Compile-Time Scheduling and Assignment of Data-Flow Program Graphs with Data-Dependent Iteration

Soonhoi Ha, Student Member, IEEE, and Edward A. Lee, Member, IEEE

Abstract—Scheduling of data-flow graphs onto parallel processors consists in 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 1) toward 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 data-flow 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 caused by 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. The method is illustrated with a practical programming example, yielding preliminary results that are very promising.

Index Terms—Data flow, data-dependent iteration, parallel processors, parallelizing compilers, quasi-static scheduling, scheduling.

I. INTRODUCTION

A data-flow representation is suitable for programming multiprocessors because parallelism can be extracted automatically from the representation [1], [2]. Each node, or actor, in a data-flow 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 data-flow 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 data-flow graph. The proposed techniques are valid both for fine-grain and large-grain.

Scheduling of parallel computations consists in 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 when a particular operation is done, 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 boundaries between these categories are 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 data-flow architecture [10], the LAU system, from the Department of Computer Science, ONERA/CERT, France [35], and the DDM1 [9]. 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 data-flow machines [23]. A machine that has a mixture of fully dynamic and static assignment scheduling is the Manchester data-flow machine [39]. 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 data-flow machines [37]. Data-flow machines evaluate data-flow 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 [2] for TI's data-driven processor (DDP) executes Fortran programs that are translated into data-flow graphs by a compiler [8] using static assignment. Another example (targeted at digital signal processing) is the NEC UPD7281 [5]. The cost of implementing tagged-token architectures has
recently been dramatically reduced using an "explicit token store" [34]. Another example of an architecture that assumes static assignment is the proposed "argument-fetching data-flow architecture" [15], which is based on the argument-fetching data-driven principle of Dennis and Gao [11].

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 [19] 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 [24]. Another automated example is a data-flow programming system for digital signal processing called Gabriel that targets multiprocessor systems made with programmable DSP's [28]. 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 [13] 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 1 to strategy 4, 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 [25], [26]. 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 [27]. 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 [25], [26]. 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 [33], [6], [40], [31], [12], [30]. 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 [22], [4], [21]. 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 [7].

Fully dynamic scheduling is most able to utilize the resources and fully exploit the concurrency of a data-flow representation of an algorithm. However it requires too much hardware and/or software run-time overhead. For instance, the MIT static data-flow machine [10] proposes an expensive broad-band 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 [18].

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.

In the next section, we set the context by explaining what we mean by data-dependent iteration, and explaining precisely the problem we are solving. A complete compiler using this solution also needs other scheduling techniques from the literature, as explained. In Section III, we introduce the notion of "assumed" execution time for a data-dependent iteration. Once the scheduler "assumes" an execution time for the iteration, it can construct a static schedule containing the iteration. The question addressed in this section is, what should the assumed execution time be? The answer is not the expected execution time, as one might expect. Instead, the answer depends on the number of processors devoted to the iteration relative to the total number of processors. In Section IV we explain how to decide how many processors to devote to the iteration. Section V describes the technique applied to a real programming example from graphics. Section VI explains precisely why the proposed technique is not optimal, except under unrealistic assumptions, but also why we should expect good performance. Through Section VI we have assumed "quasi-static" execution, which requires global synchronization of the processors. In Section VII, we show that the method applies much more broadly to self-timed and static-assignment scheduling.
II. DATA-DEPENDENT ITERATION

In data-dependent iteration, the number of iteration cycles is determined at run time and cannot be known at compile time. Two possible data-flow representations for data-dependent iteration are shown in Fig. 1 [27]. The numbers adjacent to the arcs indicate the number of tokens produced or consumed when an actor fires [25]. In Fig. 1(a), since the up-sample 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 up-sample actor. In Fig. 1(b), 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 $f(\cdot)$ 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 the "switch" actor, again until the test function $f(\cdot)$ 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 data-flow graph. In Fig. 1(a), the "iteration body" actor consists of the up-sample, data-dependent iteration, and down-sample actors. The data-dependent iteration actor may consist of a subgraph of arbitrary complexity and may itself contain data-dependent iterations. In Fig. 1(b), 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 run time, 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 data-flow graph.

