COMPILE-TIME SCHEDULING AND ASSIGNMENT OF DATAFLOW PROGRAM GRAPHS WITH DATA-DEPENDENT ITERATION

by

Soonhoi Ha and Edward Ashford Lee

Memorandum No. UCB/ERL M89/57

16 May 1989

COVER PAGE

COMPILE-TIME SCHEDULING AND ASSIGNMENT OF DATAFLOW PROGRAM GRAPHS WITH DATA-DEPENDENT ITERATION

by

Soonhoi Ha and Edward Ashford Lee

Memorandum No. UCB/ERL M89/57

16 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

COMPILE-TIME SCHEDULING AND ASSIGNMENT OF DATAFLOW PROGRAM GRAPHS WITH DATA-DEPENDENT ITERATION

by

Soonhoi Ha and Edward Ashford Lee

Memorandum No. UCB/ERL M89/57

16 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

COMPILE-TIME SCHEDULING AND ASSIGNMENT OF DATAFLOW PROGRAM GRAPHS WITH DATA-DEPENDENT ITERATION

Soonhoi Ha
Edward Ashford Lee

U. C. Berkeley
Berkeley, CA 94720
(415) 643-6686

May 16, 1989

ABSTRACT

Scheduling of dataflow graphs onto parallel processors consists of assigning actors to processors, ordering the execution of actors within each processor, and firing the actors at particular times. Many scheduling strategies do at least one of these operations at compile time to reduce run-time cost. In this paper, we classify four scheduling strategies, (1) fully dynamic, (2) static-assignment, (3) self-timed, and (4) fully static. These are ordered in decreasing run-time cost. Optimal or near-optimal compile-time decisions require deterministic, data-independent program behavior known to the compiler. Thus, moving from strategy number (1) towards (4) either sacrifices optimality, decreases generality by excluding certain program constructs, or both. This paper proposes scheduling techniques valid for strategies (2), (3), and (4). In particular, we focus on dataflow graphs representing data-dependent iteration; for such graphs, although it is impossible to deterministically optimize the schedule at compile time, reasonable decisions can be made. For many applications, good compile-time decisions remove the need for dynamic scheduling or load balancing. We assume a known probability mass function for the number of cycles in the data-dependent iteration, and show how a compile-time decision about assignment and/or ordering as well as timing can be made. The criterion we use is to minimize the expected total idle time due to the iteration; in certain cases, this will also minimize the expected makespan of the schedule. We will also show how to determine the number of processors that should be assigned to the data-dependent iteration.

1. INTRODUCTION

A dataflow representation is suitable for programming multiprocessors because parallelism can be extracted automatically from the representation [Ack82][Gos82]. Each node, or actor, in a dataflow graph represents a task to be executed according to the precedence constraints represented by arcs, which also represent the flow of data. Nodes in a dataflow graph are to be scheduled in such a way as to achieve the fastest execution from a given multiprocessor architecture. We make no assumption here about the granularity of the dataflow graph. The proposed techniques are valid both for fine-grain and large-grain.

Scheduling of parallel computations consists of assigning actors to processors, ordering the actors on each processor, and specifying their firing time, each of which can be done either at compile time or at run-time. Depending on which operations are done when, we define four classes of scheduling. 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 at run-time. 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 as well as assigning them to the processors. At run-time, the processor waits for data to be available for the next actor in its ordered list, and then fires that actor. We call this *self-timed* scheduling because of its similarity to self-timed circuits. The fourth type of scheduling is *fully static*; here the compiler determines the exact firing time of actors, as well as their assignment and ordering. This is analogous to synchronous circuits. As with most taxonomies, the boundary between these categories is not rigid.

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 [Sri86]. Dataflow machines evaluate dataflow graphs at run time, but a commonly adopted practical compromise is to allocate the actors to processors at compile time. 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 dramatically reduced 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 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]. Another automated example is a dataflow programming system for digital signal processing called Gabriel that targets multiprocessor systems made with programmable DSPs [Lee89]. 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.

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 [Lee87]. Constructs such as conditionals and data-dependent iteration make this impossible and realistic I/O behavior makes it impractical. The concept of static scheduling has been extended to solve some of these problems, using a technique called *quasi-static scheduling* [Lee89]. In quasi-static scheduling, some firing decisions are made at run-time, but only where absolutely necessary.

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. Signal processing algorithms, for example, generally fit this model [Lee87]. The run-time overhead is very low, consisting only of simple handshaking mechanisms. Furthermore, provably optimal (or close to optimal) schedules are viable. As with fully static scheduling, data-dependent behavior is excluded if the resulting schedule is to be optimal. Again, quasi-static scheduling solves some of the problems, but data-dependent iteration has been out of reach except for certain special cases.

Static-assignment scheduling is a compromise that admits data dependencies, although 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].

Fully dynamic scheduling is most able to utilize the resources and to fully exploit the concurrency of a dataflow representation of an algorithm. 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. One attempt to do this by using static (compile-time) information to assign priorities to actors to assist a dynamic scheduler was rejected by Granski et. al., who conclude that there is not enough performance improvement to justify the cost of the technique [Gra87].

In view of the high cost of fully dynamic scheduling, static-assignment and self-timed are attractive alternatives. Self-timed is more attractive for scientific computation and digital signal processing, while static-assignment is more attractive where there is more data dependency. Consequently, it is appropriate to concentrate on finding good compile-time techniques for these strategies. In this paper we propose a way to schedule a data-dependent iteration for general cases with the assumption that the probability distribution of the number of cycles of the iteration is known or can be approximated at compile time. The technique is not optimal except in certain special cases, but it is intuitively appealing and computationally tractable.

2. DATA-DEPENDENT ITERATION

Two possible dataflow representations for data-dependent iteration are shown in figure 1 [Lee89]. The numbers adjacent to the arcs indicate the number of tokens produced or consumed when an actor fires [Lee87a]. In figure 1a, since the upsample actor produces X tokens each time it fires, and the iteration body consumes only one token when it fires, the iteration body must fire X times for each firing of the upsample actor. In figure 1b, the number of iterations need not be known prior to the commencement of the iteration. Here, a token coming in from above is routed through a "select" actor into the iteration body. The "D" on the arc connected to the control input of the "select" actor indicates an initial token on that arc with value "false". This ensures that the data coming into the "F" input will be consumed the first time the "select" actor fires. After this first input token is consumed, the control input to the "select" actor will have value "true" until the function $t(\cdot)$ indicates that the iteration is finished by producing a token with value "false". During the iteration, the output of the iteration function $f(\cdot)$ will be routed around by the "switch" actor, again until the test function $t(\cdot)$ produces a token with value "false". There are many variations on these two basic models for data-dependent iteration.

