CHAPTER 31

RECURRENCES, ITERATION, and CONDITIONALS in STATICALLY SCHEDULED BLOCK DIAGRAM LANGUAGES[1]

Edward A. Lee
EECS Department
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Block diagrams have both practical and aesthetic appeal as a description of DSP algorithms, particularly when implementation on parallel hardware is contemplated, but their expressiveness is limited. Iteration (for loops, do-while) and conditionals (if-then-else) are particularly difficult to express cleanly. This paper examines some representations for these constructs and proposes compiler techniques for mapping onto parallel processors.

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

1. MOTIVATION

DSP algorithm designers would like to be able to quickly and easily experiment with algorithms without getting bogged down in an implementation quagmire. For this reason, considerable effort has gone into developing high-level software environments for DSP. Among the approaches that have been pursued are *signal representation languages* [Kop84][Mye86] and *block diagram languages*, see for example [Kel61][Mes84][Sny84][Sha87]. However, most such systems are suitable only for simulation on general purpose computers. For real-time implementation, special techniques are needed. In particular, ASIC implementations and programmable DSPs require *static scheduling*, in which the algorithm is mapped onto the hardware at *design time* or at *compile time*, but certainly not at *run time*. For example, systolic arrays can be synthesized from *dependence-graph* descriptions of an algorithm [Rao85][Kun88]. These methods have a distinct advantage of easily targeting locally connected multiple processors, whereas parallel implementations of block diagrams usually have less well-structured interconnection. However, restrictions on the structure of the algorithms limit their utility to a small class of well-structured applications. Block diagram languages, which can be viewed as data flow languages with a graphical syntax, have fewer restrictions; static scheduling is still possible as long as the application fits the *synchronous data flow (SDF)* model of computation [Ho88][Lee87a][Lee87b][Zis86][Scw85]. The limitations imposed by the SDF model are still serious, however. This paper addresses those limitations, and describes some simple techniques for extending the expressive power of statically scheduled block diagram languages.

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 means self-referential functions; it is used in conventional languages to express recurrences as well as to express iteration. Recurrences are easily expressed in block diagram languages using directed loops. Iteration will be addressed shortly.

A block diagram with a recurrence is represented schematically in figure 1. The feedback path describes a recurrence, and as with any discrete-time feedback path, it must have a delay to be computable. The delay, which corresponds to a z^{-1} operator, is indicated with diamond containing a D.

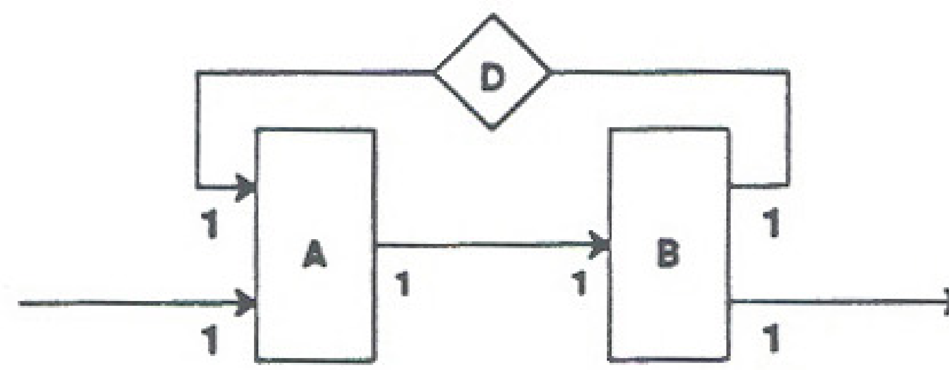

Figure 1. A block diagram with a recurrence. Recurrences are expressed using directed loops.

To understand how recurrences can be implemented efficiently, it is helpful to review the SDF model. With any data flow model, a block can fire whenever it has sufficient data at its inputs, i.e. the tokens (samples) 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 node 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 block consumes and produces exactly one token on each

port each time it fires. If a block diagram consists exclusively of such blocks, then we have a *homogeneous* SDF graph.

