RECURRENCES, ITERATION, AND CONDITIONALS IN STATICALLY SCHEDULED DATA FLOW

by

Edward A. Lee

Memorandum No. UCB/ERL M89/52

9 May 1989

COVER PAGE

RECURRENCES, ITERATION, AND CONDITIONALS IN STATICALLY SCHEDULED DATA FLOW

by

Edward A. Lee

Memorandum No. UCB/ERL M89/52

9 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

RECURRENCES, ITERATION, AND CONDITIONALS IN STATICALLY SCHEDULED DATA FLOW

by

Edward A. Lee

Memorandum No. UCB/ERL M89/52

9 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

RECURRENCES, ITERATION, and CONDITIONALS in STATICALLY SCHEDULED DATA FLOW[1]

Edward A. Lee

Assistant Professor
U. C. Berkeley
Berkeley, CA 94720
(415) 642-0455
eal@janus.Berkeley.EDU

ABSTRACT

Statically scheduled data flow is a much more efficient way of exploiting concurrency in algorithms than dynamically scheduled data flow, but the domain of algorithms that can be scheduled at compile time is limited. Iteration (for loops, do-while) and conditionals (if-then-else) are particularly difficult. This paper examines some data flow representations for these constructs and proposes compiler techniques for mapping them onto parallel processors. Since a minimal amount of dynamic scheduling is involved, the technique is termed *quasi-static* scheduling.

Index Terms — allocation, conditionals, data flow, iteration, parallel processing, partitioning, recurrences, recursion, static scheduling.

[1] This research is supported by NSF grant no. MIP-8657523 (PYI) and Cygnet Corp., a division of Everex Systems Inc.

1. MOTIVATION

Data flow representations of algorithms have concurrency that can be easily exploited, but the cost of exploiting it is often prohibitive when actors are scheduled dynamically (at run time). Even for large-grain actors, the hardware or software overhead of run-time scheduling is considerable. One solution is static (compile-time) scheduling, but the algorithms for which this can be done are restricted to those fitting the *synchronous data flow* (SDF) model [Lee87]. Actors that fire conditionally or an indeterminate number of times are excluded. The expressive power of languages based on SDF therefore excludes conditionals and data-dependent iteration.

One solution that has been attempted is to use static scheduling information to assist a dynamic scheduler [Gra87]. Although this avoids the restrictions in expressive power, Granski et. al. conclude that there is not enough performance improvement to justify the cost of the technique.

The aim of this paper is to expand static scheduling beyond the limitations of SDF without resorting to fully dynamic scheduling. In other words, instead of implementing the firing rule for *all* actors at runtime, it is only implemented for those actors or subgraphs that fire conditionally. Although this basic idea is both simple and obvious, the mechanics are not so simple. In fact, for some constructs, notably data-dependent iteration, we have found scheduling techniques that are satisfactory only for certain special cases, so further work is warranted.

Static scheduling is attractive because it permits efficient implementation of data flow graphs on parallel architectures where the processing elements are conventional microprocessors or programmable DSPs. In other words, special hardware is not required to support it. Furthermore, it is important for hard-real-time applications, where the data-dependent runtime of a dynamic schedule cannot be tolerated. Finally, it is *essential* for application specific IC implementations, where scheduling is done at *design* time, and then is frozen into hardware, as in silicon compilation. An example of a design-time scheduling technique is the synthesis of systolic arrays from a dependence-graph description of the algorithm [Rao85][Kun88], but the restrictions on the algorithm structure are even greater than those of SDF. We conclude that static scheduling is important enough to warrant considerable effort to extend the domain of algorithms for which it can be done.

In this paper, for clarity of exposition, I use a graphical representation of algorithms. For some application domains, such as digital signal processing, a language with a graphical syntax may be appropriate [Lee88], but it should be recognized that the techniques described here apply equally well to functional, applicative, and data flow languages that can be easily translated into data flow graphs for compilation.

We begin with a discussion of recurrences in data flow even though it is well known that recurrences can be statically scheduled. This serves as a forum to review and extend the mechanics of *delays*. Later we will combine iteration with recurrences and the extended delays will become important. Conditionals are the first non-static construct considered. A scheduling technique is proposed that is particularly appropriate for hard-real-time applications. Three types of iteration are then discussed: *manifest* iteration is where the number of iterations is known at compile time; *data determined* iteration is where the number of iterations is known before the iteration begins at run time, but is not known at compile time; and *convergent* iteration is where the number of iterations is not known until the iteration is finished. The last two are scheduled the same way, but their data flow representations are different.

2. RECURRENCES

Although the data flow literature sometimes claims that *recursion* is not possible in data flow languages, it should not be inferred that there is any difficulty expressing or implementing *recurrences*. Recurrences are computations where the current output depends on previous outputs. Recursion by contrast means self-referential functions; it is used in conventional languages to express recurrences as well as to express iteration. Tail recursion, for example, is usually used to express iteration. Recurrences are easily expressed in data flow graphs using directed loops.