For simplicity, we will group the body of a data-dependent iteration into one node, and call it a data-dependent iteration actor. In other words, we assume a hierarchical dataflow graph. In figure 1a, the "iteration body" actor consists of the upsample, data-dependent iteration, and downsample actors. The data-dependent iteration actor may consist of a sub-graph of arbitrary complexity, and may itself contain data-

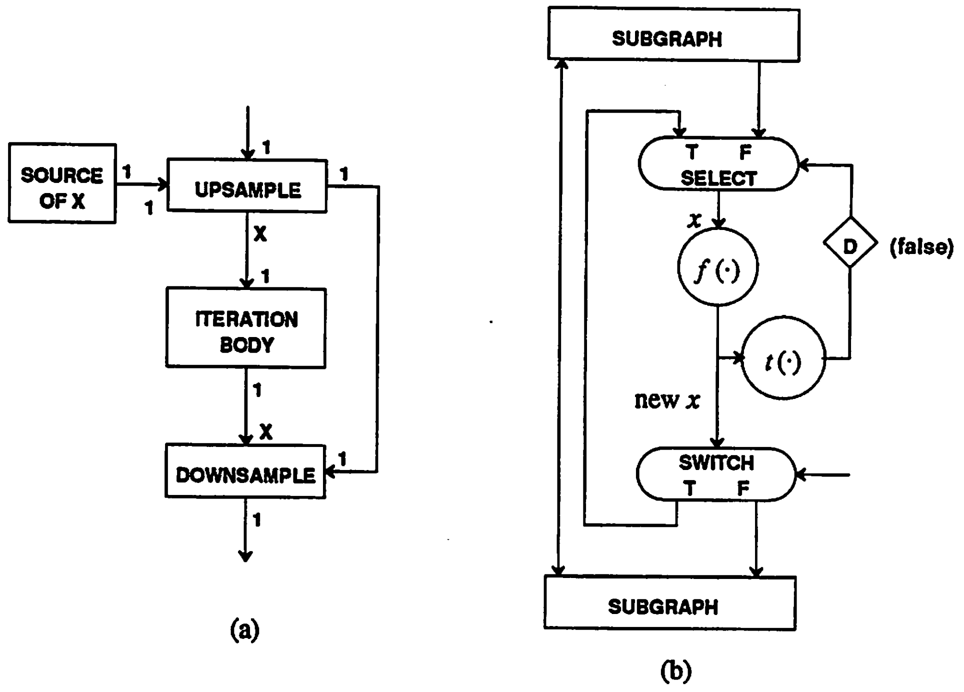


Figure 1. Data-dependent iteration can be represented using the either of the dataflow graphs shown. The graph in (a) is used when the number of iterations is known prior to the commencement of the iteration, and (b) is used otherwise.

dependent iterations. In figure 1b, everything between the "select" and the "switch", inclusive, is the data dependent iteration actor. In both cases, the data-dependent iteration actor can be viewed as an actor with a stochastic runtime, but unlike atomic actors, it can be scheduled onto several processors. Although our proposed strategy can handle multiple and nested iteration, for simplicity all our examples will have only one iteration actor in the dataflow graph.

The proposed scheme has two components. First, the compiler must determine which processors to allocate to the data-dependent iteration actor. These will be called the "iteration processors", and the rest will be called "non-iteration" processors. Second, the data-dependent iteration actor is optimally assigned an *assumed execution time* to be used by the scheduler. In other words, although its runtime will actually be random, the scheduler will assume a carefully chosen deterministic runtime and construct the schedule accordingly. The assumed runtime is chosen so that the expected total idle time due to the difference between the assumed and actual runtimes is minimal. Locally minimizing idle time is well known to fail to minimize expected makespan, except in certain special cases. (The makespan of the schedule is defined to be the time from the start of the computation to when the last processor finishes.) We will discuss these special cases and argue that the strategy is nonetheless promising, particularly when combined with other heuristics.

Using the assumed execution time, a fully-static schedule is constructed. When the program is run, the execution time of data-dependent actors will probably differ from the assumption, so processors must be synchronized. If all processors are synchronized together, using for example a global "enable" line, then we say the execution if "quasi-static". It is not fully static because absolute firing times depend on the data. If processors are pairwise synchronized, then the execution is self-timed or static-assignment, depending on whether ordering changes are permitted.

The assumed execution time and the number of processors devoted to the iteration together give the scheduler the information it needs to schedule all actors around the data-dependent iteration. It does not address, however, how to schedule the data-dependent iteration itself. We will not concentrate on this issue because it is the standard problem of statically scheduling a periodic dataflow graph onto a set of processors [Lee87a]. Nonetheless, it is worth mentioning techniques that can be used. To reduce the computational complexity of scheduling and to allow any number of nested iterations without difficulty, *blocked scheduling* can be used. In blocked scheduling, all iteration processors are synchronized after each cycle of the iteration so that the pattern of processor availability is flat before and after each cycle (meaning that all processors become available for the next cycle at the same time). If the scheduling is fully static, then this can be accomplished by padding with no-ops so that each processor finishes a cycle at the same time. The wasted computation can be reduced using advanced techniques such as *re-timing* or *loop-winding*[1] [Lei83][Gir87]. In these techniques, several cycles of an iteration are executed in parallel to increase the overall throughput. For blocked scheduling, the objective is to minimize the makespan of one cycle. Throughput can also be improved using optimal periodic scheduling strategies, such as cyclo-static scheduling [Sch86]. The proposal below applies regardless of which method is used, but in all our illustrations we assume blocked scheduling. We similarly avoid specifics about how the scheduling of the overall dataflow graph is performed. Our method is consistent with simple heuristic scheduling algorithms, such as Hu-level scheduling [Hu61], as well as more elaborate methods that attempt, for example, to reduce interprocessor communication costs. Broadly, our method can be used to extend any deterministic scheduling algorithm (based on execution times of actors) to include data-dependent iteration.

3. THE ASSUMED EXECUTION TIME

To schedule the actors around the data-dependent iteration actor at compile time, it is necessary to assign some fixed execution time to the data-dependent iteration actor. Since the number of cycles of the iteration to be executed is not known at compile time, we have to assume a number. The first guess might be to simply assume the *expected* execution time, which can be approximated using methods proposed by Martin and Estrin [Mar69], but this will often be far from optimal. In fact, the assumed number should depend on the ratio of the number of iteration processors to the total number of processors. When the actual execution time differs from the assumed runtime, some processors will be idled as a consequence. Our strategy is to find the assumed runtime that minimizes the expected value of this idle time. We make the bold assumption that the probability distribution of the number of cycles of the

[1] As a possibly interesting side-issue, it does not appear to have been pointed out in the literature that retiming is simply a dataflow perspective on loop-winding, so the techniques are in fact equivalent.

iteration actor is known or can be approximated at compile time.

Let the number of cycles of an iteration be a random variable I with known probability mass function $p(i)$. Denote the minimum possible value of I by MIN and the maximum by MAX. MAX need not be finite. In this section, we assume that we have already allocated somehow the number N of processors to the data-dependent iteration actor. How to allocate the number of processors will be addressed in the next section. If the total number of the processors is T, the number of non-iteration processors is $T-N$.