A delay is simply an initial token on an arc. It need not be a run-time operation, although many block diagram languages implement it as such. 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 node 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 node B. In steady-state, the nth firing of node B will produce a token that will be consumed by node A on its $(n+1)^{th}$ firing, which is exactly what we mean by a unit delay. 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 block can fire. This situation can be automatically detected and flagged as an error [Lee87a].

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 block produces and consumes a single sample on each input and output, it is easy to compute the minimum period at which the blocks can be invoked. This is called the *iteration period bound*, and is the reciprocal of the maximum rate. The iteration period bound of a homogeneous SDF graph is the maximum over all directed loops of the sum of the run times in the loop divided by the number of delays in the loop [Ren81][Coh85]. More general SDF graphs can be systematically converted to homogeneous SDF graphs for the purpose of computing the iteration period bound [Lee86].

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 block diagram languages by specifying the number of tokens produced and consumed each time a block is invoked. For example, block B in figure 2 will be invoked ten times for every invocation of block 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 blocks A and E are invoked once each, then B and D will be invoked ten times, and C will be invoked 100 times. Scheduling techniques that automatically construct appropriate schedules are given in [Lee87a].

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

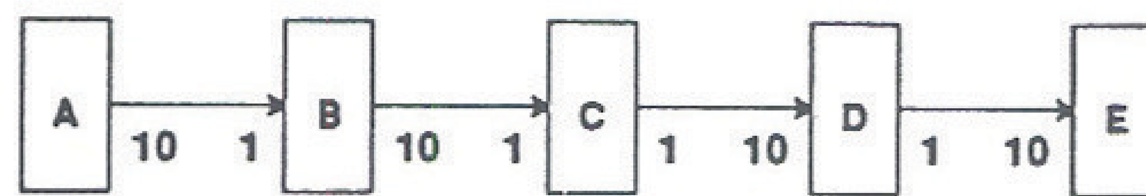

Figure 2. An SDF graph that contains nested iteration.

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

and an alternative schedule is

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

The notation $10 \times C$ means that C is invoked ten times in a row. The latter schedule will require less memory for buffering than the former.

A second issue is the effectiveness with which an algorithm is parallelized for computation on multiple processing elements (custom or programmable). There is no fundamental impedi-

ment to simultaneously firing successive invocations of blocks on parallel processors. Block diagram languages lend themselves to automatic task partitioning and synchronization [Lee87a]. Consider the iteration expressed in figure 2; the question arises as to whether successive invocations of block C can fire in parallel. Since there is no directed loop anywhere in the graph, so 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. The total number of delays in the loop (10) is equal to the size of the buffer. In this case there are no delays in the forward path, so they all get inserted in the feedback path. Notice that block B must have 10 tokens on the feedback path before it fires. These tokens represent empty positions in the buffer. Hence, whenever block 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 10 will lead to deadlock. This situation can be easily automatically avoided [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

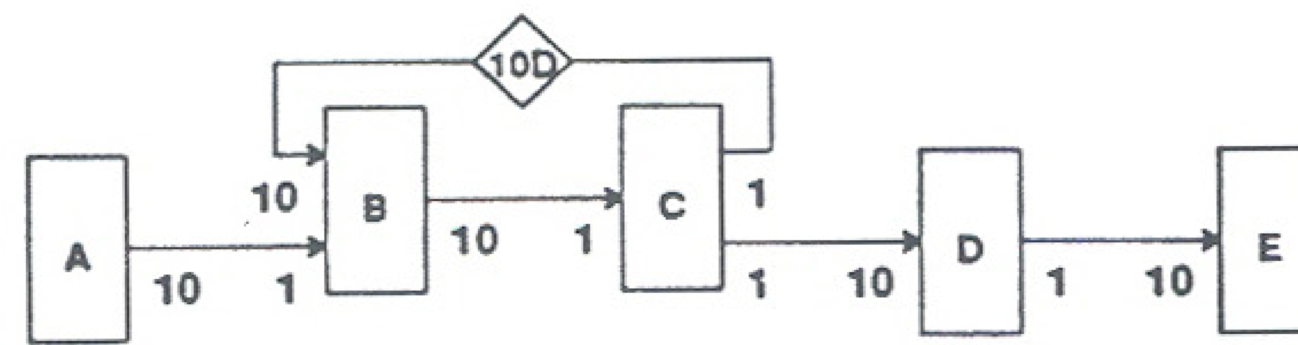

Figure 3. A modification of figure 2 to model the effect of a buffer of length 10 between blocks B and C.

