Compile-Time Scheduling of Dynamic Constructs in Dataflow Program Graphs

Soonhoi Ha, *Member, IEEE* and Edward A. Lee, *Fellow, IEEE*

Abstract—**Scheduling dataflow graphs onto processors consists of assigning actors to processors, ordering their execution within the processors, and specifying their firing time. While all scheduling decisions can be made at runtime, the overhead is excessive for most real systems. To reduce this overhead, compile-time decisions can be made for assigning and/or ordering actors on processors. Compile-time decisions are based on known *profiles* available for each actor at compile time. The profile of an actor such as the execution time and the communication patterns. However, a dynamic construct within a macro actor, such as a conditional and a data-dependent iteration, makes the profile of the actor unpredictable at compile time. For those constructs, we propose to assume some profile at compile-time and define a cost to be minimized when deciding on the profile under the assumption that the runtime statistics are available at compile-time. Our decisions on the profiles of dynamic constructs are shown to be optimal under some bold assumptions, and expected to be near-optimal in most cases. The proposed scheduling technique has been implemented as one of the rapid prototyping facilities in Ptolemy. This paper presents the preliminary results on the performance with synthetic examples.**

Keywords— **multiprocessor scheduling, dynamic constructs, profile, macro actor, dataflow program graphs.**

I. Introduction

A Dataflow graph representation, either as a programming language or as an intermediate representation during compilation, is suitable for programming multiprocessors because parallelism can be extracted automatically from the representation [1], [2] Each node, or actor, in a dataflow graph represents either an individual program instruction or a group thereof to be executed according to the precedence constraints represented by arcs, which also represent the flow of data. A dataflow graph is usually made hierarchical. In a hierarchical graph, an actor itself may represent another dataflow graph: it is called a *macro* actor.

Particularly, we define a *data-dependent* macro actor, or data-dependent actor, as a macro actor of which the execution sequence of the internal dataflow graph is data dependent (cannot be predicted at compile time). Some examples are macro actors that contain dynamic constructs such as *conditional*, *data-dependent iteration*, and *recursion*. Actors are said to be data-independent if not data-dependent.

The scheduling task consists of assigning actors to processors, specifying the order in which actors are executed on each processor, and specifying the time at which they are executed. These tasks can be performed either at compile time or at run time [3]. In the *fully-dynamic* scheduling, all scheduling decisions are made at run time. It has the flexibility to balance the computational load of processors in response to changing conditions in the program. In case a program has a large amount of non-deterministic behavior, any static assignment of actors may result in very poor load balancing or poor scheduling performance. Then, the fully dynamic scheduling would be desirable. However, the run-time overhead may be excessive; for example it may be necessary to monitor the computational loads of processors and ship the program code between processors via networks at run time. Furthermore, it is not usually practical to make globally optimal scheduling decision at run time.

In this paper, we focus on the applications with a moderate amount of non-deterministic behavior such as DSP applications and graphics applications. Then, the more scheduling decisions are made at compile time the better in order to reduce the implementation costs and to make it possible to reliably meet any timing constraints.

While compile-time processor scheduling has a very rich and distinguished history [4], [5], most efforts have been focused on deterministic models: the execution time of each actor T_i on a processor P_k is fixed and there are no data-dependent actors in the program graph. Even in this restricted domain of applications, algorithms that accomplish an optimal scheduling have combinatorial complexity, except in certain trivial cases. Therefore, good heuristic methods have been developed over the years [4], [6], [7], [8]. Also, most of the scheduling techniques are applied to a completely expanded dataflow graph and assume that an actor is assigned to a processor as an indivisible unit. It is simpler, however, to treat a data-dependent actor as a schedulable indivisible unit. Regarding a macro actor as a schedulable unit greatly simplifies the scheduling task. Prasanna *et al* [9] schedule the macro dataflow graphs hierarchically to treat macro actors of matrix operations as schedulable units. Then, a macro actor may be assigned to more than one processor. Therefore, new scheduling techniques to treat a macro actor as a schedulable unit was devised.

Compile-time scheduling assumes that static information about each actor is known. We define the *profile* of an actor as the static information about the actor necessary for a given scheduling technique. For example, if we use a list scheduling technique, the profile of an actor is simply the computation time of the actor on a processor. The communication requirements of an actor with other actors are included in the profile if the scheduling tech-

S. Ha is with the Department of Computer Engineering, Seoul National University, Seoul, 151-742, Korea. e-mail: sha@comp.snu.ac.kr
E. Lee is with the Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA 94720, USA. e-mail: eal@ohm.eecs.berkeley.edu

nique requires that information. The profile of a macro actor would be the number of the assigned processors and the local schedule of the actor on the assigned processors. For a data-independent macro actor such as a matrix operation, the profile is deterministic. However, the profile of a data-dependent actor of dynamic construct cannot be determined at compile time since the execution sequence of the internal dataflow subgraph varies at run time. For those constructs, we have to assume the profiles somehow at compile-time.

The main purpose of this paper is to show how we can define the profiles of dynamic constructs at compile-time. A crucial assumption we rely on is that we can approximate the runtime statistics of the dynamic behavior at compile-time. Simulation may be a proper method to gather these statistics if the program is to be run on an embedded DSP system. Sometimes, the runtime statistics could be given by the programmer for graphics applications or scientific applications.

By optimally choosing the profile of the dynamic constructs, we will minimize the expected schedule length of a program assuming the quasi-static scheduling. In figure 1, actor A is a data-dependent actor. The scheduling result is shown with a gantt chart, in which the horizontal axis indicates the scheduling time and the vertical axis indicates the processors. At compile time, the profile of actor A is assumed. At run time, the schedule length of the program varies depending on the actual behavior of actor A. Note that the pattern of processor availability before actor B starts execution is preserved at run time by inserting idle time. Then, after actor A is executed, the remaining static schedule can be followed. This scheduling strategy is called *quasi-static* scheduling that was first proposed by Lee [10] for DSP applications. The strict application of the quasi-static scheduling requires that the synchronization between actors is guaranteed at compile time so that no run-time synchronization is necessary as long as the pattern of processor availability is consistent with the scheduled one. It is generally impractical to assume that the exact run-time behaviors of actors are known at compile time. Therefore, synchronization between actors is usually performed at run time. In this case, it is not necessary to enforce the pattern of processor availability by inserting idle time. Instead, idle time will be inserted when synchronization is required to execute actors. When the execution order of the actors is not changed from the scheduled order, the actual schedule length obtained from run-time synchronization is proven to be not much different from what the quasi-static scheduling would produce [3]. Hence, our optimality criterion for the profile of dynamic constructs is based on the quasi-static scheduling strategy, which makes analysis simpler.