Let the assumed execution time of the data-dependent iteration actor be t. For the time being we restrict t to multiples of the execution time of one cycle of the iteration. If the execution time of a cycle is τ, then $x = t/\tau$ denotes the assumed number of cycles of the iteration. At run time, for each invocation of the iteration actor, there are three possible outcomes: the actual number i of cycles of the iteration is (1) equal to, (2) greater than, or (3) less than x. These cases are displayed in figure 2. In order for the scheduler to resume static scheduling after the iteration is complete, it must know the "pattern of processor availability". As indicated in figure 2, this pattern simply defines the relative times at which processors become free after the iteration. For now, assume this pattern is strictly enforced by some global synchronization mechanism, regardless of the number of iteration cycles actually executed at run time. This will force either the iteration processors or the non-iteration processors to be idle, depending on whether the iteration finishes early or late. This constraint is precisely what we mean by "quasi-static" scheduling of data-dependent iterations. It is not strictly static, in that exact firing times are not given at compile time, but relative firing times *are* enforced.

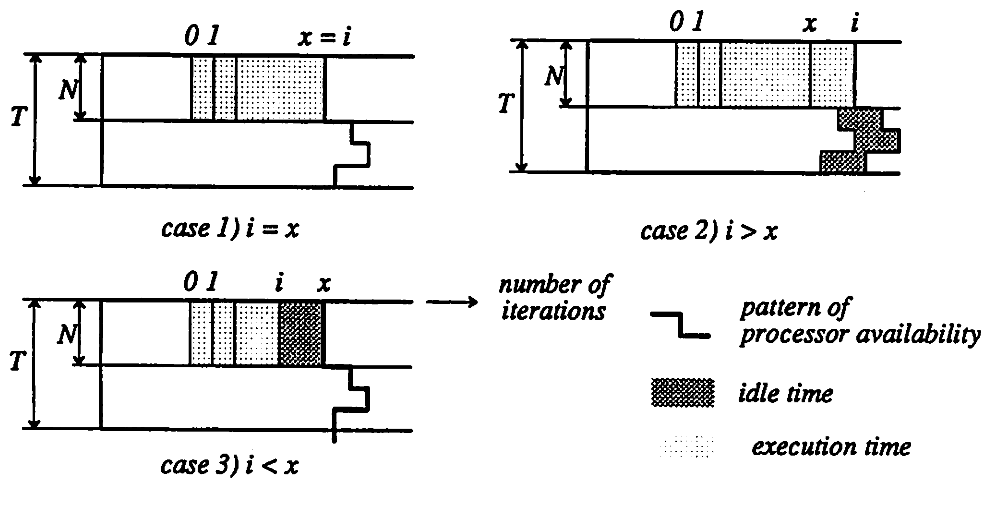


Figure 2. A static schedule is constructed using a fixed assumed number x of cycles in the iteration. The idle time due to the difference between x and the actual number of cycles i is shown for 3 cases: i is equal to, less than, or greater than the assumed number, x.

Consider the case where the assumed number x is exactly correct. Then no idle time exists on any processor (case 1, figure 2). Otherwise, the non-iteration processors will be idled if the iteration takes more than x cycles (case 2, figure 2), or else the iteration processors will be idled (case 3, figure 2). Our strategy is to select x to minimize the expected idle time on all processors for a given number of iteration processors.

Let $p(i)$ be the probability mass function of the number of iteration cycles confined within MIN and MAX. For a fixed assumed x the expected idle time $t_1(x)$ on the iteration processors is

$$t_1(x) = N\tau \sum_{i=MIN}^{x} p(i)(x-i) . \tag{1}$$

The expected idle time $t_2(x)$ on the non-iteration processors is

$$t_2(x) = (T-N)\tau \sum_{i=x+1}^{MAX} p(i)(i-x) . \tag{2}$$

The total expected idle time $t(x)$ is $t(x) = t_1(x) + t_2(x)$. The optimal value of x minimizes this quantity. From this we can get that

$$\begin{aligned} t(x) - t(x+1) &= -N\tau \sum_{i=MIN}^{x} p(i) + (T-N)\tau \sum_{i=x+1}^{MAX} p(i) \\ &= -N\tau + T\tau \sum_{i=x+1}^{MAX} p(i) , \end{aligned} \tag{3}$$

Similarly,

$$t(x) - t(x-1) = N\tau - T\tau \sum_{i=x}^{MAX} p(i) . \tag{4}$$

The optimal x will satisfy the following two inequalities: $t(x) - t(x+1) \le 0$, $t(x) - t(x-1) \le 0$. Since τ is positive, from the equations (3), (4),

$$\sum_{i=x+1}^{MAX} p(i) \le \frac{N}{T} \le \sum_{i=x}^{MAX} p(i) . \tag{5}$$

All quantities in this inequality are between 0 and 1. The left and right sides are decreasing function of x. Furthermore, for all possible x, the intervals

$$\left[\sum_{i=x+1}^{MAX} p(i) , \sum_{i=x}^{MAX} p(i) \right] \tag{6}$$

are non-overlapping and cover the interval [0,1]. Hence, either there is exactly one integer x for which N/T falls in the interval, or N/T falls on the boundary between two intervals. Consequently, (5) uniquely defines the one optimal value for x, or two adjacent optimal values.

This choice of x is intuitive. As the number of iteration processors approaches the total number, T, of processors, N/T goes to 1 and x tends towards MIN. Thus even if an iteration finishes unexpectedly early, the iteration processors will not be idled. Instead the non-iteration processors (if there are any) will be idled (figure 3a and b).

On the other hand, x will be close to MAX if N is small. In this case, unless the iteration runs through nearly MAX cycles, the iteration processors, of which there are few, will be idled while the non-iteration processors need not be idled (figure 3c and d). In both cases, the processors that are more likely to be idled at run time are the lesser of the iteration or non-iteration processors.

Consider the special case that $N/T = 1/2$. Then from (5),

$$\sum_{i=x+1}^{MAX} p(i) = 1 - \sum_{MIN}^{x} p(i) \le 1/2 \tag{7}$$

which implies that

$$\sum_{MIN}^{x} p(i) \ge 1/2\,. \tag{8}$$

Furthermore,

$$\sum_{i=x}^{MAX} p(i) \ge 1/2\,. \tag{9}$$

Taken together, (8) and (9) imply that x is the *median* of the random variable I (not the mean, as one might expect). In retrospect, this result is obvious because for any random variable I, the value of x that minimizes $E\,|I - x|$ is the median. Note that for a discrete-valued random variable, the median is not always uniquely defined, in that there can be two equally good candidate values. This is precisely the situation