can fire again. Hence, even though B will be invoked ten times for each invocation 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, 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 a block 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.

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. 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 block in an SDF graph. Every N firings, where N is to be determined, the block 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:

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

For each parallel invocation of C, the compiler supplies a *specific* memory location for it to get its input samples. Notice that this would not be possible if the circular buffer had length 11, for example. Because each of these invocations of C does not have to access a buffer through pointers, there is no contention for any particular memory location. The buffer data can be supplied to all ten invocations in parallel, assuming the hardware has a mechanism for doing this.

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 invocations of successive instances of the same block. 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].

A complete iteration model must include the ability to nest recurrences within iteration. We will illustrate this with an FIR filter implementation 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.

Consider the ways we could implement an FIR filter in a block diagram language. The most sensible way is probably to define a block 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 block diagram 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 4. When it fires, the LAST N block stores the incoming sample, and outputs that sample and the last $N-1$ samples that arrived on previous invocations, where N is the order of the FIR filter. Note that this block has a state, which can be viewed as data that flows from one invocation to the next. This is modeled as a self-loop that effectively prevents multiple simultaneous firings. The last N samples get multiplied by coefficients supplied by the COEFFICIENTS block and accumulated by the adder with a feedback loop. Finally the LAST OF N block selects the last of N of its input samples.

This description of an FIR filter is verbose, but it 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 blocks, 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 is an artifact of our implementation. Our solution is to introduce the notion of a *resetting delay*, indicated with a