A data flow graph with a recurrence is represented schematically in figure 1. The feedback path describes a recurrence. In order to avoid deadlock, any directed loop in the graph must have at least one *delay*. The delay, which corresponds to a z^{-1} operator in signal processing, is indicated with diamond containing a D.

To understand how recurrences can be implemented efficiently, it is helpful to review the SDF model. With any data flow model, an actor can fire whenever it has sufficient data at its inputs, i.e. the tokens it will consume must be available. It is up to the scheduler to determine when this is the case. In the SDF model [Lee87a][Lee87b], a special case of data flow, there are numbers adjacent to each arc to indicate the number of tokens that each actor will produce or consume on that arc when it fires. These numbers must be constant throughout the computation, and hence must be independent of the data. In figure 1 we have shown the simplest case where each actor consumes and produces exactly one token on each port each time it fires. If a data flow graph consists exclusively of such actors, then it is called a *homogeneous* SDF graph.

A *delay* does not correspond to unit time delay, but rather to a single token offset. It is simply an initial token on an arc. Such delays are sometimes called *logical delays* or *separators* to distinguish them from time delays [Jag86]. A logical delay need not be a run-time operation. Consider for example the feedback arc in figure 1, which has a unit delay. The initial token on the arc means that the corresponding input of actor

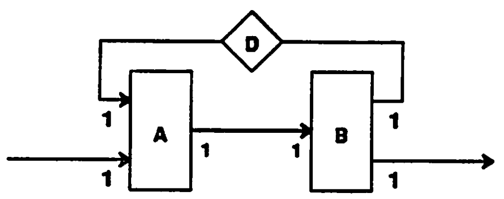

Figure 1. A data flow graph with a recurrence. Recurrences are expressed using directed loops and delays.

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; this explains the "delay" terminology. 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 conditions on the buffers, they require no run-time overhead.

It is obvious that directed loops without delays imply an immediate deadlock, since there are no initial tokens in the loop so no actor in the loop can fire. This situation can be automatically detected and flagged as an error [Lee87a].

Consider algorithms that run forever, or 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. For homogeneous SDF, where every actor produces and consumes a single sample on each input and output, it is easy to compute the minimum period at which the actor can be fired. This is called the ***iteration period bound***, and is the reciprocal of the maximum rate. Let $R(L)$ be the sum of the run times of the actors in a directed loop L. The run time of an actor is simply the time it takes to fire. The iteration period bound of a homogeneous SDF graph 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]. 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 bound to be zero, since in principle all firings of each node could occur simultaneously.

3. 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 2 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 2. 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 parallel schedules for such graphs are given in [Lee87a].

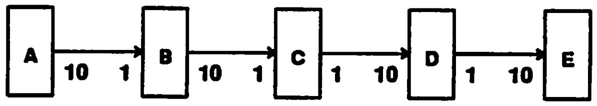

Figure 2. An SDF graph that contains nested iteration.

The efficiency of the implementation depends on the schedule and on the mechanism for buffering data passed between actors. For example, one possible single-processor schedule for figure 2 is

$$\{A, 10 \times B, 100 \times C, 10 \times D, E\},$$

and an alternative schedule is

$$\{A, 10 \times \{ B, 10 \times C, D\}, E\}.$$

The notation $10 \times C$ means that C is fired ten times in a row. The latter schedule will require less memory for buffering than the former. In either case, a brute force approach would be to synthesize in-line object code, but the resulting code size will be large. A better approach is to detect repeated patterns in the schedule for each processor and insert looping constructs into the object code.

A second issue is the effectiveness with which an algorithm is parallelized. There is no fundamental impediment to simultaneously firing successive invocations of an actor on parallel processors. Consider the iteration expressed in figure 2; the question arises as to whether successive invocations of actor C can fire in parallel. Since there is no directed loop anywhere in the graph, there is no fundamental impediment (the iteration period bound is zero). The only difficulties are practical.

One practical limitation on the parallelism arises from bounding the buffer sizes. One way to model bounded buffer sizes is with directed loops and delays [Kun88]. Consider figure 3, a modification of figure 2. Here we have added a feedback path with ten delays to model a buffer of length ten on the path from B to C. The tokens on the feedback path represent empty locations in the buffer. The total number of delays in the loop (ten) is equal to the size of the buffer. In this case there are no delays in the forward path, so they all get put on the feedback path. Actor B must have ten tokens on the feedback path (i.e. ten empty locations in the buffer) before it fires. Whenever actor C fires, it consumes a token from the forward path, freeing a buffer location, and indicating the free buffer location by putting a token on the feedback path. Notice that any buffer with length less than ten will lead to deadlock. This situation can be automatically avoided by properly choosing the buffer sizes [Lee87a].