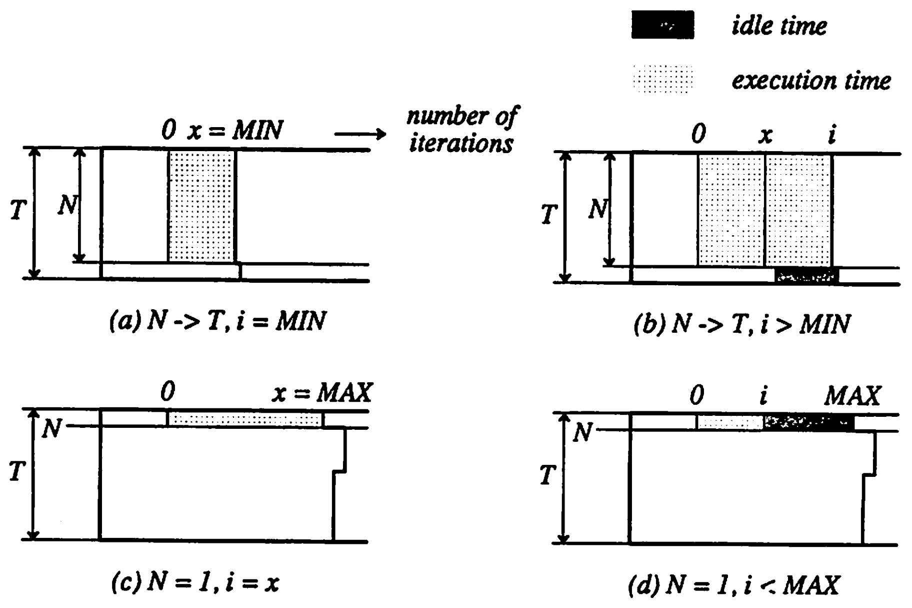


Figure 3. When the number of iteration processors N approaches the total number T, x approaches MIN and the iteration processors will not be idled for any actual number of iterations (a and b). On the other hand, when N is small, x tends toward MAX so that the non-iteration processors will not be idled (c and d).

where x falls on the boundary between two in intervals (6).

Up to now, we have implicitly assumed that the optimal x is an integer, corresponding to an integer number of cycles of the iteration. For non-integer x, the total expected idle time is restated as

$$t(x) = N\tau \sum_{i=MIN}^{\lfloor x \rfloor} p(i)(x-i) + (T-N)\tau \sum_{i=\lfloor x \rfloor +1}^{MAX} p(i)(i-x) . \tag{10}$$

Define $\delta_x = x - \lfloor x \rfloor$, so $0 \le \delta_x < 1$. Then equation (10) becomes

$$\begin{aligned} t(x) &= N\tau \sum_{i=MIN}^{\lfloor x \rfloor} p(i)(\lfloor x \rfloor - i + \delta_x) + (T-N)\tau \sum_{i=\lfloor x \rfloor +1}^{MAX} p(i)(i - \lfloor x \rfloor - \delta_x) \\ &= t(\lfloor x \rfloor) + N\tau \sum_{i=MIN}^{\lfloor x \rfloor} p(i)\delta_x - (T-N)\tau \sum_{i=\lfloor x \rfloor +1}^{MAX} p(i)\delta_x \\ &= t(\lfloor x \rfloor) + \tau\, \delta_x (N - T \sum_{i=\lfloor x \rfloor +1}^{MAX} p(i)) . \end{aligned} \tag{11}$$

This tells us that between $\lfloor x \rfloor$ and $\lfloor x \rfloor + 1$, $t(x)$ is an affine function of δ_x, so it must have its minimum at $\delta_x = 0$ or $\delta_x \to 1$ depending on the sign of the slope. Either of these results is an integer, so inequality (5) is sufficient to find the optimal value of x.

As an example, assume $p(i)$ is a uniform distribution over the range MIN to MAX. In other words,

$$p(i) = \begin{cases} \dfrac{1}{MAX - MIN + 1} & MIN \le i \le MAX \\ 0 & otherwise . \end{cases} \tag{12}$$

Then the optimal number x satisfies

$$\frac{(MAX - x)}{(MAX - MIN + 1)} < \frac{N}{T} < \frac{(MAX - x + 1)}{(MAX - MIN + 1)} , \tag{13}$$

from inequality (5). From this,

$$\begin{aligned} x &> MAX - \frac{N}{T}(MAX - MIN + 1), \text{ and} \\ x &< MAX + 1 - \frac{N}{T}(MAX - MIN + 1) . \end{aligned} \tag{14}$$

Together these imply that

$$x = MAX - \left\lfloor \frac{N}{T}(MAX - MIN + 1) \right\rfloor . \tag{15}$$

In the special case that exactly half of the processors are devoted to the iteration, x

becomes the *expected* number of cycles of the iteration, which for this distribution is the same as the median. Also, as N gets small, x tends toward MAX and as N approaches T, x tends towards MIN, just as expected.

A uniform probability mass function $p(i)$ is not a good model for many types of iteration. In situations involving convergence, a geometric probability mass function may be a better approximation. At each cycle of the iteration, we proceed to the next cycle with probability q and stop with probability $1-q$.

For generality, we still allow an arbitrary minimum number MIN of cycles of iteration. The maximum number, MAX, is infinite. Let $j=i-MIN$, where i is the number of cycles of the iteration. Then, the geometric probability mass function means that for any non-negative integer r,

$$P[j \geq r] = q^r , \tag{16}$$

and

$$P[j = r] = p(r) = q^r(1-q). \tag{17}$$

To use inequality (5), we find

$$\sum_{i=x+1}^{MAX} p(i) = \sum_{i=x+1}^{\infty} p(i) = P[j \geq x+1-MIN] = q^{x+1-MIN}. \tag{18}$$

Similarly,

$$\sum_{i=x}^{MAX} p(i) = q^{x-MIN}. \tag{19}$$

Therefore, from inequality (5), x satisfies

$$\begin{aligned} x+1-MIN &> \log_q \frac{N}{T} \\ x-MIN &> \log_q \frac{N}{T}. \end{aligned} \tag{20}$$

Combining these we get that

$$x = MIN + \left\lfloor \log_q \frac{N}{T} \right\rfloor \tag{21}$$

To gain intuition about this expression, consider the special case where $q = 0.5$ meaning that after each cycle of the iteration we are equally likely to proceed as to stop. Further specializing, when exactly half of the processors are devoted to the iteration, x becomes $MIN+1$, which is the expected number of iteration cycles, as well as the median. Note that practical applications are likely to have a larger value for q, in which case the median will be smaller than the mean.

4. PROCESSOR PARTITIONING

In the previous discussion, we assumed that we can somehow allocate the optimal number N of processors to the data-dependent iteration. Now we give a strategy determining this number. Unfortunately, in practical situations, the detailed structure of the dataflow graph has an impact on the optimal choice of N. To keep the scheduler simple, our preference is to adopt suboptimal policies that are optimal for a subset of graphs and reasonable for the rest. In particular, we can apply a similar principle to that used in section 3. We will discuss the limitations of our method in the next section.

Recall that our scheduling strategy is to assume the iteration runs for x cycles exactly and to construct a static schedule accordingly. When the actual number of cycles differs from x (as it often will), global synchronization is used to idle either the iteration processors (if the iteration finishes early) or the non-iteration processors (if the iteration finishes late). From this, we can conclude that the total cost of the data dependent iteration in quasi-static scheduling is the execution time spent on the iteration plus the idle time caused by it. This is an approximation, as discussed in the next section, because it ignores the effect that the data dependent iteration may have on other computations. Nonetheless, we propose to select N to minimize this cost.

As before, i is the number of iterations, τ_N is the run time per iteration cycle (with N iteration processors), and x_N is the assumed number of iteration cycles from the previous section (with N iteration processors). Note that τ_N and x_N are both non-increasing in N.