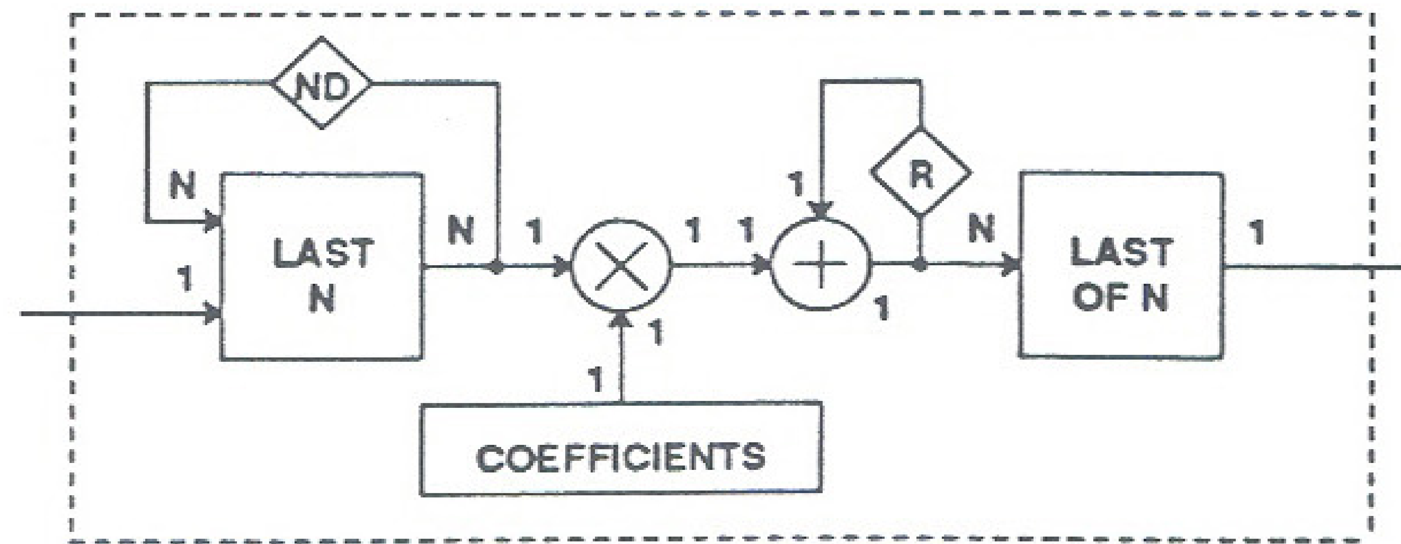

Figure 4. An FIR filter implemented using a single multiplier and adder.

diamond containing an R. The resetting delay is associated with a subgraph, which in this example is indicated with a dotted line. For each invocation of the subgraph, the delay token is reset 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 implementation of a resetting delay is simple and general.

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

We have given a comprehensive mechanism for handling manifest iteration in block diagrams, 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 block diagram languages are harder to describe and handle. A data flow graph for a functional if-then-else is shown in figure 5a. 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 block. The data flow graph is not SDF because for the switch and select blocks it is not possible to specify *a priori* the number of tokens produced and consumed on each input or output. The number is dependent on the data (the condition being tested). Consequently, parallelizing compilers that work on SDF graphs will not work in the presence of conditionals.

One attractive solution is a mixed mode programming environment, where the programmer can use block diagrams at the highest level and conventional languages such as C at a lower level [Ho88]. This is only a partial solution, however; conditionals are restricted to lie entirely within one block of the system, and concurrency within such blocks is difficult to exploit. If the complexity of the operations that are performed conditionally is high, then this approach is probably not adequate.

Another alternative is static scheduling of both branches of the if-then-else. Only the relevant result is selected. This is inefficient, however, unless one branch of the if-then-else is trivial.

A more attractive alternative is *quasi-static scheduling*. Static scheduling means that the firing of all nodes is determined at design time or compile time. 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 5b. The graph is divided into three subgraphs, the $f(\cdot)$, the $g(\cdot)$, and everything else. Each of the three subgraphs can have arbitrary