The method given in this paper can be applied to both kinds of iteration in Fig. 1 identically. There is, however, an important difference between them. In Fig. 1(b), each cycle of the iteration depends on the previous cycle. There is a recurrence that prevents simultaneous execution of successive cycles. In Fig. 1(a), there is no such restriction, unless the iteration body itself contains a recurrence. For the purposes of this paper, we will simply assume that successive cycles of the iteration must be executed sequentially. An extension that handles overlapped cycles will be reported in a separate paper.

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 noniteration 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 run time will actually be random, the scheduler will assume a carefully chosen deterministic run time and construct the schedule accordingly. The assumed run time is chosen so that the expected total idle time caused by the difference between the assumed and actual run times is minimal. It is well known that locally minimizing idle time fails 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 is 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 data-flow graph onto a set of processors [25]. 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 retiming.
or loop winding\textsuperscript{1} [29], [17]. 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 cyclostatic scheduling [36]. The proposal below applies regardless of the method used, but in all our illustrations we assume blocked scheduling. We similarly avoid specifics about how the scheduling of the overall data-flow graph is performed. Our method is consistent with simple heuristic regardless of the method used, but in all our illustrations or to reduce interprocessor communication costs. Broadly, our about how the scheduling of the overall data-flow graph is scheduling algorithms, such as Hu-level scheduling [20], 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.

III. 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 [32], 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 run time, some processors will be idled as a consequence. Our strategy is to find the assumed run time 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 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 $\text{MIN}$ and the maximum by $\text{MAX}$. $\text{MAX}$ 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 noniteration 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 $\tau$, 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 equal to, greater than, or less than $x$. These cases are displayed in Fig. 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 Fig. 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 noniteration 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.

Consider the case where the assumed number $x$ is exactly correct. Then no idle time exists on any processor (Fig. 2(a)). Otherwise, the noniteration processors will be idled if the iteration takes more than $x$ cycles (Fig. 2(b)), or else the iteration processors will be idled (Fig. 2(c)). 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 $\text{MIN}$ and $\text{MAX}$. For a fixed assumed $x$ the expected idle time $t_1(x)$ on the iteration processors is

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

(1)

The expected idle time $t_2(x)$ on the noniteration processors is

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

(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

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

(3)

Similarly,

$$t(x) - t(x - 1) = N\tau - T\tau \sum_{i = \text{MIN}}^{x} p(i).$$

(4)

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

$$\sum_{i = x + 1}^{\text{MAX}} p(i) \leq \frac{N}{T} \leq \sum_{i = \text{MIN}}^{\text{MAX}} p(i).$$

(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

$$\sum_{i = i + 1}^{\text{MAX}} p(i) \leq \sum_{i = \text{MIN}}^{\text{MAX}} p(i).$$

(6)

\textsuperscript{1}As a possibly interesting side issue, it does not appear to have been pointed out in the literature that retiming is simply a data-flow perspective on loop winding, so the techniques are in fact equivalent.
are nonoverlapping 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 \(\text{MIN}\). Thus even if an iteration finishes unexpectedly early, the iteration processors will not be idled. Instead the noniteration processors (if there are any) will be idled (Fig. 3(a) and (b)). On the other hand, when \(N\) is small, \(x\) tends toward \(\text{MAX}\) so that the noniteration processors will not be idled ((c) and (d)). In both cases, the processors that are more likely to be idled at run time are the lesser of the iteration or noniteration processors.