II. Previous Work

All of the deterministic scheduling heuristics assume that static information about the actors is known. But almost none have addressed how to define the static information of data-dependent actors. The pioneering work on this issue was done by Martin and Estrin [11]. They calculated the mean path length from each actor to a dummy terminal actor as the level of the actor for list scheduling. For example, if there are two possible paths divided by a conditional construct from an actor to the dummy terminal actor, the level of the actor is a sum of the path lengths weighted by the probability with which the path is taken. Thus, the levels of actors are based on statistical distribution of dynamic behavior of data-dependent actors. Since this is expensive to compute, the mean execution times instead are usually used as the static information of data-dependent actors [12]. Even though the mean execution time seems a reasonable choice, it is by no means optimal. In addition, both approaches have the common drawback that a data-dependent actor is assigned to a single processor, which is a severe limitation for a multiprocessor system.

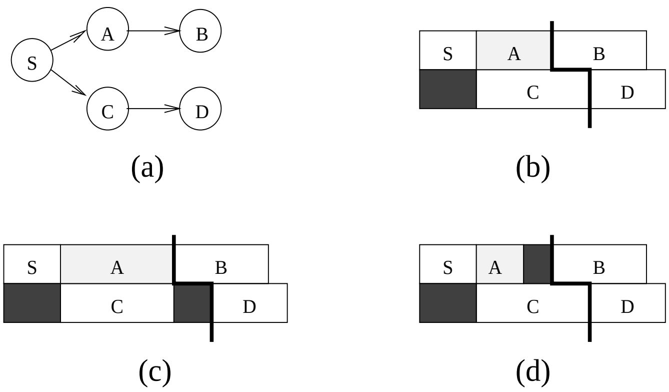

Fig. 1. (a) A dataflow graph consists of five actors among which actor A is a data-dependent actor. (b) Gantt chart for compile-time scheduling assuming a certain execution time for actor A. (c) At run time, if actor A takes longer, the second processor is padded with no-ops and (d) if actor A takes less, the first processor is idled to make the pattern of processor availability same as the scheduled one (dark line) in the quasi-static scheduling.

Two groups of researchers have proposed quasi-static scheduling techniques independently: Lee [10] and Loeffler *et al* [13]. They developed methods to schedule conditional and data-dependent iteration constructs respectively. Both approaches allow more than one processor to be assigned to dynamic constructs. Figure 2 shows a conditional and compares three scheduling methods. In figure 2 (b), the local schedules of both branches are shown, where two branches are scheduled on three processors ($N = 3$) while the total number of processors is 4 ($T = 4$).

In Lee's method, we overlap the local schedules of both branches and choose the maximum termination for each processor. For hard real-time systems, it is the proper choice. Otherwise, it may be inefficient if either one branch is more likely to be taken and the size of the likely branch is much smaller. On the other hand, Loeffler takes the local schedule of more likely branch as the profile of the conditional. This strategy is inefficient if both branches are equally likely to be taken and the size of the assumed branch is much larger. Finally, a conditional *evaluation* can be replaced with a conditional *assignment* to make the construct static; the graph is modified as illustrated in figure (c). In this scheme, both true and false branches are sched-

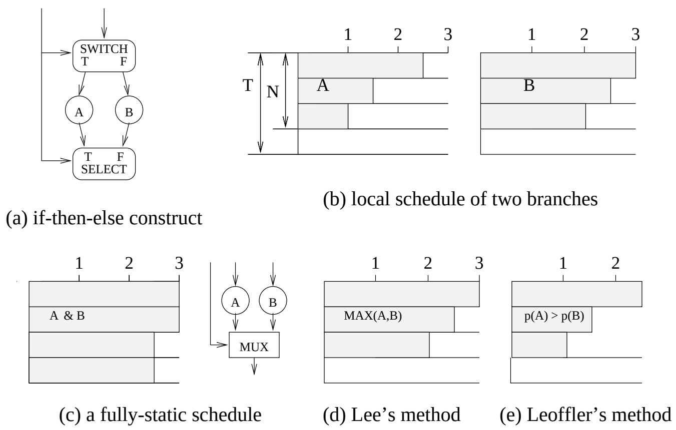

Fig. 2. Three different schedules of a conditional construct. (a) An example of a conditional construct that forms a data-dependent actor as a whole. (b) Local deterministic schedules of the two branches. (c) A static schedule by modifying the graph to use conditional assignment. (d) Lee's method to overlap the local schedules of both branches and to choose the maximum for each processor. (e) Loeffler's method to take the local schedule of the branch which is more likely to be executed.

uled and the result from one branch is selected depending on the control boolean. An immediate drawback is inefficiency which becomes severe when one of the two branches is not a small actor. Another problem occurs when the unselected branch generates an exception condition such as divide-by-zero error. All these methods on conditionals are ad-hoc and not appropriate as a general solution.

Quasi-static scheduling is very effective for a data-dependent iteration construct if the construct can make effective use of all processors in each cycle of the iteration. It schedules one iteration and pads with no-ops to make the pattern of processor availability at the termination the same as the pattern of the start (figure 3) (Equivalently, all processors are occupied for the same amount of time in each iteration). Then, the pattern of processor availability after the iteration construct is independent of the number of iteration cycles. This scheme breaks down if the construct cannot utilize all processors effectively.

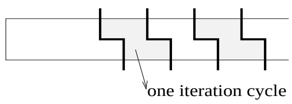

Fig. 3. A quasi-static scheduling of a data-dependent iteration construct. The pattern of processor availability is independent of the number of iteration cycles.

The recursion construct has not yet been treated successfully in any statically scheduled data flow paradigm. Recently, a proper representation of the recursion construct has been proposed [14]. But, it is not explained how to schedule the recursion construct onto multiprocessors. With finite resources, careless exploitation of the parallelism of the recursion construct may cause the system to deadlock.

In summary, dynamic constructs such as conditionals, data-dependent iterations, and recursions, have not been treated properly in past scheduling efforts, either for static scheduling or dynamic scheduling. Some ad-hoc methods have been introduced but proven unsuitable as general solutions. Our earlier result with data-dependent iteration [3] demonstrated that a systematic approach to determine the profile of data-dependent iteration actor could minimize the expected schedule length. In this paper, we extend our analysis to general dynamic constructs.