If i is smaller that x_N, the iteration processors will be idled and the total cost will be $Nx_N\tau_N$ (case 3, figure 2). On the other hand, if i is greater than x_N, the cost of the iteration consists of execution time on the iteration processors plus idle time on the non-iteration processors. In this case, the total cost becomes $Ni\tau_N + (T-N)(i-x_N)\tau_N$ (case 2, figure 2). As a result, the expected value of the cost of the iteration for a fixed N is

$$t_o(N) = \sum_{i=MIN}^{x_N-1} p(i)Nx_N\tau_N + \sum_{i=x_N}^{MAX} p(i)\,(Ni\tau_N + (T-N)(i-x_N)\tau_N)\,. \tag{22}$$

After a few manipulations, (22) becomes

$$t_o(N) = Nx_N\tau_N + T\tau_N \sum_{i=x_N}^{MAX} p(i)(i-x_N)\,. \tag{23}$$

Our proposal is to minimize this quantity. This can be done for specific distributions $p(i)$.

First, let us consider a geometric distribution on the number of cycles of the iteration. Since

$$\sum_{i=x_N}^{MAX} p(i)(i-x_N) = \frac{q}{1-q} q^{x_N - MIN} , \tag{24}$$

we get

$$t_o(N) = N\tau_N x_N + T\tau_N \frac{q}{1-q} q^{x_N - MIN} . \tag{25}$$

Since both x_N and τ_N are functions of N, dependency of $t_o(N)$ on N can not be clearly defined. If we replace x_N using (21) we get

$$t_o(N) = N\tau_N (MIN + \left\lfloor \log_q \frac{N}{T} \right\rfloor) + T\tau_N \frac{q}{1-q} q^{\left\lfloor \log_q \frac{N}{T} \right\rfloor} , \tag{26}$$

which is a complicated transcendental that looks as if it has to be minimized numerically. Fortunately, we can draw some intuitive conclusions for certain interesting special cases.

Consider the case where linear speedup of the iteration actor is possible. In other words, $\tau_N N = K$, where K is the total amount of computation in one cycle of the iteration. The (26) simplifies slightly to

$$t_o(N) = K(MIN) + K \left\lfloor \log_q \frac{N}{T} \right\rfloor + T \frac{K}{N} \frac{q}{1-q} q^{\left\lfloor \log_q \frac{N}{T} \right\rfloor} . \tag{27}$$

The first term is constant in N and the second term is decreasing in N. We will now show that the third term is approximately constant in N, suggesting that $t_o(N)$ is minimized by selecting the largest possible value, $N = T$. This is intuitively appealing, since with linear speedup applying more processors to the problem would seem to make sense. To show that the third term is approximately constant, note that

$$\frac{N}{qT} = q^{(\log_q \frac{N}{T} - 1)} > q^{\left\lfloor \log_q \frac{N}{T} \right\rfloor} \geq \frac{N}{T} . \tag{28}$$

Consequently, the third term is bounded as follows,

$$\frac{K}{1-q} > T \frac{K}{N} \frac{q}{1-q} q^{\left\lfloor \log_q \frac{N}{T} \right\rfloor} \geq \frac{Kq}{1-q} . \tag{29}$$

These bounds do not depend on N. Note, however, that when $N = T$, this third term is at its minimum, $Kq/(1-q)$. It may also be at this minimum for other values of N, but since the middle term in (26) decreases as N increases, the conclusion is that N should be made as large as possible, namely $N = T$.

Consider another extreme situation, when no speedup of the iteration is possible. In this case, $\tau_N = K$, independent of N. For the third term in (26), we use similar bounding arguments and find that both the upper and lower bounds on the third term increase linearly in N. The first term also increases linearly in N. The second term is

$$N\tau\left\lfloor \log_q \frac{N}{T} \right\rfloor \tag{30}$$

which also increases in N, so the conclusion is that if no speedup is possible, we should use as few processors as possible, or $N = 1$. This is a reassuring conclusion.

For general speedup characteristics, we cannot draw general conclusions. This suggests that a compiler implementing this technique may need to solve (26) numerically for the optimal N. If the total number of processors T is modest, then this task should not be too onerous, although we would certainly prefer to not have to do it. The task can be somewhat simplified, perhaps, by the observation that we can shrink the range of N to be examined by looking into the range of $t_o(N)$. From (20),

$$q^{x_N+1-MIN} \le \frac{N}{T} \le q^{x_N-MIN} \tag{31}$$

which implies that

$$\tau_N N(x_N + \frac{q}{1-q}) \le t_o(N) \le \tau_N N(x_N + \frac{1}{1-q}) . \tag{32}$$

For some values of N, the upper bound is smaller than the lower bound for some other value of N, so we can ignore the latter N's.

As another example, consider the case where $p(i)$ is a uniform distribution. Then equation (23) becomes

$$t_o(N) = N\tau_N x_N + T\tau_N \frac{(MAX - x_N)(MAX - x_N + 1)}{2(MAX - MIN + 1)} . \tag{33}$$

We can replace x_N with the value given by (15). Observe that if we define $R = MAX - MIN + 1$, then

$$\left\lfloor \frac{N}{T}(MAX - MIN + 1) \right\rfloor \approx \frac{N}{T}R , \tag{34}$$

when RN/T is large. This crude approximation simplifies the analysis compared to a bounding argument like that above, which can be carried out and leads to the same conclusion. If in addition we assume linear speedup, so that $N\tau_N = K$, then (33) simplifies to

$$t_o(N) = K(MAX) + \frac{K}{2}\left[1 - \frac{N}{T}R\right] . \tag{35}$$

We see that this function is decreasing linearly in N, suggesting again that we should select the maximum $N = T$.

To summarize, we have derived a general cost function that depends on the speedup attainable for the iteration as more processors are devoted to it. The cost function was given for the special cases when the probability mass function for the number of cycles of the iteration is geometric or uniform. Furthermore, simple special situations lead to intuitive results. Namely, if linear speedup is attainable, then we should devote all the processors to the iteration. If no speedup is possible, then we should devote no more than one processor to the iteration. For more general situations, finding the optimal number of processors requires numerically solving a complicated transcendental.

5. OPTIMALITY

The solution we have given is the optimal solution to a simple, but unrealistic problem. Observe that the makespan of a schedule can be given as follows,

$$\text{makespan} = \frac{1}{T}(IN\tau_N + Y + A) \tag{36}$$

where $IN\tau_N$ is the total computation time devoted to the iteration (lightly shaded in figure 2), Y is the idle time due to our quasi-static synchronization strategy (dark shading in figure 2), and A is the rest of the computation, including all idle time that may result both within the schedule and at the end. Our solution minimizes the makespan under the bold (and unrealistic) assumption that A is independent of our decisions for N and x_N. For fixed N, the first term in (36) is independent of x_N, so our choice of x_N, which minimizes the second term, is optimal. For variable N, our strategy is to minimize the sum of the first two terms.