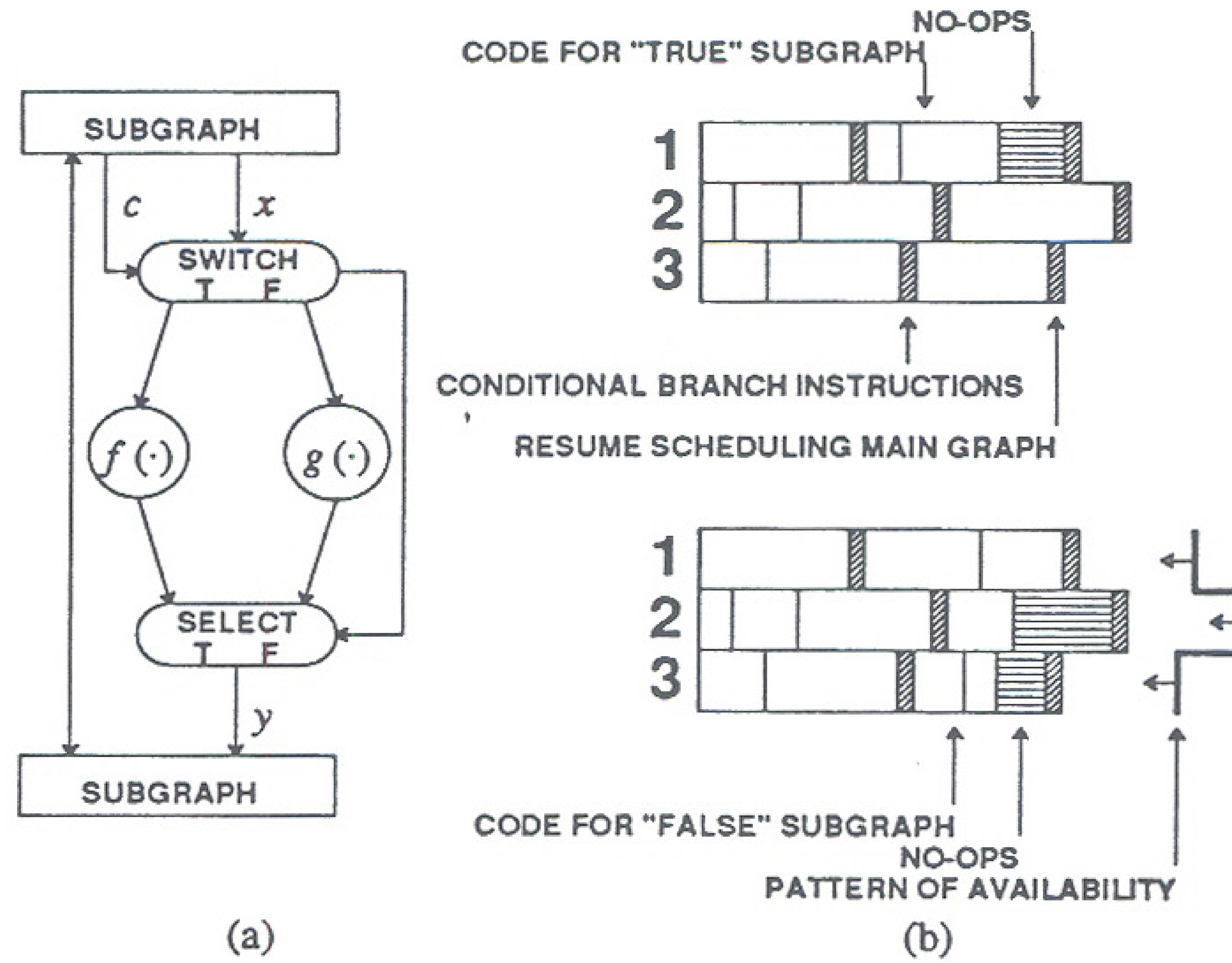

Figure 5. a. A data flow graph for the expression: y := if(c) then f(x) else g(x). We assume that $f(\cdot)$ and $g(\cdot)$ represent subgraphs of arbitrary complexity. b. Gantt charts for two schedules corresponding to two possible decisions. The schedules are padded with no-ops so that the pattern of availability after the conditional is independent of the decision.

complexity, and the $f(\cdot)$ and $g(\cdot)$ subgraphs can themselves have if-then-else constructs. Assume without loss of generality that the "everything else" graph is an SDF graph. It includes the switch and select blocks. Note that the control token c is routed through the switch block to the select to ensure that the switch has precedence over the select within this subgraph. The "everything else" subgraph can be scheduled statically. In figure 5b we illustrate a Gantt chart for a three processor schedule. The scheduler proceeds normally until it comes time to schedule the switch node. At this point, conditional branch instructions are spliced directly into the microcode or assembly code of the target processors. Then the scheduler calls itself recursively to construct a schedule for the "true" subgraph, $f(\cdot)$, figure 5b (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 5b (bottom). The same pattern of processor availability is assumed. When this scheduler returns, the two schedules, figure 5b 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. Scheduling of the main graph can then resume.

This strategy is particularly well suited to real-time applications, where the schedule must complete in a certain 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 not always 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. The probability of each branch of the if-then-else must be known. The higher probability branch is scheduled first and the pattern of terminations is observed. Then the lower probability branch is scheduled 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. Since the absolute termination times may differ, the completion time of the overall schedule will depend on the branch taken. For this reason, this method is not as attractive for 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 blocks 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 block itself. For real-time applications, the largest of the two 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.

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 can be efficiently scheduled, but for some algorithms the number of iterations depends on the data. Although such algorithms are not generally used in real-time computations, it is nonetheless important to be able to support them. 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.

