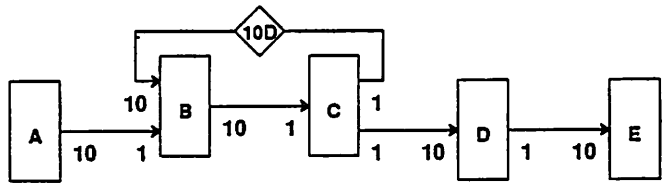

Figure 6. A modification of figure 5 to model the effect of a buffer of length ten between actors B and C.

buffer) before it fires. Whenever actor C fires, it consumes one token from the forward path, freeing a buffer location, and indicating the free buffer location by putting a token on the feedback path. The minimum buffer size that avoids deadlock is ten.

This non-homogeneous SDF graph could be converted to a homogeneous SDF graph and the iteration period bound computed, but in this simple example the iteration period bound is easily seen by inspection. It is clear that after each firing of B, C must fire ten times before B can fire again. The ten firings can occur in parallel, so the minimum period of a periodic schedule is $R_B + R_C$, where R_X is the runtime of actor X. In other words, successive firings of B cannot occur in parallel because of the buffer space limitations. By contrast if the buffer had length 100, then ten invocations of B could fire simultaneously, assuming there are no other practical difficulties.

A second limitation on the parallelism can arise from the addressing mechanism of the buffers. Each buffer can be implemented as a FIFO queue, as done in Davis' DDM [Dav78]. Delays are correctly handled, but then access to the buffer becomes a critical section of the parallel code. FIFO queues are most cheaply implemented as circular buffers with pointers to the read and write locations. However, parallel access to the pointers becomes a problem. If successive invocations of an actor are to fire simultaneously on a several processors, then great care must be taken to ensure the integrity of the pointers. A typical approach would be to lock the pointers while one processor has control of the FIFO queue, but this partially seri-

alizes the implementation. Furthermore, this requires special hardware to implement an indivisible test-and-set operation, assuming the target hardware is a shared memory machine.

A less expensive alternative is static buffering [Lee87b]. Static buffering is based on the observation that there is a periodicity in the buffer access that a compiler can exploit. It preserves the behavior of FIFO queues (namely it correctly handles delays and ordering of tokens), but avoids read and write pointers. Specifically, suppose that all buffers are implemented with fixed-length circular buffers, implementing FIFO queues, where each length has been pre-determined to be long enough to sustain the run without causing a deadlock. Then consider an input of any actor in an SDF graph. Every N firings, where N is to be determined, the actor will get its input token(s) from the same memory location. The compiler can hard-code these memory locations into the implementation, bypassing the need for pointers to the buffer. Systematic methods for doing this, developed in [Lee87b], can be illustrated by example. Consider the graph in figure 6, which is a representation of figure 5 with the buffer between B and C assigned the length 10. A parallel implementation of this can be represented as follows:

```
FIRE A
DO ten times {
        FIRE B
        DO in parallel ten times {
                FIRE C
        }
        FIRE D
}
FIRE E
```

For each parallel firing of C, the compiler supplies a *specific* memory location for it to get its input tokens. Notice that this would not be possible if the FIFO buffer had length 11, for example, because the second time the inner DO loop is executed the memory locations accessed by C would not be the same as the first time. But with a FIFO buffer of length 10, invocations of C need not access the buffer through pointers, so there is no contention for

access to the pointers. The buffer data can be supplied to all ten firings in parallel, assuming the hardware has a mechanism for doing this (such as shared memory).

Using static buffers there is no need for an indivisible test-and-set operation. This is true even if full/empty semaphores are used in the individual buffer locations to synchronize parallel processors. The savings comes from the observation that each shared memory location is written by exactly one processor and read by exactly one processor. In particular, there are no buffer pointers that might be read or written by more than one processor.

An alternative to static buffering that also permits parallel firings of successive instances of the same actor is token matching [Arv82]. However, even the relatively low cost of some implementations of token matching [Pap88] would be hard to justify for SDF graphs, where static buffering can be used.