In the next section, we will show how dynamic constructs are assigned their profiles at compile-time. We also prove the given profiles are optimal under some unrealistic assumptions. Our experiments enable us to expect that our decisions are near-optimal in most cases. Section 4,5 and 6 contains an example with *data-dependent iteration, recursion, and conditionals* respectively to show how the profiles of dynamic constructs can be determined with known runtime statistics. We implement our technique in the Ptolemy framework [15]. The preliminary simulation results will be discussed in section 7. Finally, we discuss the limits of our method and mention the future work.

III. Compile-Time Profile of Dynamic Constructs

Each actor should be assigned its compile-time profile for static scheduling. Assuming a quasi-static scheduling strategy, the proposed scheme is to decide the profile of a construct so that the average schedule length is minimized assuming that all actors except the dynamic construct are data-independent. This objective is not suitable for a hard real-time system as it does not bound the worst case behavior. We also assume that all dynamic constructs are uncorrelated. With this assumption, we may isolate the effect of each dynamic construct on the schedule length separately. In case there are inter-dependent actors, we may group those actors as another macro actor, and decide the optimal profile of the large actor. Even though the decision of the profile of the new macro actor would be complicated in this case, the approach is still valid. For nested dynamic constructs, we apply the proposed scheme from the inner dynamic construct first. For simplicity, all examples in this paper will have only one dynamic construct in the dataflow graph.

The *run-time cost* of an actor i, C_i, is the sum of the total computation time devoted to the actor and the idle time due to the quasi-static scheduling strategy over all processors. In figure 1, the run-time cost of a data-dependent actor A is the sum of the lightly (computation time) and darkly shaded areas after actor A or C (immediate idle time after the dynamic construct). The schedule length of a certain iteration can be written as

$$schedule\ \ length = \frac{1}{T}(C_i + R), \tag{1}$$

where T is the total number of processors in the system, and R is the rest of the computation including all idle time that may result both within the schedule and at the end. Therefore, we can minimize the expected schedule length by minimizing the expected cost of the data-dependent actor or dynamic construct if we assume that R is independent of our decisions for the profile of actor i. This assumption is unreasonable when precedence constraints make R dependent on our choice of profile. Consider, for example, a situation where the dynamic construct is always on the critical path and there are more processors than we can effectively use. Then, our decision on the profile of the construct will directly affect the idle time at the end of the schedule, which is included in R. On the other hand, if there is enough parallelism to make effective use of the unassigned processors and the execution times of all actors are small relative to the schedule length, the assumption is valid. Realistic situations are likely to fall between these two extremes.

To select the optimal compile-time profile of actor i, we assume that the statistics of the runtime behavior is known at compile-time. The validity of this assumption varies to large extent depending on the application. In digital signal processing applications where a given program is repeatedly executed with data stream, simulation can be useful to obtain the necessary information. In general, however, we may use a well-known distribution, for example uniform or geometric distribution, which makes the analysis simple. Using the statistical information, we choose the profile to give the least expected cost at runtime as the compile-time profile.

The profile of a data-dependent actor is a local schedule which determines the number of assigned processors and computation times taken on the assigned processors. The overall algorithm of profile decision is as follows. We assume that the dynamic behavior of actor i is expressed with parameter k and its distribution $p(k)$.

```
min = LARGE;
// T is the total number of processors.
// N is the number of processors assigned to the actor.
for N = 1 to T {
        // A(N,k) is the actor cost with parameter N, k
        // p(k) is the probability of parameter k
        temp = Σ_k p(k)A(N, k);
        if (temp < min) min = temp;
}
```

In the next section, we will illustrate the proposed scheme with *data-dependent iteration, recursion, and conditionals* respectively to show how profiles are decided with runtime statistics.

IV. Data Dependent Iteration

In a data-dependent iteration, the number of iteration cycles is determined at runtime and cannot be known at compile-time. Two possible dataflow representations for data-dependent iteration are shown in figure 4 [10].

The numbers adjacent to the arcs indicate the number of tokens produced or consumed when an actor fires [2]. In figure 4 (a), since the upsample actor produces M tokens each time it fires, and the iteration body consumes only one token when it fires, the iteration body must fire M times for each firing of the upsample actor. In figure 4 (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 function $t()$ indicates that the iteration is finished by producing a token with value "false". During the iteration, the output of the iteration function $f()$ will be routed around by the "switch" actor, again until the test function $t()$ produces a token with value "false". There are many variations on these two basic models for data-dependent iteration.

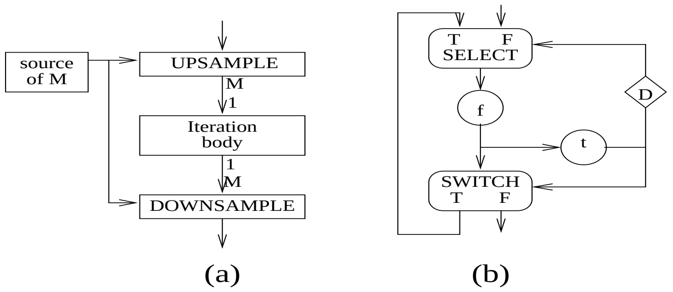

Fig. 4. Data-dependent iteration can be represented using the either of the dataflow graphs shown. The graph in (a) is used when the number of iterations is known prior to the commencement of the iteration, and (b) is used otherwise.

The previous work [3] considered a subset of data-dependent iterations, in which simultaneous execution of successive cycles is prohibited as in figure 4 (b). In figure 4 (a), there is no such restriction, unless the iteration body itself contains a recurrence. Therefore, we generalize the previous method to permit overlapped cycles when successive iteration cycles are invokable before the completion of an iteration cycle. Detection of the intercycle dependency from a sequential language is the main task of the parallel compiler to maximize the parallelism. A dataflow representation, however, reveals the dependency rather easily with the presence of delay on a feedback arc.