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

\[
\sum_{i=\lfloor x \rfloor + 1}^{\text{MAX}} p(i) = 1 - \sum_{i=\text{MIN}}^{\lfloor x \rfloor} p(i) \leq 1/2
\]

which implies that

\[
\sum_{i=\text{MIN}}^{\lfloor x \rfloor} p(i) \geq 1/2.
\]

Furthermore,

\[
\sum_{i=\lfloor x \rfloor}^{\text{MAX}} p(i) \geq 1/2. \tag{9}
\]

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 where \(x\) falls on the boundary between two intervals in (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 noninteger \(x\), the total expected idle time is restated as

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

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

\[
t(x) = N \tau \sum_{i=\text{MIN}}^{\lfloor x \rfloor} p(i)(\lfloor x \rfloor - i + \delta_x).
\]
To use inequality (5), we find

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

$$= q^{x+1-MIN}. \tag{18}$$

Similarly,

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

Therefore, from inequality (5), $x$ satisfies

$$x+1-MIN > \log_q \frac{N}{T} \tag{20}$$

$$x-MIN < \log_q \frac{N}{T}. \tag{20}$$

Combining these, we get that

$$x = MIN + \left\lceil \log_q \frac{N}{T} \right\rceil. \tag{21}$$

To gain intuition about this expression, consider the special case where $q = 0.5$, meaning that after each cycle of iteration we are as 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.

The expressions for $x$, the assumed number of iteration cycles, are simple enough to be of practical use in a parallelizing compiler that assumes a geometric or uniform probability mass function. However, there remains the question of determining how many processors to devote to an iteration.

### IV. 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 data-flow 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 principle similar to that used in Section III. We will discuss the limitations of our method in Section VI.

Recall that our scheduling strategy is to assume that 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 noniteration 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, $\tau_N$ 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 $\tau_N$ and $x_N$ are both nonincreasing in $N$.

If $i$ is smaller than $x_N$, the iteration processors will be idled and the total cost will be $N \cdot x_N \cdot \tau_N$ (Fig. 2(c)). 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 noniteration processors. In this case, the total cost becomes $N \cdot \tau_N + (T - N)(i - x_N) \cdot \tau_N$ (Fig. 2(b)). As a result, the expected value of the cost of the iteration for a fixed $N$ is

$$ t_o(N) = \sum_{i=x_N}^{x_N-1} \frac{q}{i} (N x_N \tau_N + \sum_{i=x_N}^{\max} p(i) (i - x_N)). \qquad (22) $$

After a few manipulations, (22) becomes

$$ t_o(N) = N x_N \tau_N + \sum_{i=x_N}^{\max} p(i) (i - x_N). \qquad (23) $$

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

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

$$ \sum_{i=x_N}^{\max} p(i) (i - x_N) = \frac{q}{1 - q} \cdot q^{x_N - x_N}, \qquad (24) $$

we get

$$ t_o(N) = N x_N \tau_N + \frac{q}{1 - q} \cdot q^{x_N - x_N}. \qquad (25) $$

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

$$ t_o(N) = N \tau_N \left( \min + \frac{\log_q N}{T} \right) + \frac{q}{1 - q} \cdot q^{x_N - x_N}. \qquad (26) $$

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, $\tau_N = K$, where $K$ is the total amount of computation in one cycle of the iteration. The (26) simplifies slightly to

$$ t_o(N) = K (\min + K \left( \frac{\log_q N}{T} \right)) + \frac{q}{1 - q} \cdot q^{x_N - x_N}. \qquad (27) $$

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

$$ \frac{N}{qT} > q^{\log_q N} \cdot q \geq N \cdot T. \qquad (28) $$

Consequently, the third term is bounded as follows:

$$ \frac{K}{1 - q} > T \frac{q}{N} \frac{q}{1 - q} \cdot q^{\log_q N} \geq \frac{Kq}{1 - q}. \qquad (29) $$

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 upper and lower bounds on the third term increase linearly in $N$. The first term also increases linearly in $N$. The second term is

$$ N \tau \left[ \log_q \frac{N}{T} \right] \qquad (30) $$

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 not to 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),

$$ q^{x_N} + 1 \leq \frac{N}{T} \leq q^{x_N - 1} \qquad (31) $$

which implies that

$$ \tau_N \left( x_N + \frac{q}{1 - q} \right) \leq t_o(N) \leq \tau_N \left( x_N + 1 \right). \qquad (32) $$

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 (23) becomes

$$ t_o(N) = N \tau_N x_N + \frac{\tau_N}{2} \frac{\max - x_N + 1}{2}. \qquad (33) $$

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

$$ \frac{N}{T} (\max - \min + 1) \approx \frac{N}{T} R \qquad (34) $$
when $RN/T$ is large. This crude approximation simplifies the analysis compared with a bounding argument such as 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

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

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 where 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.