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. Hence, even though B will be fired ten times for

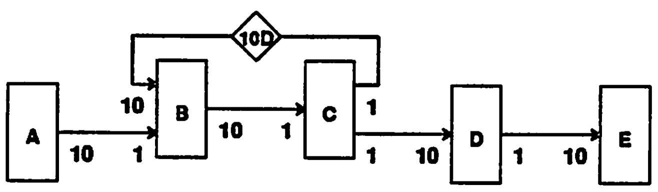

Figure 3. A modification of figure 2 to model the effect of a buffer of length ten between actors B and C.

each firing of A, the ten invocations of B cannot occur in parallel because of the buffer space limitations. By contrast if the buffer had length 100, then all 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 so that 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 effectively serializes 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), 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 3, which is a representation of figure 2 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.

Static buffering is key to efficient parallel firings of successive instances of the same actor. It avoids critical sections of code that must access the same data in parallel. In order to do static buffering, the lengths of the buffers must be carefully selected. For details, see [Lee87b].

In figure 2 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 4 we show four such actors that have proved useful for a wide variety of examples. The first, figure 4a, simply outputs the last of N tokens, where N is a parameter of the actor. The second, figure 4b, takes one input token and outputs a pattern that may or may not depend on the value of the input token. The pattern has the form $W \cdots WY \cdots Y$, where the values of W and Y are either a parameter of the actor or the value of the input token, and the number of repetitions of W and Y are parameters. The third, figure 4c, 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 a *state* of the actor; it effectively prevents multiple simultaneous invocations of the actor. We will see an example shortly that uses this actor.

The fourth actor, figure 4d, provides one of several mechanisms for initialization code that runs prior to an infinitely repeated algorithm, typical for example of real-time applications. It consumes a token exactly once, on its first invocation, and then outputs that token on every subsequent invocation. It is not an SDF actor because the number of tokens consumed depends on whether it is being invoked for the first time. This is indicated in figure 4d by providing a range (0,1) at the input instead of a fixed number of tokens consumed. The scheduling strategy is straightforward, and foreshadows techniques discussed later in the paper. The data flow graph is divided in two by a cutset consisting of the input arcs to all "repeat forever" actors. If it cannot be so divided, the graph is incorrectly constructed. Once it is divided, a schedule is

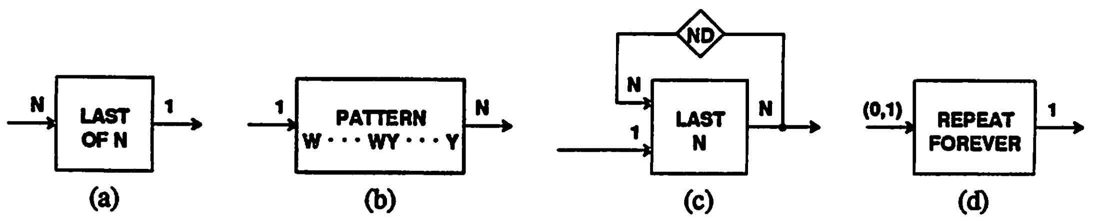

Figure 4. Four SDF actors useful for iteration.

constructed for the two subgraphs. Each schedule, even the one that will be repeated cyclically, can be constructed using the techniques given in [Lee87a].

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, but the reader should bear in mind that the issues are fundamental and apply to a wide variety of computations.

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 5. The iteration actors are drawn from figure 4. 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 selected by the "last of *N*" actor.

The FIR filter in figure 5 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

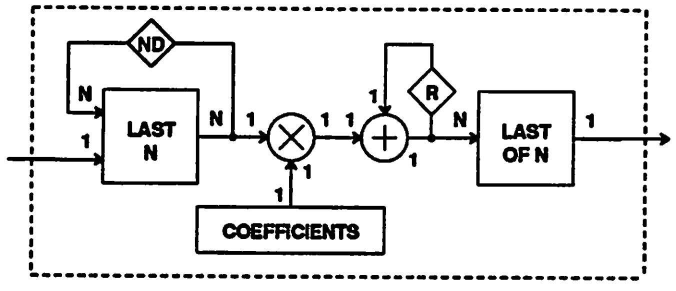

Figure 5. An FIR filter implemented using a single multiplier and adder.

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.

A resetting delay can be initialized to a data value that is supplied as an input to the subgraph associated with the delay. For example, the "repeat forever" actor in figure 4d is used in combination with a reseting delay in figure 6. The initialization subgraph is invoked only once to compute the initial value for the delay. This is represented as an input to the delay that must also be an input to the subgraph associated with the delay (enclosed in dashed lines). The delay can be viewed as an actor that consumes initialization data only when it resets.

The resetting delay can be used in general whenever we have nested iterations where the inner iterations involve recurrences using variables 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

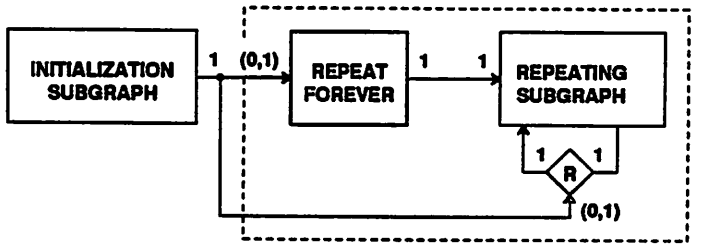

Figure 6. A data flow graph with an initialization subgraph that gets invoked only once and a repeating subgraph that gets invoked an infinite number of times. This is typical of real-time applications. The repeating subgraph is shown with a delay that gets initialized to the value computed by the initialization subgraph.