We assume that the probability distribution of the number of iteration cycles is known or can be approximated at compile time. Let the number of iteration cycles be a random variable I with known probability mass function $p(i)$. For simplicity, we set the minimum possible value of I to be 0. We let the number of assigned processors be N and the total number of processors be T. We assume a blocked schedule as the local schedule of the iteration body to remove the unnecessary complexity in all illustrations, although the proposed technique can be applicable to the overlap execution schedule [16]. In a blocked schedule, all assigned processors are assumed to be available, or synchronized at the beginning. Thus, the execution time of one iteration cycle with N assigned processors is t_N as displayed in figure 5 (a). We denote by s_N the time that must elapse in one iteration before the next iteration is enabled. This time could be zero, if there is no data dependency between iterations. Given the local schedule of one iteration cycle, we decide on the assumed number of iteration cycles, x_N, and the number of overlapped cycles k_N. Once the two parameters, x_N and k_N, are chosen, the profile of the data-dependent iteration actor is determined as shown in figure 5 (b). The subscript N of t_N, s_N, x_N and k_N represents that they are functions of N, the number of the assigned processors. For brevity, we will omit the subscript N for the variables without confusion. Using this profile of the data-dependent macro actor, global scheduling is performed to make a hierarchical compilation. Note that the pattern of processor availability after execution of the construct is different from that before execution. We do not address how to schedule the iteration body in this paper since it is the standard problem of static scheduling.

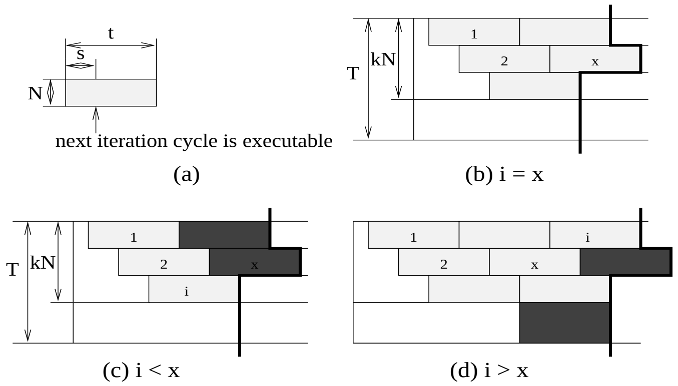

Fig. 5. (a) A blocked schedule of one iteration cycle of a data-dependent iteration actor. A quasi-static schedule is constructed using a fixed assumed number x of cycles in the iteration. The cost of the actor is the sum of the dotted area (execution time) and the dark area (idle time due to the iteration). There displays 3 possible cases depending on the actual number of cycles i in (b) for $i = x$, (c) for $i < x$, and (d) for $i > x$.

According to the quasi-static scheduling policy, three cases can happen at runtime. If the actual number of cycles coincides with the assumed number of iteration cycles, the iteration actor causes no idle time and the cost of the actor consists only of the execution time of the actor. Otherwise, some of the assigned processors will be idled if the iteration takes fewer than x cycles (figure 5 (c)), or else the other processors as well will be idled (figure 5 (d)). The expected cost of the iteration actor, $C(N, k, x)$, is a sum of the individual costs weighted by the probability mass function of the number of iteration cycles. The expected cost becomes

$$C(N,k,x) = \sum_{i=0}^{x} p(i)Ntx \ + \sum_{i=x+1}^{\infty} p(i)(Ntx \ + \ Tt\lceil\frac{i-x}{k}\rceil). \tag{2}$$

By combining the first term with the first element of the second term, this reduces to

$$C(N,k,x) = Ntx \ + \ Tt \sum_{i=x+1}^{\infty} p(i)\lceil\frac{i-x}{k}\rceil. \tag{3}$$

Our method is to choose three parameters (N, k, and x) in order to minimize the expected cost in equation (3). First, we assume that N is fixed. Since $C(N, k, x)$ is a decreasing function of k with fixed N, we select the maximum possible number for k. The number k is bounded by two ratios: $\frac{T}{N}$ and $\frac{t}{s}$. The latter constraint is necessary to avoid any idle time between iteration cycles on a processor. As a result, k is set to be

$$k = min(\lfloor\frac{T}{N}\rfloor, \lfloor\frac{t}{s}\rfloor). \tag{4}$$

The next step is to determine the optimal x. If a value x is optimal, the expected cost is not decreased if we vary x by $+1$ or -1. Therefore, we obtain the following inequalities,

$$\begin{aligned} C(N,k,x) \quad - \quad & C(N,k,x+1) = \\ & -Nt + Tt\sum_{i=0}^{\infty} p(x+1+ik) \le 0 \\ C(N,k,x) \quad - \quad & C(N,k,x-1) = \\ & Nt - Tt\sum_{i=0}^{\infty} p(x+ik) \le 0. \end{aligned} \tag{5}$$

Since t is positive, from inequality (5),

$$\sum_{i=0}^{\infty} p(x+1+ik) \le \frac{N}{T} \le \sum_{i=0}^{\infty} p(x+ik). \tag{6}$$

If k is equal to 1, the above inequality becomes same as inequality (5) in [3], which shows that the previous work is a special case of this more general method.

Up to now, we decided the optimal value for x and k for a given number N. How to choose optimal N is the next question we have to answer. Since t is not a simple function of N, no closed form for N minimizing $C(N, k, x)$ exists, unfortunately. However, we may use exhaustive search through all possible values N and select the value minimizing the cost in polynomial time. Moreover, our experiments show that the search space for N is often reduced significantly using some criteria.

Our experiments show that the method is relatively insensitive to the approximated probability mass function for i [3]. Using some well-known distributions which have nice mathematical properties for the approximation, we can reduce the summation terms in (3) and (6) to closed forms. Let us consider a geometric probability mass function with parameter q as the approximated distribution of the number of iteration cycles. This models a class of asymmetric bell-shaped distributions. The geometric probability mass function means that for any non-negative integer r,

$$P[j \ge r] = q^r, \ P[j = r] = p(r) = q^r(1-q). \tag{7}$$

To use inequality (6), we find

$$\sum_{i=0}^{\infty} p(x+ik) = (1-q)\frac{q^x}{1-q^k}. \tag{8}$$

Therefore, from the inequality (6), the optimal value of x satisfies

$$\frac{1-q}{1-q^k}q^{x+1} \le \frac{N}{T} \le \frac{1-q}{1-q^k}q^x. \quad (9)$$

Using floor notation, we can obtain the closed form for the optimal value as follows:

$$x = \lfloor log_q \frac{N(1-q^k)}{T(1-q)} \rfloor. \quad (10)$$

Furthermore, equation (3) is simplified by using the fact

$$\sum_{i=x+1}^{\infty} p(i) \lceil \frac{i-x}{k} \rceil = \frac{q^{x+1}}{1-q^k}, \quad (11)$$

getting

$$C(N,k,x) = Ntx + Tt\frac{q^{x+1}}{1-q^k}. \quad (12)$$

Now, we have all simplified formulas for the optimal profile of the iteration actor. Similar simplification is possible also with uniform distributions [17]. If k equals to 1, our results coincide with the previous result reported in [3].