V. AN EXAMPLE

We can illustrate our method with an application from graphics, in which a geometric shape is displayed and rotated in three dimensions, with perspective. This is an attractive application because the program is simple and can be written with or without iteration, and the iteration can be data-dependent or not. We can compare quite a variety of realizations. Not surprisingly, we find that using data-dependent iteration considerably decreases the total amount of computation compared with programs that avoid data-dependent iteration. Furthermore, when we use data-dependent iteration, our scheduling method results in a program that is only 3% slower than the best that can be expected from dynamic scheduling, for this example. We are comparing against a hopelessly optimistic model of dynamic scheduling, so with most of the computation is in these transformations, the cost is high. For a particular test shape (a blocklettered G), we determined that the implementation that avoided iteration required an average of 2581 instruction cycles (on four processors) to draw one line. This is the first entry in Fig. 5. Programs using iteration are much less expensive.

In Fig. 4(b), the execution time of each actor is given in Motorola 56000 instruction cycles (currently 75 ns). These are not ideal implementations of these actors, but they are working implementations in the Gabriel system. Suppose that we have four processors. Then, we may assign $1 \leq N \leq 4$ processors to the data-dependent iteration actor. To make the best decision on how many processors are assigned to the actor, we can construct the global schedule (Fig. 4(d)). This Gantt chart shows the number of assigned processors, as explained in Section IV. For each $N$, we calculate $x_N$ and the corresponding $t_o(N)$ assuming a given probability mass function of the length of line segment. The cost of iteration, shown in Fig. 4(c), and the schedule, shown in Fig. 4(d), assume a geometric distribution with $MIN = 0$ and $q = 0.95$. In the actual scheduling process, the number $N$ greater than 3 is not considered at all since the makespan of the subgraph within the iteration actor is not shortened with more than two processors. Thus, the search space for $N$ can often be reduced significantly. From the numbers in Fig. 4(c), we choose to assign $N = 2$ processors to the iteration. After the decision is made, we can construct the global schedule (Fig. 4(d)). This Gantt chart shows the assumed length of the iteration as a shaded region.

To make the program more parallelizable, we retimed the graph in front of the data-dependent iteration actor. This is perfectly reasonable for this application, and can be automated [29]. With the specific example we used, we achieved reasonably high processor utilization (82.6%) and low makespan (429 cycles). Of course, at execution time, the number of cycles of the iteration will vary, so the performance will vary. Since there is idle time right at the end of noniteration processors owing to the iteration actor, from the reasoning in Section VI
below, we expect that the schedule is not optimal. However, it is certainly near optimal in this case.

The major question that remains unanswered is how to determine the stochastic model that best fits an iteration. Our choice here of a geometric model with $q = 0.95$ (the probability of continuing is 0.95) is probably not very accurate. We applied the program to simple geometric shape (letter “G”) in order to compare the run-time performance with several scheduling decisions (see Fig. 5). The performance is measured by the average number of cycles to draw one line and depends on the specific shape being drawn. As discussed earlier, fully static scheduling without iteration gives the worst result. Another method that can use fully static scheduling is to perform the maximum number of iteration cycles every time, which gives the second worst result, as shown in the second row of Fig. 5. Next we approximate the run-time statistics using geometric distribution with two different parameters: $q = 0.9$ and $q = 0.95$. The first value grossly underestimates the average length of the lines drawn. The second value results is a probability mass function with the appropriate mean but the wrong shape. For the fifth experiment shown in Fig. 5, we use exactly the correct probability mass function, computed by histograming the lengths of the lines in the geometric shape being drawn. In the sixth experiment, we calculate the performance for fully-dynamic scheduling, ignoring overhead.