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 comprehensive mechanism for handling manifest iteration in data flow graphs, and for synthesizing efficient parallel implementations. It is worth mentioning that dependence graph methods handle manifest iteration using the notion of an *index space* [Kun88][Rao85] but have the significant disadvantage that all variables 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 the index space.

4. CONDITIONALS

Conditionals in data flow graphs are harder to describe and handle efficiently. One attractive solution is a mixed-mode programming environment, where the programmer can use data flow 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 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 probably not adequate.

A simple alternative that is frequently 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 7. The MUX actor consumes a token on each of the T, F, and control inputs and outputs either the T or F token. 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.

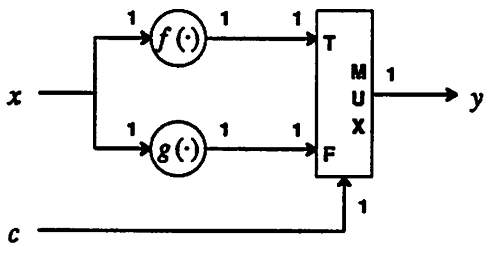

Figure 7. A data flow graph with conditional assignment. Both $f(\cdot)$ and $g(\cdot)$ are evaluated, and only one of the two outputs is selected.

As an example, consider the problem of squaring the entries of an array A[·] until the first negative entry. In pseudo-code,

```
        do I=1,N
                if (A[I] < 0) then goto L else A[I] := A[I] * A[I]
        end do
   L:   rest of program
```

This example combines iteration and conditionals. To translate this into a data flow graph, it is helpful to write a single-assignment version as follows:

```
        flag[1] := false
        do I=1,N
                if ((A[I] < 0) OR flag[I])
                then
                        B[I] := A[I]
                        flag[I+1] := true
                else
                        B[I] := A[I] * A[I]
                        flag[I+1] := false
        end do
```

A data flow graph implementing this algorithm is shown in figure 8. Both branches of the conditional are evaluated. In this simple example, this is efficient.

The above example not only illustrates combined iteration and conditionals, but also illustrates a data flow technique for handling arrays. Namely, the array elements form

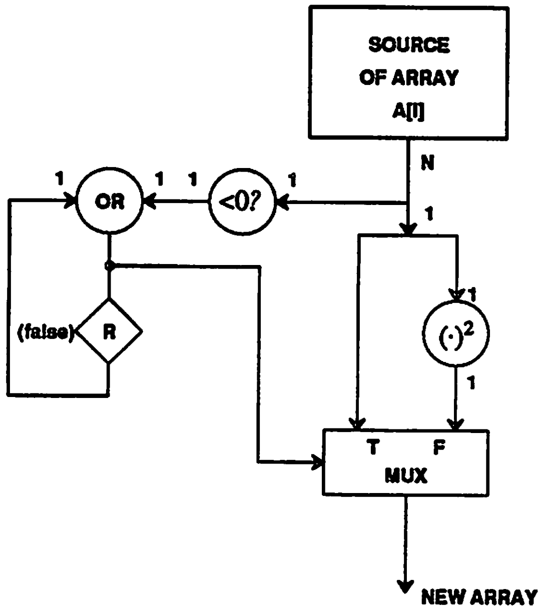

Figure 8. An SDF graph that squares the elements of an array until the first negative element of the array.

a stream, and instead of *modifying* the array, a new array is created. This is required (at least conceptually) in order to abide by the functional style of programming [Bac78]. Although this simple example can be made efficient, random access of data structures in data flow can be expensive; For a complete discussion of alternative techniques, see [Gau86].

An alternative data flow graph for an if-then-else is shown in figure 9a. A data token, x, is routed by the switch to one of two functions depending on the value of the boolean token c. The appropriate function fires, and its result is selected by the select actor. This data flow graph is not SDF because for the switch and select actors it is not possible to specify *a priori* the number of tokens produced and consumed on each input or output. For example, when the switch actor fires, it will produce a token on one of two outputs, depending on the condition c. Consequently, parallelizing compilers that work on SDF graphs will not work on graphs of this type. Instead, we propose *quasi-static scheduling*. In quasi-static scheduling, some firing decisions are made at run time, but only where absolutely necessary. One such scheduling strategy is illustrated in figure 9b for three processors. The figure shows two Gantt charts that indicate the activity of each of three processors over time. Two possible Gantt charts are shown, one for each possible branch decision. The only difficulty is ensuring that after each possible branch decision, the pattern of processor availability should be the same. Thus, although actors fire conditionally, the timing after they have fired is the