V. Recursion

Recursion is a construct which instantiates itself as a part of the computation if some termination condition is not satisfied. Most high level programming languages support this construct since it makes a program compact and easy to understand. However, the number of self-instantiations, called the *depth of recursion*, is usually not known at compile-time since the termination condition is calculated at run-time. In the dataflow paradigm, recursion can be represented as a macro actor that contains a **SELF** actor (figure 6). A **SELF** actor simply represents an instance of a subgraph within which it sits.

If the recursion actor has only one **SELF** actor, the function of the actor can be identically represented by a data-dependent iteration actor as shown in figure 4 (b) in the previous section. This includes as a special case all tail recursive constructs. Accordingly, the scheduling decision for the recursion actor will be same as that of the translated data-dependent iteration actor. In a generalized recursion construct, we may have more than one **SELF** actor. The number of **SELF** actors in a recursion construct is called the *width* of the recursion. In most real applications, the width of the recursion is no more than two. A recursion construct with width 2 and depth 2 is illustrated in figure 6 (b) and (c). We assume that all nodes of the same depth in the computation tree have the same termination condition. We will discuss the limitation of this assumption later. We also assume that the run-time probability mass function of the depth of the recursion is known or can be approximated at compile-time.

The potential parallelism of the computation tree of a generalized recursion construct may be huge, since all nodes at the same depth can be executed concurrently. The maximum degree of parallelism, however, is usually not known at compile-time. When we exploit the parallelism of the construct, we should consider the resource limitations. We may have to restrict the parallelism in order not to deadlock the system. Restricting the parallelism in case the maximum degree of parallelism is too large has been recognized as a difficult problem to be solved in a dynamic dataflow system. Our approach proposes an efficient solution by taking the degree of parallelism as an additional component of the profile of the recursion construct.

Suppose that the width of the recursion construct is k. Let the depth of the recursion be a random variable I with known probability mass function $p(i)$. We denote the *degree* of parallelism by d, which means that the descendents at depth d in the computation graph are assigned to different processor groups. A descendent recursion construct at depth d is called a *ground* construct (figure 7 (a)). If we denote the size of each processor group by N, the total number of processors devoted to the recursion becomes Nk^d. Then, the profile of a recursion construct is defined by three parameters: the assumed depth of recursion x, the degree of parallelism d, and the size of a processor group N. Our approach optimizes the parameters to minimize the expected cost of the recursion construct. An example of the profile of a recursion construct is displayed in figure 7 (b).

Let τ be the sum of the execution times of actors **a**,**c**, and **h** in figure 6. And, let τ_o be the sum of the execution times of actors **a** and **b**. Then, the schedule length l_x of a ground construct becomes

$$l_x = \tau(k^0+k^1+\cdots+k^{x-d-1})+\tau_o k^{x-d} = \tau\frac{k^{x-d}-1}{k-1}+\tau_o k^{x-d}, \quad (13)$$

when $x \ge d$. At run time, some processors will be idled if the actual depth of recursion is different from the assumed depth of recursion, which is illustrated in figure 7 (c) and (d). When the actual depth of recursion i is smaller than the assumed depth x, the assigned processors are idled. Otherwise, the other processors as well are idled. Let R be the sum of the execution times of the recursion besides the ground constructs. This basic cost R is equal to $\frac{N\tau(k^d-1)}{k-1}$.

For $i \le x$, the runtime cost, C_1, becomes

$$C_1 = R + Nk^d(\tau\frac{k^{x-d}-1}{k-1} + \tau_o k^{x-d}), \quad (14)$$

assuming that x is not less than d. For $i > x$, the cost C_2 becomes

$$\begin{aligned} C_2 &= R + Nk^d(\tau\frac{k^{i-d}-1}{k-1} + \tau_o k^{i-d}) + \\ &\quad (T - Nk^d)(\frac{\tau}{k-1} + \tau_o)(k^{i-d} - k^{x-d}). \quad (15)\end{aligned}$$

Therefore, the expected cost of the recursion construct, $C(N,x,d)$ is the sum of the run-time cost weighted by the probability mass function.

$$C(N,d,x) = R + \sum_{i=0}^{x} p(i)C_1 + \sum_{i=x+1}^{\infty} p(i)C_2. \quad (16)$$

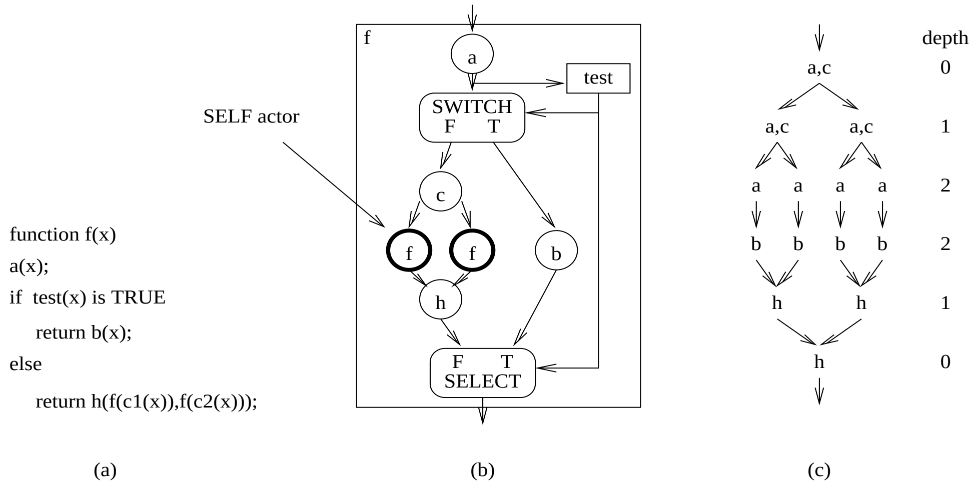

Fig. 6. (a) An example of a recursion construct and (b) its dataflow representation. The **SELF** actor represents the recursive call. (c) The computation tree of a recursion construct with two **SELF** actors when the depth of the recursion is two.