The key assumption is unreasonable when precedence constraints make A dependent on our choices. Consider for example, the situation where there are more processors than we can effectively use, and the data-dependent iteration is in the critical path for all possible outcomes of I. In this case, it may be helpful to devote more processors to the iteration than the optimal number predicted in section 4. On the other hand, suppose there are no precedence constraints. Then the key assumption is not bad as long as the execution times of all actors are small relative to the makespan. Realistic situations are likely to fall between these two extremes. Perhaps the best solution is to use our policy, but permit the programmer to indicate a different preference through annotation of the program.

6. STATIC ASSIGNMENT AND SELF-TIMED SCHEDULING

Once the number of iteration processors and the assumed number of iteration cycles are decided, we can construct a static schedule accordingly. Quasi-static scheduling means global synchronization that makes the pattern of processor availability after the iteration consistent with the scheduled one, as shown in figure 2. This implies hardware for global synchronization, which may be less expensive than the handshaking required for self-timed execution (a simple wired-or circuit would suffice). However, some idle time compulsorily inserted may be unnecessary in reality. Furthermore, if handshaking is omitted, then the system is intolerant of run-time fluctuations, due for example to interrupts or I/O operations. Hence the quasi-static scheduling strategy is regarded as impractical. Nonetheless, it suggests a good strategy for static-assignment or self-timed scheduling. In static-assignment, we discard all information from the quasi-static schedule except the assignment of actors to processors. In self-timed scheduling, we also use the ordering of actors within processors.

In static-assignment scheduling, actors are assigned to processors without defining the execution order. Unlike dynamic load balancing or techniques that compromise between interprocessor communication cost and load balance, our proposed strategy considers arbitrary precedence relations at compile time. If the actual computation times are similar to those assumed by the scheduler, then our technique can get close to the minimal makespan.

An example of static-assignment scheduling is shown in figure 4. A dataflow program consists of six actors with precedence relationships shown in figure 4a. Actor D represents a data-dependent iteration. Suppose that the program is statically scheduled using our technique, and the resulting assignment puts actors D, C, and F onto the first processor, and the rest onto the second processor. The ordering and timing information is discarded. Assuming D has a data-dependent execution time, the run-time schedule depends on its outcome. Two possible schedules are shown in figure 4b and c. By inspection, we can see in figure 4 that the schedules shown are optimal in the sense of minimizing makespan. However, designing a run-time scheduler that reliably produces these schedules is not easy. Assume that when a processor becomes free, if there is an actor ready to be fired, then the run-time scheduler will fire it. This is not necessarily optimal, but in deterministic processor scheduling it can be shown to be reasonable. Then the only decision to be made by the scheduler occurs when there is more than one actor ready to fire. In figure 4b, the run-time scheduler never faces this decision, so a very simple strategy will yield the schedule shown. In figure 4c, however, after the completion of actor A, the second processor must decide between firing B or E. E is the better choice, but it is not clear at all how the scheduler might know this. An immediate idea is to use some of the static information that was discarded: specifically the ordering information. However, this does not guarantee the right choice, because the static information is based on an assumption about the data-dependent execution time, and the outcome may be far from this. The alternative of stochastic modeling of the program is not very promising either, because only the most grossly oversimplified stochastic models yield to optimization.

The above observations lead to an interesting conclusion. In static-assignment scheduling, the run-time scheduler on each processor faces an ambiguous decision only if more than one of the actors assigned to it are ready to fire when the last actor completes. If this situation arises rarely, then a naive scheduler will work well.

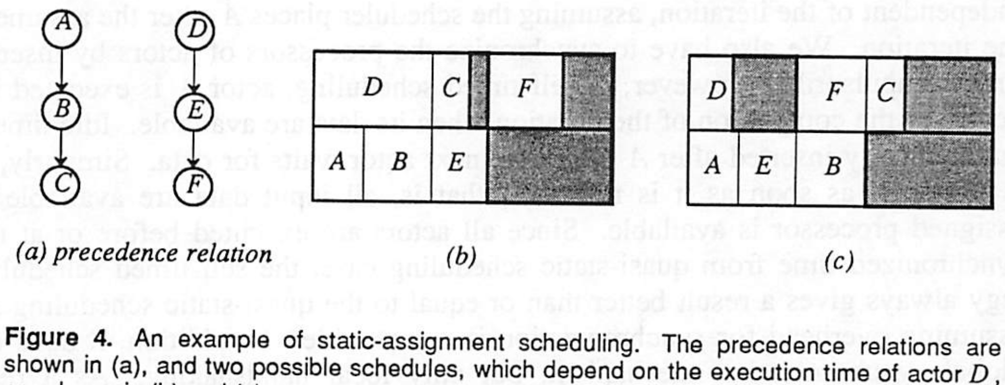


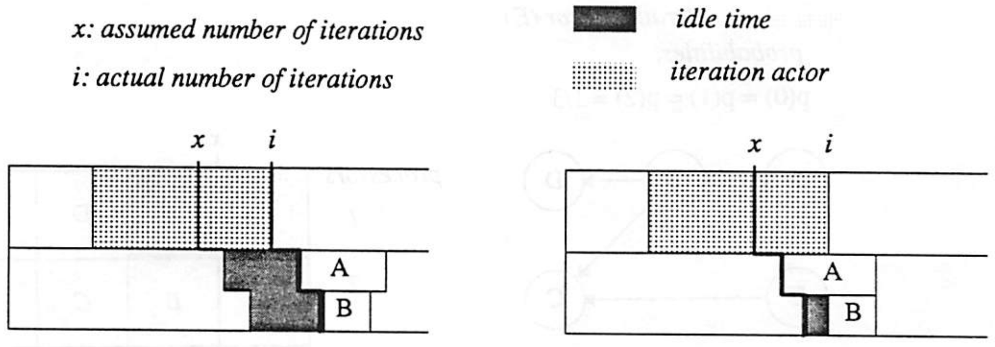
Figure 4. An example of static-assignment scheduling. The precedence relations are shown in (a), and two possible schedules, which depend on the execution time of actor D, are shown in (b) and (c).

However, under the same conditions, a self-timed strategy would work just as well, and the cost would be lower. On the other hand, if the situation arises frequently, then we do not know how to make the decision. Practical proposals are to make the decision arbitrarily, subject to a "fairness" principle, in which no actor will be tried twice before all other actors have been tried [Gao83]. It may be profitable to augment this strategy by using information discarded from the static schedule, but as argued before, this is not guaranteed to lead to an optimal schedule.

A comparison with the Granski, et. al. proposal [Gra87] is in order. In fully dynamic scheduling, assignment is easy, assuming the target architecture is homogeneous. It does not matter which free processor gets an actor, once the decision has been made to fire that actor. So the decisions to be made by the scheduler are simply which of the actors that are ready to be fired should be fired. If the number of actors that are ready to be fired is smaller than the number of available processors, then there is no decision to be made, and the scheduler will not be helped by static information. It is only if the number of ready actors is large that static information can help. In [Gra87] the authors report that the improvement due to using static information in a dynamic scheduler degrades to no improvement for large numbers of processors. We just stated the reason for this.