In figure 5 we use actors that produce more tokens than they consume, or consume more tokens than they produce. Proper design of these actors can lead to iteration constructs semantically similar to those encountered in conventional programming languages. In figure 7 we show three such actors that have proved useful. The first, figure 7a, simply outputs the last of N tokens, where N is a parameter of the actor. The second, figure 7b, takes one input token and repeats it on the output. The third, figure 7c, takes one input token each time it fires, and outputs the last N tokens that arrived. It has a self-loop used to remember the past tokens (and initialize them). This can be viewed as the state of the actor; it effectively

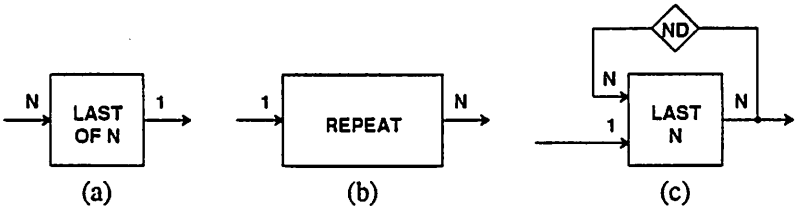

Figure 7. Three SDF actors useful for iteration.

prevents multiple simultaneous invocations of the actor. We will see an example shortly that uses this actor.

A complete iteration model must include the ability to nest recurrences within iteration. We will illustrate this with a finite impulse response (FIR) digital filter because it is a simple example. An FIR filter computes the inner product of a vector of coefficients and a vector with the last N input tokens, where N is the order of the filter. It is usually assumed to repeat forever, firing each time a new input token arrives. Consider the possible implementations using a data flow graph. A large grain approach is to define an actor with the implementation details hidden inside. An alternative is a fine grain implementation with multiple adders and multipliers and a delay line. A third possibility is to use iteration and a single adder and multiplier. This first and last possibilities have the advantage that the complexity of the data flow graph is independent of the order of the filter. A good compiler should be able to do as well with any of the three structures. One implementation of the last possibility is shown in figure 8. The iteration actors are drawn from figure 7. The COEFFICIENTS actor simply outputs a stream of N coefficients; it produces one coefficient each time it fires, and reverts to the beginning of the coefficient list after reaching the end. It could be implemented with a directed loop with N delays, or a number of other ways. The product of the input data and the coefficients is accumulated by the adder with a feedback loop. The output of the filter is

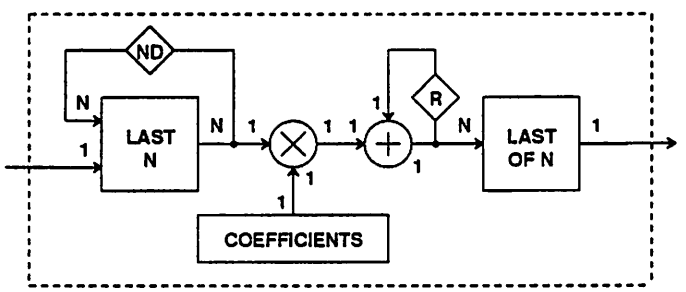

Figure 8. An FIR filter implemented using a single multiplier and adder.

selected by the "last of N" actor.

The FIR filter in figure 8 has the advantage of exploitable concurrency combined with a graph complexity that is independent of the order of the filter. Note, however, that there is a difficulty with the feedback loop at the adder. Recall from above that a delay is simply an initial token on the arc. If this initial token has value zero, then the first output of the FIR filter will be correct. However, after every N firings of the adder, we wish to *reset* the token on that arc to zero. This could be done with some extra actors, but a fundamental difficulty would remain. The presence of that feedback loop implies a limitation on the parallelism of the FIR filter, and that limitation would be an artifact of our implementation. Our solution is to introduce the notion of a *resetting delay*, indicated with a diamond containing an R. The resetting delay is associated with a subgraph, which in this example is surrounded with a dashed line. For each invocation of the subgraph, the delay token is re-initialized to zero. Furthermore, the scheduler knows that the precedence is broken when this occurs, and consequently it can schedule successive FIR output computations simultaneously on separate processors.