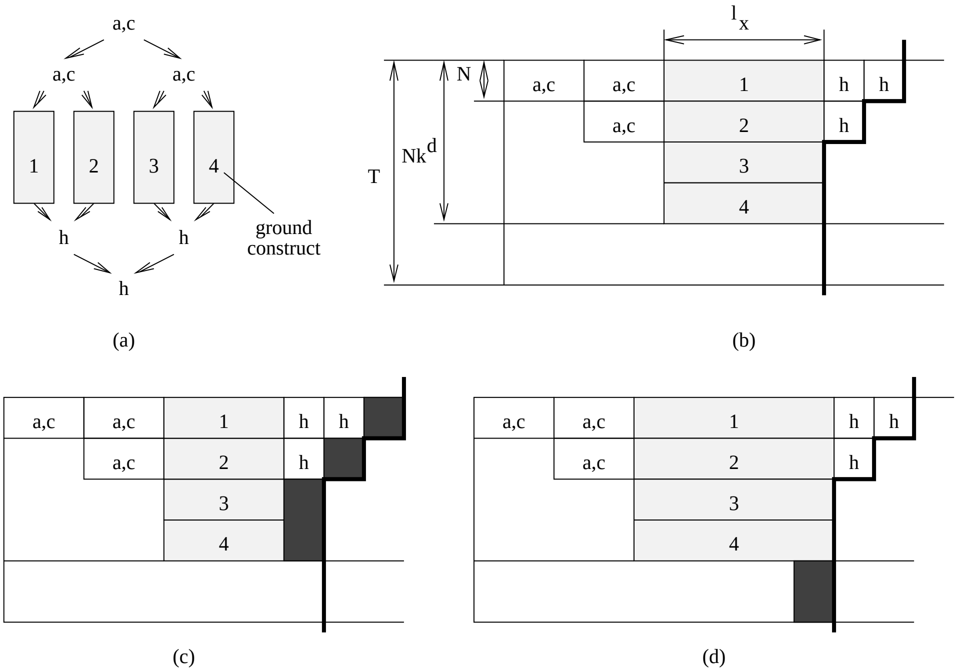

Fig. 7. (a) The reduced computation graph of a recursion construct of width 2 when the degree of parallelism is 2. (b) The profile of the recursion construct. The schedule length of the ground construct is a function of the assumed depth of recursion x and the degree of parallelism d. A quasi-static schedule is constructed depending on the actual depth i of the recursion in (c) for $i < x$ and in (d) for $i > x$.

After a few manipulations,

$$\begin{aligned} C(N,d,x) &= N(\tau\frac{k^x-1}{k-1} + \tau_o k^x) + \\ & \sum_{i=x+1}^{\infty} p(i) T(\frac{\tau}{k-1} + \tau_o)(k^{i-d} - k^{x-d}) \end{aligned} \quad (17)$$

First, we assume that N is fixed. Since the expected cost is a decreasing function of d, we select the maximum possible number for d. The number d is bounded by the processor constraint: $Nk^d \leq T$. Since we assume that the assumed depth of recursion x is greater than the degree of parallelism d, the optimal value for d is

$$d = min(\lfloor log_k \frac{T}{N} \rfloor, x). \quad (18)$$

Next, we decide the optimal value for x from the observation that if x is optimal, the expected cost is not decreased when x is varied by $+1$ and -1. Therefore, we get

$$\begin{aligned} & C(N,d,x) - C(N,d,x+1) = \\ & \quad (\tau + \tau_o(k-1))(-Nk^x + Tk^{x-d} \sum_{i=x+1}^{\infty} p(i)) \leq 0 \\ & C(N,d,x) - C(N,d,x-1) = \end{aligned}$$

$$(\tau + \tau_o(k-1))(Nk^{x-1} - Tk^{x-d-1}\sum_{i=x}^{\infty} p(i)) \leq 0. \quad (19)$$

Rearranging the inequalities, we get the following,

$$\sum_{i=x+1}^{\infty} p(i) \leq \frac{Nk^d}{T} \leq \sum_{i=x}^{\infty} p(i). \quad (20)$$

Note the similarity of inequality (20) with that for data-dependent iterations (6). In particular, if k is 1, the two formulas are equivalent as expected. The optimal values d and x depend on each other as shown in (18) and (20). We may need to use iterative computations to obtain the optimal values of d and x starting from $d = \lfloor log_k \frac{T}{N} \rfloor$.

Let us consider an example in which the probability mass function for the depth of the recursion is geometric with parameter q. At each execution of depth i of the recursion, we proceed to depth $i+1$ with probability q and return to depth $i-1$ with probability $1-q$. From the inequality (20), the optimal x satisfies

$$q^{x+1} \leq \frac{Nk^d}{T} \leq q^x. \quad (21)$$

As a result, x becomes

$$x = \lfloor \log_q \frac{Nk^d}{T} \rfloor. \quad (22)$$

Up to now, we assume that N is fixed. Since τ is a transcendental function of N, the dependency of the expected cost upon the size of a processor group N is not clear. Instead, we examine the all possible values for N, calculate the expected cost from equation (3) and choose the optimal N giving the minimum cost. The complexity of this procedure is still polynomial and usually reduced significantly since the search space of N can be reduced by some criteria. In case of geometric distribution for the depth of the recursion, the expected cost is simplified to

$$C(N,d,x) = N(\tau\frac{k^x-1}{k-1} + \tau_o k^x) + T(\tau + \tau_o(k-1))k^{x-d}\frac{q^{x+1}}{1-qk}. \quad (23)$$

In case the number of child functions is one ($k=1$), our simplified formulas with a geometric distribution coincide with those for data-dependent iterations, except for an overhead term to detect the loop termination.

Recall that our analysis is based on the assumption that all nodes of the same depth in the computation tree have the same termination condition. This assumption roughly approximates a more realistic assumption, which we call the *independence* assumption, that all nodes of the same depth have equal probability of terminating the recursion, and that they are independent each other. This equal probability is considered as the probability that all nodes of the same depth terminate the recursion in our assumption. The expected number of nodes at a certain depth is the same in both assumptions even though they describe different behaviors. Under the independence assumption, the shape of the profile would be the same as shown in figure 7: the degree of parallelism d is maximized. Moreover, all recursion processors have the same schedule length for the ground constructs. However, the optimal schedule length l_x of the ground construct would be different. The length l_x is proportional to the number of executions of the recursion constructs inside a ground construct. This number can be any integer under the independence assumptions, while it belongs to a subset $\{0, k^1, k^2, ...\}$ under our assumption. Since the probability mass function for this number is likely to be too complicated under the independence assumption, we sacrifice performance by choosing a sub-optimal schedule length under a simpler assumption.

VI. Conditionals

Decision making capability is an indispensable requirement of a programming language for general applications, and even for signal processing applications. A dataflow representation for an if-then-else and the local schedules of both branches are shown in figure 2 (a) and (b).