In self-timed scheduling, we define the execution order of actors at compile time, thus avoiding the difficulty of designing the local controller. In the example of figure 4, suppose that actors are constrained to execute in the order given by figure 4b. In this case, we sacrifice some freedom to optimize at execution time. However, if the variability in execution time is small enough, then there is little justification for paying the run time cost of static-assignment scheduling. Of course, if the explicit token store mechanism of Papadopoulos [Pap88] proves to be truly low cost, then the additional adaptability of static-assignment scheduling makes it more attractive. As pointed out earlier, however, tractable static-assignment scheduling is *not* guaranteed to outperform self-timed. It is easy to construct demonstration examples where, for example, an iteration finishes well before expected, causing an order change that results in a *larger* makespan than if there were no order change.

The difference between quasi-static and self-timed scheduling is shown in figure 5. In quasi-static scheduling, actors A,B are executed after the iteration even if actor A is independent of the iteration, assuming the scheduler places A after the assumed end of the iteration. We also have to synchronize the processors of actors by inserting idle time compulsorily. However, in self-timed scheduling, actor A is executed independently of the completion of the iteration when its data are available. Idle time may be automatically inserted after A while the next actor waits for data. Similarly, actor B is executed as soon as it is runnable; that is, all input data are available and the assigned processor is available. Since all actors are executed before or at the same synchronized time from quasi-static scheduling case, the self-timed scheduling strategy always gives a result better than or equal to the quasi-static scheduling strategy, assuming overhead for synchronization is comparable. In addition, it does not need global synchronization mechanism, but only local handshaking. As a result, we believe that self-timed scheduling is more attractive.


x: assumed number of iterations
i: actual number of iterations
idle time
iteration actor
x
i
A
B
x
i
A
B

(a) quasi-static scheduling *(b) self-timed scheduling*

Figure 5. Comparison between quasi-static scheduling and self-timed scheduling. In quasi-static scheduling, the pattern of processor availability after the iteration is enforced by global synchronization. In self-timed scheduling, the pattern is only enforced if the precedences require it. Here we have assumed that actor B is dependent on the iteration but actor A not.

Self-timed scheduling overcomes a difficulty of quasi-static scheduling illustrated in figure 6. According to the dataflow graph given in figure 6a, the proposed quasi-static schedule is shown in (b). However, suppose the actual number of iterations i exceeds the assumed number x. A strict quasi-static schedule would execute as shown in figure 6c, while a self-timed schedule would execute as shown in (d). In this case, our proposed schedule is no more optimal than that in (d), because we considered only the idle time before the completion of the iteration actor when deciding the value x. In other words, our choice of x only locally optimal. In this example, the idle time after the iteration depends on x. Self-timed execution can sometimes compensate for this deficiency in the scheduling strategy. Idle time immediately after the completion of the iteration has no effect on the performance since there is no compulsory idle time. In other words, for self-timed execution, the schedules in figure 6b and d are equivalent.

This does not lead us to the conclusion that the strategy we propose is optimal under self-timed execution. Consider the two schedules in figure 7, which assume the same precedence graph from figure 6a. Under self-timed scheduling, the schedule in figure 6a is clearly preferable to that in figure 6b, because even if the iteration runs twice as long as the assumed number x, the makespan will not be affected. Our scheduling strategy thus far imposes no constraints that would prefer the schedule in figure 6a. Intuitively, care should be taken to schedule actors after the iteration actor in static-assignment or self-timed scheduling. For examples of this type, the problem can be largely avoided by the following heuristic; all else being equal, actors independent of the iteration should be assigned to the non-iteration processors after the iteration. This heuristic may be easily incorporated in the original static scheduling without significant cost.

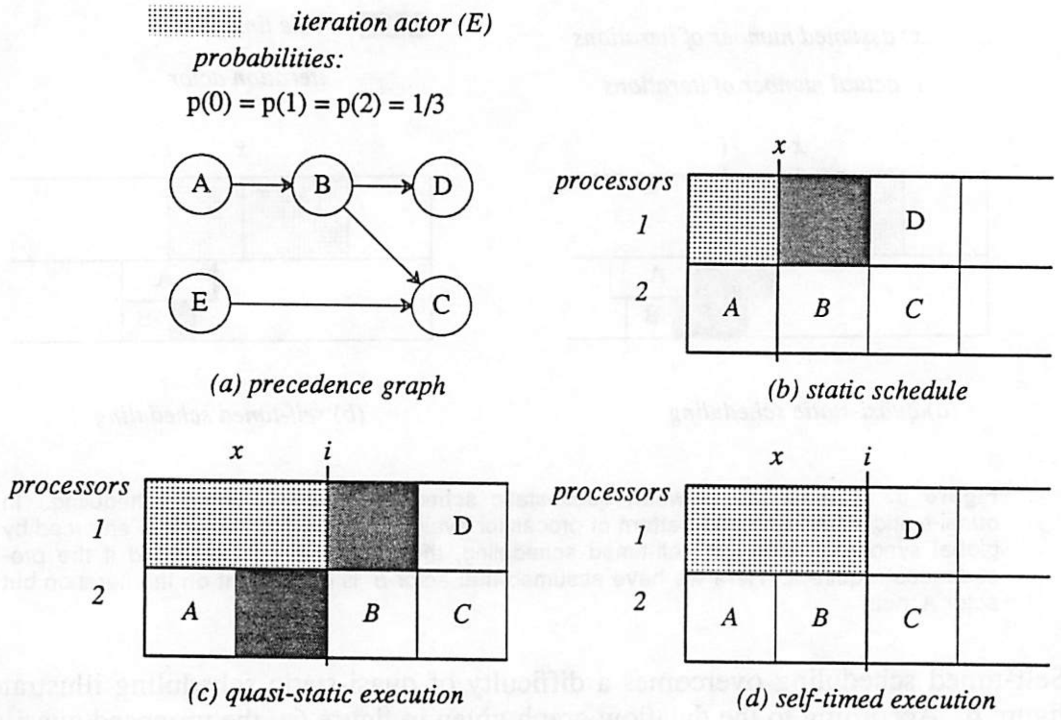


Figure 6. An example showing that a difficulty in quasi-static scheduling is overcome in self-timed scheduling. In the precedence graph shown in (a), assume the iteration actor E is equally likely to run for 0, 1, or 2 iterations of unit length. Assume that one of two processors is to be devoted to the iteration. Then our proposed strategy yields the static schedule in (b). Assume now that a given execution results in the iteration running for two cycles. Static execution of the schedule, in which global synchronization enforces the pattern of processor availability after the iteration, results in the schedule shown in (c), while self-timed execution results in the schedule shown in (d).