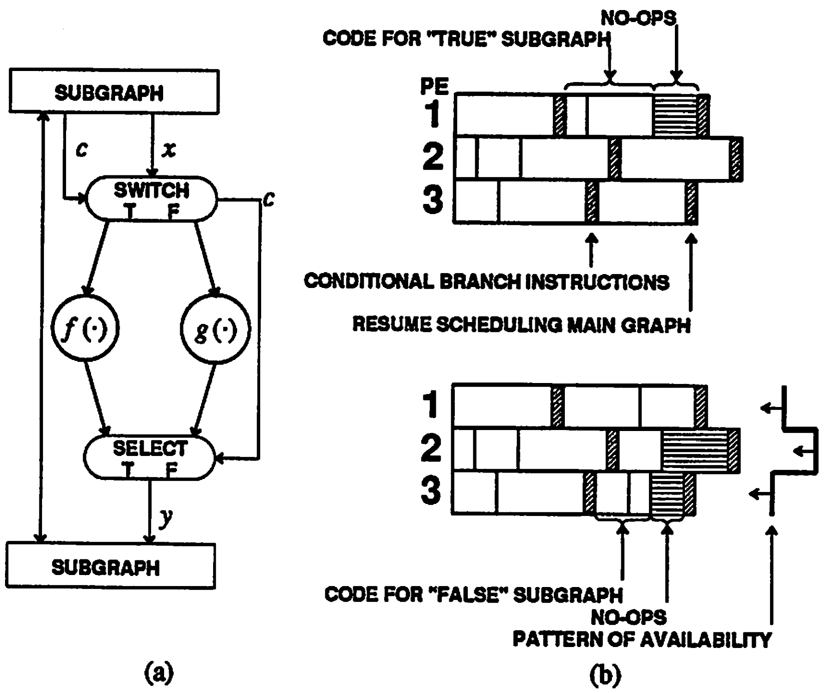

Figure 9. a. A data flow graph for the expression: y := if(c) then f(x) else g(x). We assume that $f(\cdot)$ and $g(\cdot)$ represent subgraphs of arbitrary complexity. b. Gantt charts for two schedules corresponding to two possible decisions. The schedules are padded with no-ops so that the pattern of availability after the conditional is independent of the decision.

same, so static scheduling can proceed.

To construct such a schedule, the data flow graph is divided into three subgraphs, the $f(\cdot)$, the $g(\cdot)$, and everything else. Each of the three subgraphs can have arbitrary complexity, and the $f(\cdot)$ and $g(\cdot)$ subgraphs can themselves have if-then-else constructs (as well as iteration). We will schedule the actors within each subgraph statically, and synthesize conditional branch instructions that effectively control the firing of entire subgraphs. Assume without loss of generality that the "everything else" graph is an SDF graph. It includes the switch and select actors. Since this subgraph will be treated by the scheduler as a separate graph, the control token c is routed through the switch actor to the select to ensure that the switch has precedence over the select within this subgraph. The "everything else" subgraph can be scheduled statically. To the left of the conditional branch instructions in figure 9b is a normal static schedule consisting of actors from the everything else graph. When it comes time to schedule the switch node, conditional branch instructions are spliced directly into the object code of the target processors. Then the scheduler calls itself recursively to construct a static schedule for the "true" subgraph, $f(\cdot)$, figure 9b (top). The only difference between this scheduling task and an ordinary SDF scheduling task is that the initial pattern of processor availability is arbitrary. In other words, instead of assuming all processors are available at the same time, different times are possible. This presents no difficulty, and ordinary critical path methods can be applied. If the true subgraph itself contains if-then-else constructs, then the scheduler will again call itself recursively, so arbitrary nesting of if-then-else's is permitted. When the true subgraph has been scheduled, the scheduler returns control to the top level scheduler, which then calls the scheduler recursively to schedule the "false" subgraph, $g(\cdot)$, figure 9b (bottom). The same initial pattern of processor availability is assumed, so the true and false schedules overlap in time. This explains the use of two Gantt charts to illustrate the schedule. When this scheduler returns, the two schedules, figure 9b top and bottom, are compared, and the worst case termination time on each processor is determined. The two schedules can then be padded with no-ops so that the pattern of processor availability is now independent of the branch taken. This is essential for static scheduling of the main graph to then resume.

This strategy is particularly well suited to real-time applications, where the schedule must complete in the same time regardless of the branch taken. In fact, if we assume that all branch decisions in the graph are independent, then there is no cost associated with the no-op padding. Of course, this assumption is often not realistic. The no-op padding itself can be omitted if synchronization between processors is enforced so that any processor that needs data from another processor will wait until that data is available.

For non-real-time applications an alternative suggested by Loeffler et. al. [Loe88] is attractive. It must be known which branch of the if-then-else is more likely. The higher probability branch is statically scheduled first, much as above, and the pattern of terminations is observed. Then the lower probability branch is scheduled to overlap in time, and padded with no-ops so that its *pattern* of terminations is the same. Note that only the *pattern* of terminations needs to be same, not the absolute termination times. Consequently, this approach works regardless of which branch takes longer to execute. However, the absolute termination times may differ, so the completion time of the overall schedule will depend on the branch taken. For this reason, this method is not attractive for hard-real-time applications.