The resetting delay can be used in any SDF graph where we have nested iterations where the inner iterations involve recurrences that must be initialized. In other words, anything of the form:

```
DO some number of times {
        Initialize X
        DO some number of times {
                new X = f(X)
        }
}
```

The implementation of a resetting delay is simple and general. For the purposes of implementation, the scheduler first treats the delay as if it were an actor that consumes one token and produces one token each time it fires. Recall that in practice no runtime operation is required to implement a delay, so there actually is no such actor. However, by inserting this

mythical actor, the scheduler can determine how many times it would fire (if it did exist) for each firing of the associated subgraph. The method for doing this is given in [Lee87a], and consists of solving a simple system of equations. For each resetting delay, the scheduler obtains a number N of invocations between resets; this number is used to break the precedence of the arc for every N^{th} token and to insert object code that re-initializes the delay value. The method works even if the subgraph is not invoked as a unit, and even if it is scattered among the available processors. It is particularly simple when in-line code is generated. However, when the iteration is implemented by the compiler using loops, then a small amount of run-time overhead may have to associated with some delays in order to count invocations.

We have given a mechanism for handling manifest iteration in data flow graphs, and for synthesizing efficient parallel implementations. It is worth mentioning that dependence graph methods [Kun88][Rao85] handle manifest iteration using the notion of an *index space* but have the significant disadvantage that all variables used in the algorithm must iterate over the same index space. This restriction is not present in SDF. On the other hand, the functionality of the resetting delay is more cleanly expressed as boundary conditions on an index space.

4.3. Conditional Assignment

Conditionals in dataflow graphs are harder to describe and schedule statically. One attractive solution is a mixed-mode programming environment, where the programmer can use dataflow at the highest level and conventional languages such as C at a lower level. Conditionals would be expressed in the conventional language. This is only a partial solution, however, because conditionals would be restricted to lie entirely within one large grain actor, and concurrency within such actors is difficult to exploit. If the complexity of the operations that are performed conditionally is high, then this approach is not adequate.

A simple alternative that is sometimes suitable is to replace *conditional evaluation* with *conditional assignment.* The functional expression

$$y \leftarrow \text{if } (c) \text{ then } f(x) \text{ else } g(x)$$

can be implemented as shown in figure 9. The MUX actor consumes a token on each of the T, F, and control inputs and copies either the T or F token to the output. Hence, both $f(x)$ and $g(x)$ will be computed and only one of the results will be used. When these functions are simple, this approach is efficient; indeed it is commonly used in deeply pipelined processors to avoid conditional branches. For hard-real-time applications, it is also efficient when *one of the two* subgraphs is simple. Otherwise, however, the cost of evaluating both subgraphs may be excessive, so alternative techniques are required.

4.4. Quasi-Static Scheduling

Statically scheduled control may prove acceptable for a subset of applications, perhaps mostly in signal processing. However, the limitations are quite serious, so there is strong motivation for removing them. At Berkeley we have been developing *quasi-static* scheduling strategies that may solve some of these problems [Ha89]. The basic principle is that dynamic control is used only where absolutely necessary. For instance, with an if-then-else, control is dynamically transferred to one of two statically scheduled subgraphs. Similarly, for a data-dependent iteration (such as a do-while), a static schedule for each cycle of the iteration is dynamically repeated. The challenge, of course, is to develop strategies for constructing the static schedules for the subgraphs. These techniques imply changes to the ordered-shared-

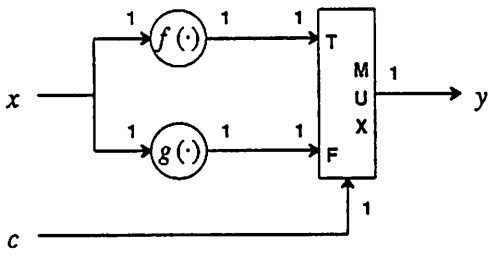

Figure 9. A dataflow graph with conditional assignment. Both $f(\cdot)$ and $g(\cdot)$ are evaluated, and only one of the two outputs is selected.

memory architecture. For further details, see [Ha89].

5. CONCLUSIONS

It is well known that data-independent dataflow graphs can be scheduled statically, obviating the need for additional runtime hardware to control the execution. We have illustrated low-cost parallel architectures that take advantage of this, and have resurrected reference counts, a mechanism for managing shared data structures that is well suited to static scheduling. Furthermore, we have shown that dataflow graphs can include recurrences and manifest iteration, and still be statically scheduled. The execution times of actors can vary slightly without seriously affecting the implementation, but wide variations can have considerable adverse impact on execution speed. Furthermore, conditional firing of actors is excluded from the model. For applications with little decision making, such as signal processing and some scientific computing, this approach appears attractive. To broaden the application base, quasi-static scheduling may provide a solution by introducing dynamic control only where absolutely necessary [Ha89].