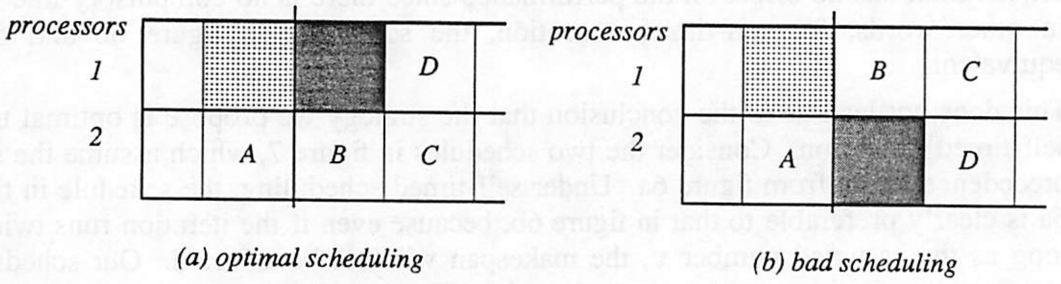


Figure 7. For the same precedence graph as in figure 6, two static schedules with the same makespan are shown. However, if the actual number of iterations turns out to be two, the schedule in (a) is better than that in (b).

7. CONCLUSION

Static-assignment and self-timed scheduling strategies look like the most promising compromises between hardware cost/performance and flexibility. The choice should depend on the amount of data-dependent behavior in the expected applications. Both strategies require compile-time decisions; they require that tasks be assigned to processors at compile time, and in addition, self-timed scheduling requires that the order of execution of the tasks be specified. If there is no data-dependency in the application, then these decisions can be made optimally (or nearly so, to avoid complexity problems). When there is data-dependency, however, optimal or near optimal compile-time strategies become intractable. Most previously proposed solutions include random choices, clustering (to minimize communication overhead), and load balancing. These solutions either ignore precedence relationships in the dataflow graph, or use heuristics based on oversimplified stochastic models. This is justifiable if there is so much data-dependency that the precedence relationships are constantly changing. However, there is a large class of applications, including scientific computations and digital signal processing, where this is not true.

Nearly all applications of parallel computers involve *some* data-dependent behavior. Consequently, there is a clear need for compile-time strategies that can use precedence information in these cases. Quasi-static scheduling strategies have been previously proposed that can handle conditionals and some forms of iteration [Lee89]. The main contribution of this paper is to extend these techniques to handle data-dependent iterations, and to propose that the resulting static schedules give the information needed by a compiler in self-timed and static-assignment situations. The resulting technique can be used to enhace many scheduling algorithms, including those that try to reduce interprocessor communication together with reducing makespan. The proposed method should work well when the amount of data dependency is small, but we admittedly cannot quantify at what level the technique breaks down.

The probability mass function of the number of iteration cycles must be known or estimated at compile time for each iterative construct in the program. Using these probabilities, we find an "assumed" number x of iterations that the scheduler can use to construct a static schedule. This number is selected to minimize the expected idle time on all processors at runtime due to the difference between x and the actual number of iteration cycles executed. This idle time is computed by assuming that the processors are globally synchronized. When half the processors are devoted to the iteration, the resulting choice for x is the *median* number of iterations (not the mean). It is shown that if the execution is self-timed, then the performance can only improve over the quasi-static case, and that the information generated by the quasi-static schedule can be used at very low cost. For static-assignment scheduling, tractable runtime scheduling algorithms may actually lead to *worse* schedules than the quasi-static case, although most of the time the schedules will be better.

REFERENCES

[Ack82]
W. B. Ackerman, "Data Flow Languages", *Computer* , **Vol. 15, No. 2**, pp. 15-25, February, 1982.

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

[Bac78]
J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM* , **Vol. 21, No. 8**, pp. 613-641, August, 1982.

[Bur81]
F. W. Burton and M. R. Sleep, "Executing Functional Programs on A Virtual Tree of Processors", *Proc. ACM Conf. Functional Programming Lang. Comput. Arch.*, pp. 187-194, 1981.

[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.

[Cor79]
M. Cornish, D. W. Hogan, and J. C. Jensen, "The Texas Instruments Distributed Data Processor", *Proc. Louisiana Computer Exposition*, Lafayette, La., March 1979, pp. 189-193.

[Dav78]
A. L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine", *Proc. Fifth Ann. Symp. Computer Architecture*, April, 1978, pp. 210-215.

[Den80]
J. B. Dennis, "Data Flow Supercomputers" *Computer*, **13** (11), November 1980.

[Den88]
J. B. Dennis and G. R. Gao "An Efficient Pipelined Dataflow Processor Architecture" To appear in *Proceedings of the IEEE*, also in the *Proc. ACM SIGARCH Conf. on Supercomputing*, Florida, Nov., 1988.

[Efe82]
K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems", *IEEE Computer*, pp. 50-56, June, 1982.

[Fis84]
J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", *Computer*, July, 1984, **17**(7).

[Gao83]
A Pipelined Code Mapping Scheme for Static Dataflow Computers, PhD dissertation, Laboratory for Computer Science, MIT, Cambridge, MA (1983).

[Gao88]
G. R. Gao, R. Tio, and H. H. J. Hum, "Design of an Efficient Dataflow Architecture without Data Flow", *Proc. Int. Conf. on Fifth Generation Computer Systems*, 1988.

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

[Gir87]
E. F. Girczyc, "Loop Winding - A Data Flow Approach to Functional Pipelining", *ISCAS* , pp. 382-385, 1987.

[Gos82]
K. P. Gostelow, "The U-Interpreter", *Computer* , **Vol. 15, No. 2**, pp. 42-49, February, 1982.

[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.

[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.

[Iqb86]
M. A. Iqbal, J. H. Saltz, and S. H. Bokhari, "A Comparative Analysis of Static and Dynamic Load Balancing Strategies", *Int. Conf. on Parallel Processing*, pp. 1040-1045, 1986.

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

[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).

[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.

[Lee89]
E. A. Lee "Recurrences, Iteration, and Conditionals in Statically Scheduled Data Flow", Memorandum UCB/ERL M89/52, Electronics Research Lab., UC Berkeley, Berkeley CA 94720, May 9, 1989.

[Lee89]
E. A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP" *IEEE Trans. on ASSP*, To Appear (1989).

[Lei83]
C. E. Leiserson, "Optimizing Synchronous Circuitry by Retiming", *Third Caltech Conference on VLSI*, Pasadena, CA, March, 1983.

[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.

[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.

[Muh87]
H. Muhlenbeim, M. Gorges-Schleuter, and O. Kramer, "New Solutions to the Mapping Problem of Parallel Systems : The Evolution Approach", *Parallel Computing*, **4**, pp. 269-279, 1987.

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

[Pla76]
A. Plas, *et. al.*, "LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment", *Proc. 1976 Int. Conf. Parallel Processing*, pp. 293-302.

[Sch86]
D. A. Schwartz and T. P. Barnwell, III, "Cyclo-Static Solutions: Optimal Multiprocessor Realizations of Recursive Algorithms", *VLSI Signal Processing*, IEEE Press (1986).

[Sri86]
V. P. Srini, "An Architectural Comparison of Dataflow Systems", *Computer*, pp. 68-88, March, 1986.

[Veg84]
S. R. Vegdahl, " A Survey of Proposed Architectures for the Execution of Functional Languages", *IEEE Trans. Computers*, **Vol c-33, No. 12**, pp. 1050 - 1071, December, 1984.

[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.