Consider the data flow graph in figure 6. It implements the expression:

$$\text{do } \{x := f(x)\} \text{ while } t(x).$$

Because of the switch and select blocks, it is not an SDF graph. However, it can again be divided into subgraphs. Assume without loss of generality that the outer subgraph is an SDF graph. Then it can be scheduled as normal. When it comes time to schedule the select block, special action must again be taken. The delay at the control input of the select block 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 select block can be scheduled, after which the $f(\cdot)$ and $t(\cdot)$ subgraphs can be scheduled. When these are done, then the scheduler implements the switch block by splicing into the code conditional branch instructions. As long as $t(x)$ evaluates to "true", each processor will branch back to where the select block was scheduled, as shown in figure 6b. 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

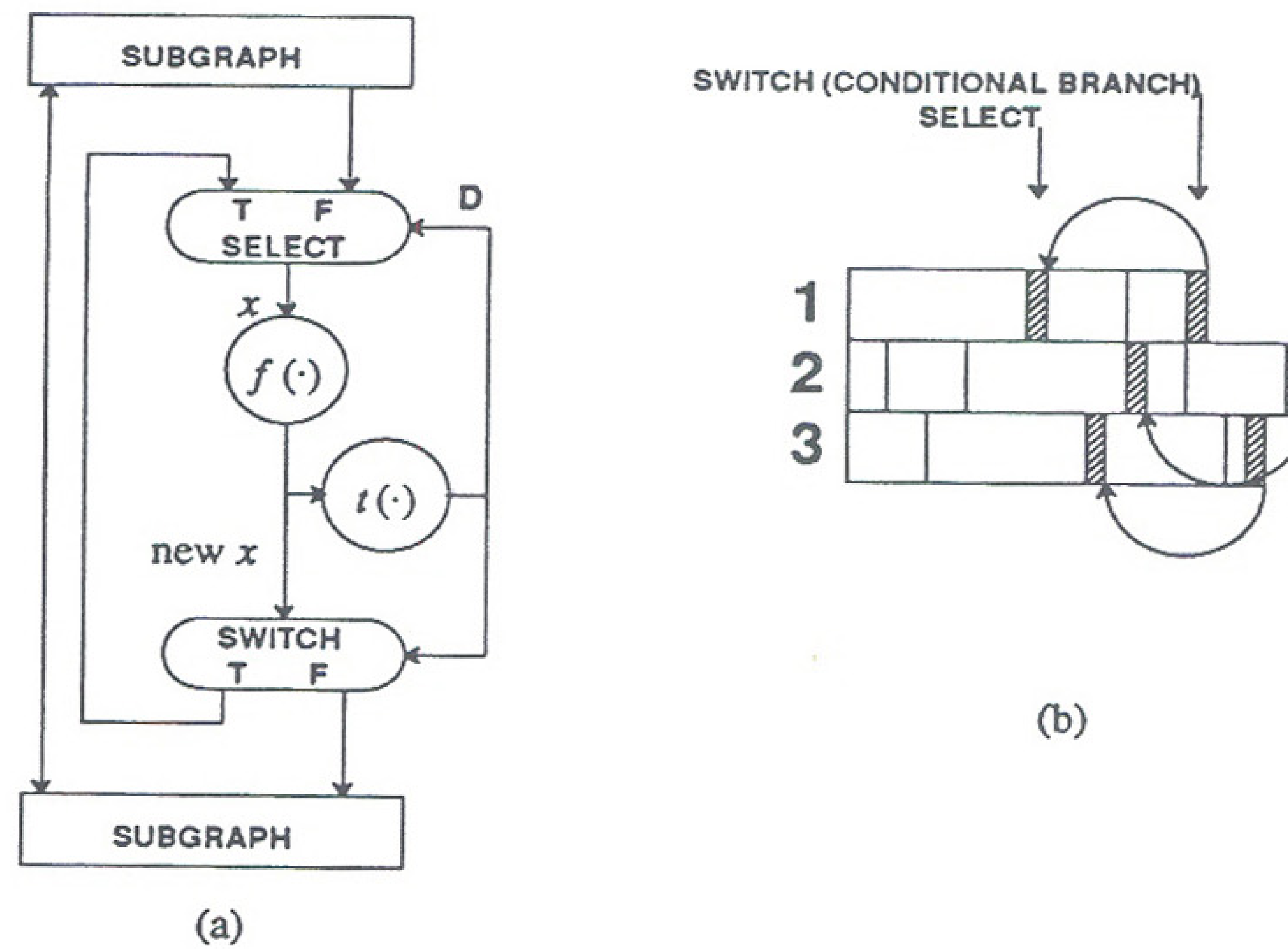

Figure 6. a. A do-while loop implementing y := do {x := f(x)} while t(x). b. A representation of a scheduling strategy where the iteration period is made the same on all processors.

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 6b. 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 inner loop cannot make use of all of the processors. In the worst case, it will use only one processor, and all other processors will be executing no-ops during the entire time the do-while is being executed. However, there appears to be no way to avoid this and still be able to resume static scheduling of the outer subgraph once the conditional loop is finished.

One practical difficulty arises when trying to determine the priorities of blocks 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 blocks that come before the do-while? Again, a reasonable answer is provided by Granski, et. al. [Gra87]. If the expected number of iterations is known, then the expected Hu level can be computed.

A second practical difficulty concerns the algorithm used to construct the schedule of the iteration blocks, $f(\cdot)$ and $t(\cdot)$. 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 blocks, this is not a serious problem, and an exhaustive method can be used to find the schedule. However, if the iteration has many blocks, 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. Another possibility is cyclo-static scheduling [Sch85]. Although the techniques proposed there can have combinatorial complexity, under certain circumstances the complexity is manageable for a moderate number of blocks.