REFERENCES

[Arv82]
Arvind and K. P. Gostelow, "The U-Interpreter", *Computer* **15**(2), February 1982.

[Arv87]
Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: Data Structures for Parallel Computing", Computation Structures Group Memo 269, MIT, February 1987 (revised March 1989). Also to appear in *ACM Transactions on Programming Languages and Systems.*

[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.

[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, L. M.-T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing", *IEEE Computer*, pp. 57-69, November, 1980.

[Chu87]
W. W. Chu and L. M.-T. Lan, "Task Allocation and Precedence Relations for Distributed Real-Time Systems", *IEEE Trans. on Computers*, **C-36**(6), pp. 667-679, June 1987.

[Coh85]
G. Cohen, D. Dubois, J. P. Quadrat, and M. Viot, "A Linear-System-Theoretic View of Discrete-Event Processes and its Use for Performance evaluation in Manufacturing", *IEEE Trans. on Automatic Control*, **AC-30**, 1985, pp. 210-220.

[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.

[Dri86]
J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, "Making Data Structures Persistent", in *Proc. of the 18th Ann. ACM Symp. on Theory of Computing*, Berkeley, CA, pp.109-121, May 1986.

[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).

[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.

[Gau86]
J. L. Gaudiot, "Structure Handling in Data-Flow Systems", *IEEE Trans. on Computers*, **C-35**(6), June 1986.

[Gau87]
J. L. Gaudiot, "Data-Driven Multicomputers in Digital Signal Processing", *IEEE Proceedings*, **Vol. 75, No. 9**, pp. 1220-1234, September, 1987.

[Gra87]
M. Granski, I. Koren, 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 and Assignment of Dataflow Program Graphs with Data-Dependent Iteration", Memorandum no. UCB/ERL M89/57, Electronics Research Lab, U. C. Berkeley, Berkeley, CA 94720.

[Hoa78]
C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, August 1978, 21(8)

[Hu61]
T. C. Hu, "Parallel Sequencing and Assembly Line Problems", *Operations Research*, 9(6), pp. 841-848, 1961.

[Hud86]
P. Hudak, "A Semantic Model of Reference Counting and its Abstraction", in *Proc. of the 1986 ACM Conf. on Lisp and Functional Programming*, MIT, Cambridge, MA, pp. 351-363, August, 1986.

[Ian88]
R. A. Iannucci, "A Dataflow/von Neumann Hybrid Architecture", Technical Report TR-418, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1988.

[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.

[Jag86]
H. V. Jagadish, R. G. Mathews, T. Kailath, and J. A. Newkirk, "A Study of Pipelining in Computing Arrays", *IEEE Trans. on Computers*, C-35(5), May 1986.

[Kel84]
R. M. Keller, F. C. H. Lin, and J. Tanaka, "Rediflow Multiprocessing", *Proc. IEEE COMPCON*, pp. 410-417, February, 1984.

[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).

[Lee86]
E. A. Lee, "A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors", *Memorandum No. UCB/ERL M86/54*, EECS Dept., UC Berkeley (PhD Dissertation), 1986.

[Lee87a]
E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Graph for Digital Signal Processing", *IEEE Trans. on Computers*, January, 1987.

[Lee87b]
E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow", *IEEE Proceedings*, September, 1987.

[Lee88]
E. A. Lee, "Programmable DSP Architectures, Part I", *ASSP Magazine*, October, 1988.

[Lee89a]
E. A. Lee, "Programmable DSP Architectures, Part II", *ASSP Magazine*, January, 1989.

[Lee89b]
E. A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP" *IEEE Trans. on ASSP*, November, 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.

[Muh87]
H. Muhlenbein, 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.

[Nik89]
R. S. Nikhil and Arvind, "Can Dataflow Subsume von Neumann Computing?", *Proc. of the 16th Annual Int. Symp. on Computer Architecture*, Jerusalem, Isreal, May 28 - June 1, 1989.

[Pap88]
G. M. Papadopoulos, *Implementation of a General Purpose Dataflow Multiprocessor*, Dept. of Electrical Engineering and Computer Science, MIT, PhD Thesis, August, 1988.

[Pet77]
J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

[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.

[Ren81]
M. Renfors and Y. Neuvo, "The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints", *IEEE Trans. on Circuits and Systems*, **CAS-28(3)**, March 1981.

[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.