In both cases, the presence of the if-then-else has an impact on the scheduling technique used for the main graph. In particular, suppose that a critical path method such as the Hu level scheduling method [Hu61] is being used [Lee87a]. This is a popular and effective suboptimal hueristic with manageable complexity. In this case, it is necessary to assign levels (loosely equivalent to priorities) to actors that feed data to the if-then-else, but the strictly speaking these levels depend on the branch decision. There are at least two possible levels that can be assigned to the switch actor itself, for example. For real-time applications, the largest of the possible levels should be assigned. For non-real-time applications, a weighted combination of the levels can be used, where the weights correspond to the probabilities of the decisions [Mar69].

Interestingly, this idea of assigning an *expected* Hu level to nodes in the graph was applied (without success) by Granski, et. al. to guide a *dynamic* scheduler [Gra87]. Applied to quasi-static scheduling, however, this approach has much more promise. Granski et. al. give a useful approximation to the mean critical path length that can be used to reduce the complexity of the compiler.

5. DATA-DEPENDENT ITERATION

We have shown how manifest iteration and conditionals can be efficiently scheduled, but for some iterative algorithms the number of iterations depends on the data. Such algorithms are not generally used in hard-real-time computations, but must be supported for more general applications. Data-dependent iteration is more difficult than conditionals. Nonetheless, viable quasi-static scheduling techniques exist. However, the techniques proposed here are close to optimal only for certain special cases.

We consider two cases: in *data determined* iteration, the number of iterations is known before the iteration begins but not known at compile time; in *convergent* iteration it is only known after the iteration is complete. The two will be scheduled the same way but expressed differently.

For data determined iteration, we simply need to modify the blocks in figure 4 to take an extra input token, the value of which will determine the number of tokens produced or consumed when the actor fires. The general framework is shown in figure 10. The

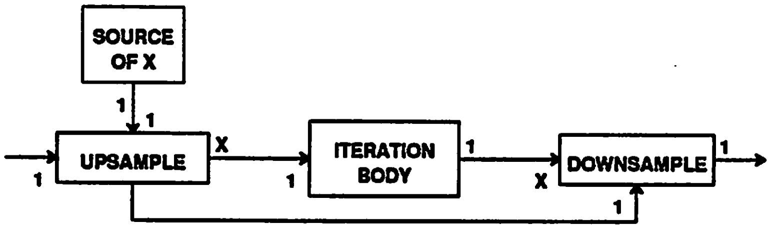

Figure 10. A general framework for data determined iteration.

upsample and downsample actors can be drawn from from those in figure 4, modified to take an extra input.

Consider for example a data flow graph that computes Y^X using data determined iteration and a recurrence, as shown in figure 11. The delay resets to one each time the subgraph indicated in the dashed lines is invoked. The "last of X" actor selects the final result by using the value of its control input token to determine how many tokens to consume.

Another example is a data flow graph that computes the factorial of a number using data determined iteration and a recurrence, as shown in figure 12. It is a special case of figure 10 where the upsample actor is omitted. The two delays are reset to zero and unity respectively each time the subgraph enclosed in the dashed line fires.

One difficulty that is immediately evident concerns buffering. The PATTERN actor in figure 11, for example, seems to require an output buffer with a size that depends on the data. Furthermore, when this buffer is large we have an inefficient use of

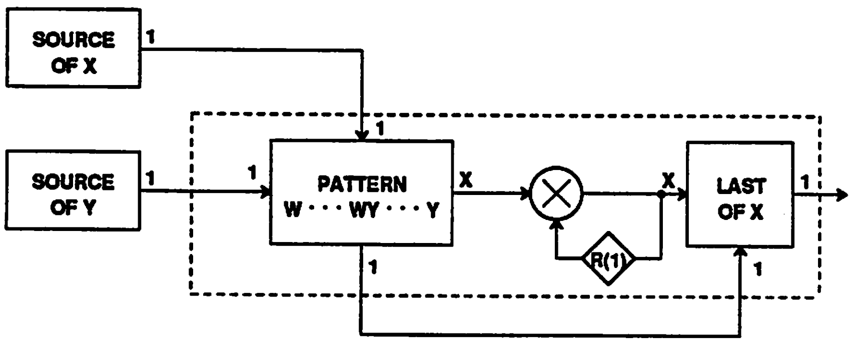

Figure 11. A data flow graph that computes Y^X using data determined iteration and a recurrence. The PATTERN actor is configured to repeat the Y input token a number of times given by the control input token (X). For example, to compute 10^2, the PATTERN actor outputs two successive tokens with value 10.