We assume that the probability p_1 with which the "TRUE" branch (branch 1) is selected is known. The "FALSE" branch (branch 2) is selected with probability $p_2 = 1 - p_1$. Let t_{ij} be the finishing time of the local schedule of the i-th branch on the j-th processor. And let $\hat{t}_j$ be the finishing time on the j-th processor in the optimal profile of the conditional construct. We determine the optimal values $\{\hat{t}_j\}$ to minimize the expected runtime cost of the construct. When the i-th branch is selected, the cost becomes

$$\sum_{k=1}^{N} \hat{t}_k + T \max[0, \max_{j\in[1,N]}(t_{ij} - \hat{t}_j)]. \quad (24)$$

Therefore, the expected cost $C(N)$ is

$$C(N) = \sum_{k=1}^{N} \hat{t}_k + T\sum_{i=1}^{2} p_i \max[0, \max_{j\in[1,N]}(t_{ij} - \hat{t}_j)]. \quad (25)$$

It is not feasible to obtain the closed form solutions for $\hat{t}_j$ because the max function is non-linear and discontinuous. Instead, a numerical algorithm is developed.

1. Initially, take the maximum finish time of both branch schedules for each processor according to Lee's method [10].
2. Define $\alpha_i = \max[0, \max_j(t_{ij} - \hat{t}_j), 0)]$. Initially, all $\alpha_i = 0$. The variable α_i represents the excessive cost per processor over the expected cost when branch i is selected at run time. We define the *bottleneck* processors of branch i as the processors $\{j\}$ that satisfy the relation $t_{ij} - \hat{t}_j = \alpha_i$. For all branches $\{i\}$, repeat the next step.
3. Choose the set of bottleneck processors, Θ_i, of branch i only. If we decrease $\hat{t}_j$ by δ for all $j \in \Theta_i$, the variation of the expected cost becomes $\Delta C(N) = (-|\Theta_i| + Tp_i)\delta$. Increase δ until the set Θ_i needs to be updated. Update Θ_i and repeat step 3.

Now, we consider the example shown in figure 2. Suppose $p_1 = 0.3$ and $p_2 = 0.7$. The initial profile in our algorithm is same as Lee's profile as shown in figure 8 (a), which happens to be same as Loeffler's profile in this specific example. The optimal profile determined by our algorithm is displayed in figure 8 (b).

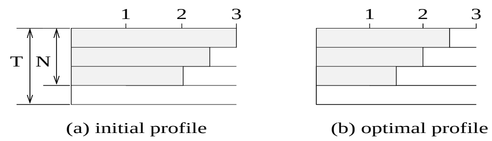

Fig. 8. Generation of the optimal profile for the conditional construct in figure 2. (a) initial profile (b) optimal profile.

We generalized the proposed algorithm to the M-way branch construct by *case* construct. To realize an M-way branch, we prefer to using *case* construct to using a nested if-then-else constructs. Generalization of the proposed algorithm and proof of optimality is beyond the scope of this paper. For the detailed discussion, refer to [17]. For a given number of assigned processors, the proposed algorithm determines the optimal profile. To obtain the optimal number of assigned processors, we compute the total expected cost for each number and choose the minimum.

VII. An Example

The proposed technique to schedule data-dependent actors has been implemented in **Ptolemy**, which is a heterogeneous simulation and prototyping environment being developed in U.C.Berkeley, U.S.A. [15]. One of the key objectives of Ptolemy is to allow many different computational models to coexist in the same system. A *domain* is a set of blocks that obey a common computational model. An example of mixed domain system is shown in figure 9. The synchronous dataflow (SDF) domain contains all data-independent actors and performs compile-time scheduling. Two branches of the conditional constructs are represented as SDF subsystems, so their local schedules are generated by a static scheduler. Using the local schedules of both branches, the dynamic dataflow(DDF) domain executes the proposed algorithm to obtain the optimal profile of the conditional construct. The topmost SDF domain system regards the DDF domain as a macro actor with the assumed profile when it performs the global static scheduling.

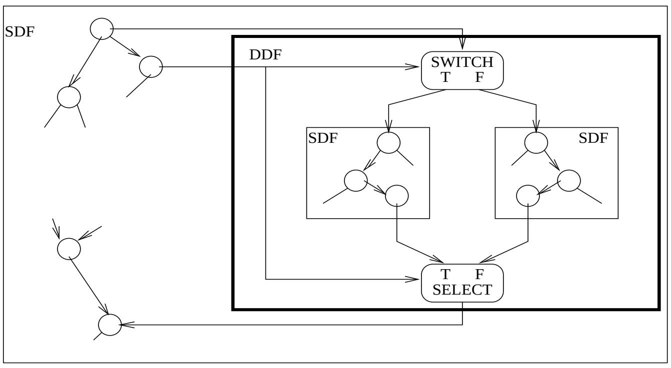

Fig. 9. An example of mixed domain system. The topmost level of the system is a SDF domain. A dynamic construct(if-then-else) is in the DDF domain, which in turn contains two subsystems in the SDF domain for its branches.

We apply the proposed scheduling technique to several synthetic examples as preliminary experiments. These experiments do not serve as a full test or proof of generality of the technique. However, they verify that the proposed technique can make better scheduling decisions than other simple but ad-hoc decisions on dynamic constructs in many applications. The target architecture is assumed to be a shared bus architecture with 4 processors, in which communication can be overlapped with computation.

To test the effectiveness of the proposed technique, we compare it with the following scheduling alternatives for the dynamic constructs.

Method 1. Assign all processors to each dynamic construct

Method 2. Assign only one processor to each dynamic construct

Method 3. Apply a fully dynamic scheduling ignoring all overhead

Method 4. Apply a fully static scheduling

Method 1 corresponds to the previous research on quasi-static scheduling technique made by Lee [10] and by Loeffler et. al. [13] for data dependent iterations. Method 2 approximately models the situation when we implement each dynamic construct as a single big atomic actor. To simulate the third model, we list all possible outcomes, each of which can be represented as a data-independent macro actor. With each possible outcome, we replace the dynamic construct with a data-independent macro actor and perform fully-static scheduling. The scheduling result from Method 3 is non-realistic since it ignores all the overhead of the fully dynamic scheduling strategy. Nonetheless, it will give a yardstick to measure the relative performance of other scheduling decisions. By modifying the dataflow graphs, we may use fully static scheduling in Method 4. For a conditional construct, we may evaluate both branches and select one by a multiplexor actor. For a data-dependent iteration construct, we always perform the worst case number of iterations. For comparison, we use the average schedule length of the program as the performance measure.