The results are remarkable. Using the exact probability mass function, we are within 3% of the best that can be expected from fully dynamic scheduling, for this program (fifth line, Fig. 5). Using a function with the right mean but the wrong shape, the result is identical (fourth line). Using a function with the wrong mean and the wrong shape, we are still within 12% of the best that can be expected from fully dynamic scheduling (third line).

These results are particularly promising because we are making comparisons with a fully dynamic scheduling strategy that is far more sophisticated than what would be practical, and we are ignoring the scheduling overhead. Specifically, we assume the dynamic scheduler somehow knows how many cycles of the iteration will be executed before each cycle of the overall program begins. It then uses a critical path method (Hu-level scheduling) to construct a schedule for this number of cycles. Since practical dynamic scheduling algorithms are much more primitive, we view the performance of this algorithm as a bound on the performance of all dynamic schedulers. When we count the run-time overhead, the fully dynamic scheduling will be abandoned without hesitation for this example.

The promising results for this program should be viewed only as promising results based on one program. We are developing a programming environment that will permit much more extensive experimentation with practical programs; only after those experiments are complete will we know just how general this method is. Nonetheless, the experiments we have done show that with a good stochastic model for the iteration,
at least some programs will get schedules that are about as good as can be expected in practice. They also show that the scheduling method depends on the validity of the stochastic model for the iteration. However, we make the very preliminary postulate that the performance of the technique will not be highly sensitive to the stochastic model since even a crude model might give a near-optimal number for the iteration cycles. This can only be verified by trying many examples, something that requires first developing much more infrastructure. Should the sensitivity prove to be greater, we can then envision successive refinements of the schedule based on observations of the executing program.

VI. 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:

$$\text{makespan} = \frac{1}{t} (I + \tau_N + Y + A) \quad (36)$$

where $I$ is the total computation time devoted to the iteration (lightly shaded in Fig. 2), $Y$ is the idle time caused by our quasi-static synchronization strategy (dark shading in Fig. 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, i.e., $A$ is independent of $I$. For fixed $N$, the first term in (36) is independent of $x_N$, and 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 IV. 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.

VII. 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 Fig. 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, caused for example by interrupts of 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. First we use the techniques of the previous sections to construct a quasi-static schedule. To get a self-timed execution, we insert handshaking at run time and ignore the firing times dictated by the quasi-static schedule. To get static-assignment execution, we discard all information from the quasi-static schedule except the assignment of actors to processors.

A. Static-Assignment Scheduling

In static-assignment scheduling, actors are assigned to processors without defining the execution order. In contrast to 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 Fig. 6. A data-flow program consists of six actors with precedence relationships shown in Fig. 6(a). 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 Fig. 6(b) and (c). By inspection, we can see in Fig. 6 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 Fig. 6(b), the run-time scheduler never faces this...
HA AND LEE: COMPILE-TIME SCHEDULING AND ASSIGNMENT OF DATA-FLOW PROGRAM GRAPHS 1235

Fig. 6. 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).

decision, so a very simple strategy will yield the schedule shown. In Fig. 6(c), 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. 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 [4]. 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 [18] 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 [18] the authors report that the improvement brought about by the use of static information in a dynamic scheduler degrades to no improvement for large numbers of processors. We just stated the reason for this.

B. Self-Timed Scheduling

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 Fig. 6, suppose that actors are constrained to execute in the order given by Fig. 6(b). 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 [34] 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 Fig. 7. In quasi-static scheduling, actors \( A \) and \( B \) 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 the 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 a global synchronization mechanism, but only local handshaking. As a result, we believe that self-timed scheduling is more attractive.

Self-timed scheduling overcomes a difficulty of quasi-static scheduling illustrated in Fig. 8. In the precedence graph shown in Fig. 8(a), assume the iteration actor \( E \) is equally likely to run for 0, 1, or 2 iterations of unit length and one of two processors is to be devoted to the iteration. Then our proposed strategy yields the quasi-static schedule in (b). However, suppose the actual number of iterations \( i \) exceeds the assumed number \( x \). A strict quasi-static schedule, in which global synchronization enforces the pattern of processor availability after the iteration, would execute as shown in Fig. 8(c) 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 \( x \) is 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 Fig. 8(b) and (d) are equivalent.