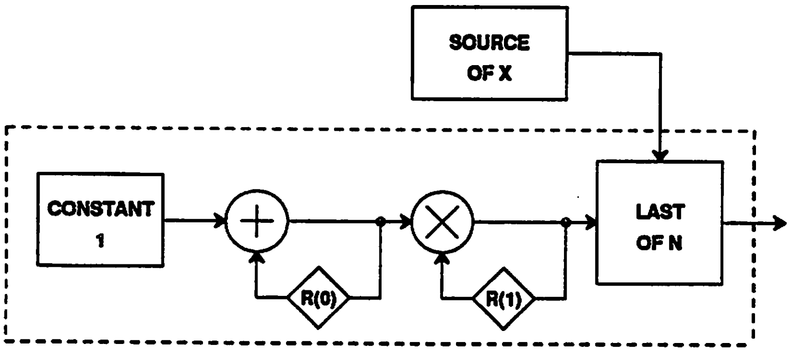

Figure 12. A data flow graph that computes the factorial of a number using data determined iteration and a recurrence.

memory. A smart compiler using static buffering, however, can generate code that requires at most two memory locations on the output buffer, one for the token in the first part of the pattern (W) and one for the token in the second part (Y). Similarly, the LAST OF X actor only requires one memory location for its input buffer because the tokens that will be discarded can be overwritten. In both cases, the compiler must have detailed knowledge of the functionality of the actors.

The two examples figure 11 and figure 12 contain recurrences in the body of the iteration. These recurrences limit the ability to schedule successive iterations in parallel. If the recurrences were absent, however, then there would be no fundamental impediment. In addition, successive invocations of the loop body can proceed in parallel, for example multiple factorial computations. However, practical difficulties arise with the mechanics of both buffering and scheduling, so further work is required to be able to exploit this parallelism.

Before considering the scheduling strategy for data determined iteration, let us consider convergent iteration as shown in figure 13. It implements the expression:

do {new x := f(x)} while t(x).

The difference here is that the data being computed in the iteration is used to determine when the iteration is finished (or has "converged"). The delay at the control input of the select actor should be an initial token with boolean value "false" so that the first token selected comes from the upper subgraph, rather than from the feedback loop. The body of the iteration itself is a recurrence. Both the boolean test token and the new value for x are fed back. Consequently, unlike data determined iteration, the

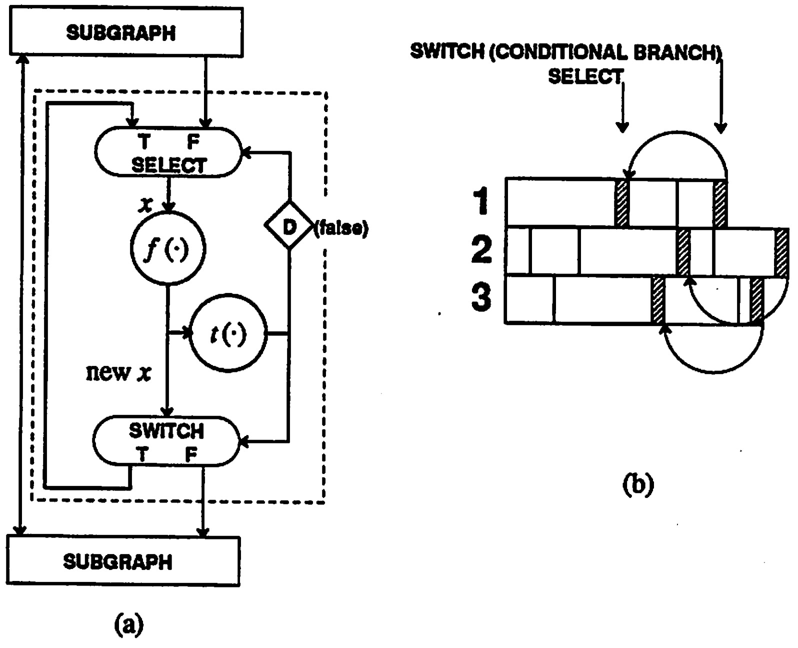

Figure 13. a. A do-while loop implementing y := do {x := f(x)} while t(x). b. A representation of a scheduling strategy where the iteration period is made the same on all processors.

iterative construct in figure 13 precludes multiple simultaneous invocations of successive iterations. Other variations on this graph, implementing for example a while-do, are easy to visualize. Because of the switch and select actors, it is not an SDF graph.

Although the data flow representation of convergent iteration is quite different from data determined iteration, similar scheduling strategies can be used. In either case, the graph is divided into two subgraphs, the body of the iteration and everything else. In figure 10, the upsample and downsample actors are included in the "everything else" subgraph because they are are synchronous with it. In figure 13, the switch and select actors are part of the iteration body. Assume without loss of generality that the outer subgraph in both cases is an SDF graph. Then it can be scheduled as normal. When it comes time to schedule the iteration, special action must again be taken. In both cases, code to implement a loop will be spliced into the object code for each processor used by the iteration. One iteration is then scheduled. When this is done, then the scheduler splices into the code for each processor conditional branch instructions. After these conditional branch instructions have been written, the scheduler can resume scheduling the outer subgraph. However, it is essential that the scheduler know the pattern of processor availability after the conditional branch instructions. It need not know the absolute times (indeed it cannot know them because the iteration is data dependent), but it must know the availability time of each processor relative to the others. In order for the pattern of availability to be independent of the number of iterations, it follows that the time spent by each processor in each iteration must be the same, as shown in figure 13b. If the scheduler puts less work onto one processor, then its schedule should be padded with no-ops.