The above arguments are easily extended to other data-dependent iteration forms such as while-do.

6. CONCLUSIONS

Techniques are proposed for efficiently compiling iteration and conditionals in languages based on data flow. Block diagram languages thereby acquire the full expressive power of conventional programming languages, but retain both the appealing user model and the ability to efficiently and automatically compile them for parallel computation. The same techniques can be applied to languages that are semantically similar to block diagram languages, such as functional and applicative languages. It is expected therefore that these techniques will become essential in software DSP design environments. However, the methods for scheduling data dependent iterations may yield unsatisfactory schedules in some circumstances, implying a need for further work in this area.

REFERENCES

[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.

[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.

[Kel61]
Kelly, Lochbaum, and Vyssotsky, "A Block Diagram Compiler", Bell Sys. Tech. J., **40(3)**, May, 1961.

[Kop84]
G. E. Kopec, "The Integrated Signal Processing System ISP" *IEEE Trans. on ASSP* **32(4)**, August, 1984.

[Kun88]
S. Y. Kung, *VLSI Array Processors*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Ho88]
W.-H. Ho, E. A. Lee and D. G. Messerschmitt, "High Level Data Flow Programming for Digital Signal Processing", in *VLSI DSP III* (this issue), IEEE Press, 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.

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

[Mes84]
D. G. Messerschmitt, "A Tool for Structured Functional Simulation", *IEEE Journal on Selected Areas in Communications*, SAC-2(1), January, 1984.

[Mye86]
C. Myers, "Signal Representation for Symbolic and Numerical Processing", MIT Tech. Report No. 521, Research Laboratory of Electronics (Ph.D. Thesis), August 1986.

[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.

[Sha87]
K. S. Shanmugan, G. J. Minden, E. Komp, T. C. Manning, and E. R. Wiswell, "Block-Oriented System Simulator (BOSS)", Telecommunications Laboratory, University of Kansas, 1987.

[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.

[Sny84]
L. Snyder, "Parallel Programming and the Poker Programming Environment", *Computer*, **17(7)** July, 1984.

[Zis86]
M. A. Zissman, G. C. O'Leary, and D. H. Johnson, "A Block Diagram Compiler for a Digital Signal Processing MIMD Computer", DSP Workshop Presentation, October 1986, Chatham, MA.