This does not lead us to the conclusion that the strategy we propose is optimal under self-timed execution. Consider
Fig. 7. Comparison between (a) quasi-static scheduling and (b) 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.

Fig. 8. An example showing that a difficulty in quasi-static scheduling is overcome in self-timed scheduling. According to the precedence graph in (a), the proposed quasi-static schedule is shown in (b). Assume now that the actual number of iterations is 2. Static execution of the schedule results in the schedule shown in (c), while self-timed execution results in the schedule shown in (d).

Fig. 9. For the same precedence graph as in Fig. 8, two static schedules ((a) optimal scheduling and (b) bad scheduling) with the same makespan are shown. However, if the actual number of iterations turns out to be 2, the schedule in (a) is better than that in (b).

the two schedules in Fig. 9, which assume the same precedence graph from Fig. 8(a). Under self-timed scheduling, the schedule in Fig. 9(a) is clearly preferable to that in Fig. 9(b), 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 Fig. 9(a). 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 noniteration processors after the iteration. This heuristic may be easily incorporated in the original static scheduling without significant cost.

**VIII. 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; 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 data-flow 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 previously been proposed that can handle conditionals and some forms of iteration [27]. 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 enhance 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 successive cycle of an iteration and to support conditionals. This number is selected so that the information generated by the quasi-static schedule could be as little as 3% slower than an ideal (and highly unrealistic) fully dynamic schedule. This is because the performance can only improve over the quasi-static case, although most of the potential practical impact, but they are very promising.

The technique is illustrated using one programming example. These results are only a very preliminary indication of the potential practical impact, but they are very promising. For this one program, we found that the resulting quasi-static schedule could be as little as 3% slower than an ideal (and highly unrealistic) fully dynamic schedule. This performance depends on a reasonable (but not exact) stochastic model of the iteration, assumed by the compiler. For the particular program we selected, the performance does not degrade rapidly as the stochastic model gets further from actual program behavior, suggesting that a compiler can use fairly simple techniques to estimate the model. We are developing a programming environment that will permit much more extensive experimentation with the technique.

We are also currently working on natural extensions to the methods described here and will report on these in a subsequent paper. These extensions include the ability to overlap successive cycles of an iteration and to support conditionals and recursion using the same principles.

REFERENCES


Soonhoi Ha (S'91) received the B.S. (honors) and M.S. degrees in electrical and electronics engineering from Seoul National University, Seoul, Korea, in 1985 and 1987, respectively. He is currently a Ph.D. degree candidate in the Electrical Engineering and Computer Sciences Department at the University of California at Berkeley. His research interests include parallel computation, multiprocessor scheduling algorithms, and mixed paradigm simulation.

Edward A. Lee (S'80–M'86) received the B.S. degree from the Yale University in 1979, the S.M. degree from the Massachusetts Institute of Technology in 1981, and the Ph.D. degree from the University of California at Berkeley in 1986.

From 1979 to 1982 he was a member of technical staff at Bell Telephone Laboratories, Holmdel, NJ, in the Advanced Data Communications Laboratory. From 1979 to 1982 he was a member of technical staff at Bell Telephone Laboratories, Holmdel, NJ, in the Advanced Data Communications Laboratory, where he did extensive work with early programmable DSP's and exploratory work on voice-band data modem techniques and simultaneous voice and data transmission. He is now an associate professor in the Electrical Engineering and Computer Sciences Department at the University of California at Berkeley. His research activities include parallel computation, architecture and software techniques for programmable DSP's, design environments for the development of real-time software, and digital communication.

Dr. Lee was a recipient of a 1987 NSF Presidential Young Investigator award, and IBM faculty development award, the 1986 Sakrison prize at U.C. Berkeley for the best thesis in electrical engineering, and a paper award from the IEEE Signal Processing Society. He is coauthor of Digital Communication (with D. G. Messerschmitt, Kluwer Academic, 1988) and Digital Signal Processing Experiments (with Alan Kamas, Prentice Hall, 1989) and has published numerous technical papers.