Although the scheduling strategy outlined above seems reasonable, there are some serious disadvantages and some practical difficulties. One disadvantage is evident when the body of the iteration cannot make use of all of the processors. In the worst case, it will use only one processor, and all other processors will be idle during the entire time the do-while is being executed. Techniques are needed to make use of these idle processors. This involves deciding how many processors to allocate to the iteration. If this is done, it would permit, for example, successive invocations of the subgraph enclosed in dashed lines in figure 13 to occur simultaneously on different sets of processors.

One practical difficulty arises when trying to determine the priorities of actors that are used to compute the input to the do-while. Suppose for example that the Hu level scheduling algorithm [Hu61] is being used again. What level should be assigned to actors that come before the do-while? Again, a reasonable answer is provided in [Gra87] and [Mar67]. If the probability that the iteration will continue is constant and known, then the expected Hu level can be computed.

A second practical difficulty concerns the algorithm used to construct the schedule of the iteration actors. The scheduler will be given the original pattern of processor availability, and it must schedule one iteration in such a way that the pattern of availability is the same after the iteration as before. Hence the objective is to minimize the maximum span of one iteration of the schedule on each processor. This is almost the same as the original SDF scheduling problem! The only difference is the original pattern of availability. Unfortunately, known optimal algorithms have combinatorial complexity. If the iteration has few actors, this is not a serious problem, and an exhaustive method can be used to find the schedule. However, if the iteration has many actors, heuristics must be used.

One possibility is to use the Hu level scheduling algorithm [Hu61] combined with *blocked scheduling*, as proposed for SDF scheduling in [Lee87a]. In blocked scheduling, all processors are synchronized after each iteration. This effectively avoids dealing with the dependencies across iterations and reduces the scheduling problem to the standard one of minimizing the makespan of a graph with precedences. However, synchronizing all processors implies a flat pattern of availability both before and after the body of the iteration. This means that before the iteration is begun, the processors will have to be padded with no-ops until the time at which each is available is the same. This obviously implies wasted computations.

One way to view the blocked scheduling strategy is as *nested static schedules*. Consider an algorithm with nested iterations. The main (outer) SDF graph is scheduled statically with static sub-schedules for the bodies of the iterations. The interface between these static schedules, as well as the code that controls the number of invocations of each sub-schedule, is in essence performing dynamic scheduling. However, entire subgraphs are dynamically scheduled, rather than individual actors. The overhead is much lower and is incurred only when the algorithm demands it.

A possibly practical alternative to blocked scheduling is cyclo-static scheduling [Sch85]. Although cyclo-static scheduling algorithms can have combinatorial complexity, under certain circumstances the complexity is manageable for a moderate number of actors.

A final comment about the use of resetting delays in data dependent iteration is in order. If the resetting delay is to reset just prior to each new set of iterations, then the method proposed before breaks down. The compiler cannot compute the number of invocations of the delay occur between resets because it depends on the data! For both data determined and convergent iteration, however, the body of the iteration is scheduled as a unit, so the compiler can easily insert code to reset all relevant delays just prior to the code that implements the iteration.

6. CONCLUSIONS

Techniques are proposed for efficient parallel scheduling of recurrences, iteration, and conditionals in languages based on data flow. The schedules are mostly static, with run time overhead incurred only when the algorithm demands it. Thus the advantages of static scheduling are made available to a much broader class of algorithms than before. However, the methods for scheduling data dependent iterations may yield unsatisfactory schedules in some circumstances, implying a need for further work in this area.

7. ACKNOWLEDGEMENTS

The helpful comments of Douglas Niehaus are gratefully acknowledged.

REFERENCES

[Bac78]
J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM*, 21(8), August 1978.

[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.

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

[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.

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

[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.

[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 Graphs 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, E. Goei, H. Heine, W.-H. Ho, S. Bhattacharyya, and E. Guntvedt, "Gabriel: A Design Environment for DSP", submitted to *IEEE Trans. on ASSP*, July, 1988.

[Loe88]
C. Loeffler, A. Ligtenberg, H. Bheda, and G. Moschytz, "Hierarchical Scheduling System for Parallel Architectures", *Proceedings of Eusipco*, Grenoble, September, 1988.

[Mar67]
D. F. Martin and G. Estrin, "Models of Computations and Systems - Cyclic to Acyclic Graph Transformation", *IEEE Trans. Electron. Comput.*, EC-16, pp. 70-79, Feb. 1967.

[Mar69]
D. F. Martin and G. Estrin, "Path Length Computations on Graph Models of Computations", *IEEE Trans. on Computers*, C-18, pp. 530-536, June 1969.

[Rao85]
S. K. Rao, "Regular Iterative Algorithms and their Implementations on Processor Arrays", PhD

Dissertation, Information Systems Laboratory, Stanford University, October, 1985.

[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.

[Sch85]
D. A. Schwartz, "Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs", PhD Dissertation, Georgia Institute of Technology Technical Report DSPL-85-2, July 1985.