As an example, consider a **For** construct of data-dependent iteration as shown in figure 10. The number inside each actor represents the execution length. To increase the parallelism, we pipelined the graph at the beginning of the **For** construct.

The scheduling decisions to be made for the For construct are how many processors to be assigned to the iteration body and how many iteration cycles to be scheduled explicitly. We assume that the number of iteration cycles is uniformly distributed between 1 and 7. To determine the optimal number of assigned processors, we compare the expected total cost as shown in table I. Since the iteration body can utilize two processors effectively, the expected total cost of the first two columns are very close. However, the schedule determines that assigning one processor is slightly better. Rather than parallelizing the iteration body, the scheduler automatically parallelizes the iteration cycles. If we change the parameters, we may want to parallelize the iteration body first and the iteration cycles next.

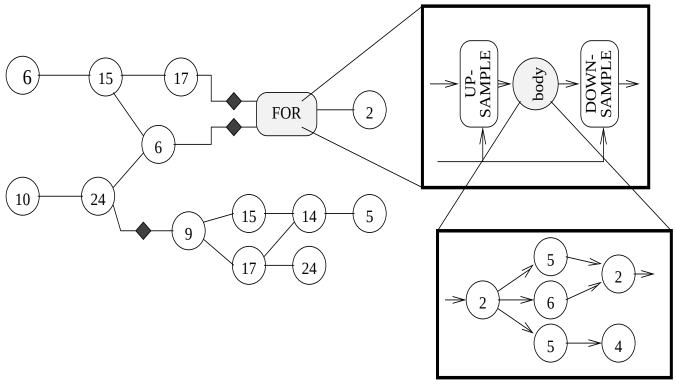

Fig. 10. An example with a **For** construct at the top level. The subsystems associated with the For construct are also displayed. The number inside an actor represents the execution length of the actor.

The proposed technique considers the tradeoffs of parallelizing inner loops or parallelizing outer loops in a nested loop construct, which has been the main problem of parallelizing compilers for sequential programs. The resulting Gantt chart for this example is shown in figure 11.

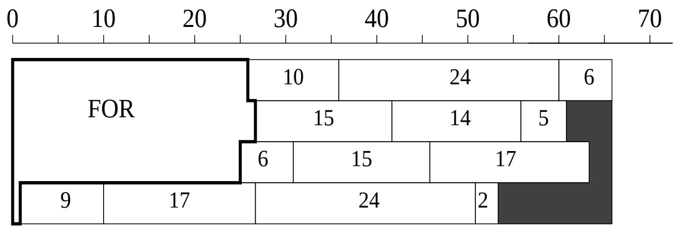

Fig. 11. A Gantt chart disply of the scheduling result over 4 processors from the proposed scheduling technique for the example in figure 10. The profile of the For construct is identified.

If the number of iteration cycles at run time is less than or equal to 3, the schedule length of the example is same as the schedule period 66. If it is greater than 3, the schedule length will increase. Therefore, the average schedule length of the example becomes 79.9. The average schedule length from other scheduling decisions are compared in table II. The proposed technique outperforms other realistic methods and achieves 85% of the ideal schedule length by Method 3. In this example, assigning 4 processors to the iteration body (Method 1) worsens the performance since it fails to exploit the intercycle parallelism. Confining the dynamic construct in a single actor (Method 2) gives the worst performance as expected since it fails to exploit both intercycle parallelism and the parallelism of the iteration body. Since the range of the number of iteration cycles is not big, assuming the worst case iteration (Method 4) is not bad.

This example, however, reveals a shortcoming of the proposed technique. If we assign 2 processors to the iteration body to exploit the parallelism of the iteration body as well as the intercycle parallelism, the average schedule length becomes 77.7, which is slightly better than the scheduling result by the proposed technique. When we calculate the expected total cost to decide the optimal number of processors to assign to the iteration body, we do not account for the global effect of the decision. Since the difference of the expected total costs between the proposed technique and the best scheduling was not significant, as shown in table I, this non-optimality of the proposed technique could be anticipated. To improve the performance of the proposed technique, we can add a heuristic that if the expected total cost is not significantly greater than the optimal one, we perform scheduling with that assigned number and compare the performance with the proposed technique to choose the best scheduling result.

The search for the assumed number of iteration cycles for the optimal profile is not faultless either, since the proposed technique finds a local optimum. The proposed technique selects 3 as the assumed number of iteration cycles. It is proved, however, that the best assumed number is 2 in this example even though the performance difference is negligible. Although the proposed technique is not always optimal, it is certainly better than any of the other scheduling methods demonstrated in table II.

Experiments with other dynamic constructs as well as nested constructs have been performed to obtain the similar results that the proposed technique outperforms other ad-hoc decisions. The resulting quasi-static schedule could be at least 10% faster than other scheduling decisions currently existent, while it is as little as 15 % slower than an ideal (and highly unrealistic) fully-dynamic schedule. In a nested dynamic construct, the compile-time profile of the inner dynamic construct affects that of the outer dynamic construct. In general, there is a trade-off between exploiting parallelism of the inner dynamic construct first and that of the outer construct first. The proposed technique resolves this conflict automatically. Refer to [17] for detailed discussion.

Let us assess the complexity of the proposed scheme. If the number of dynamic constructs including all nested ones is D and the number of processors is N, the total number of profile decision steps is order of ND, $O(ND)$. To determine the optimal profile also consumes $O(ND)$ time units. Therefore, the overall complexity is order of ND. The memory requirements are the same order od magnitude as the number of profiles to be maintained, which is also order of ND.

VIII. Conclusion

As long as the data-dependent behavior is not dominating in a dataflow program, the more scheduling decisions are made at compile time the better, since we can reduce the hardware and software overhead for scheduling at run time. For compile-time decision of task assignment and/or ordering, we need the static information, called profiles, of all actors. Most heuristics for compile-time decisions assume the static information of all tasks, or use ad-hoc approximations. In this paper, we propose a systematic method to decide on profiles for each dynamic construct. We define the compile-time profile of a dynamic construct as an assumed local schedule of the body of the dynamic

TABLE I
THE EXPECTED TOTAL COST OF THE FOR CONSTRUCT AS A FUNCTION OF THE NUMBER OF ASSIGNED PROCESSORS

Number of assigned processors	1	2	3	4
Expected total cost	129.9	135.9	177.9	N/A

TABLE II
PERFORMANCE COMPARISON AMONG SEVERAL SCHEDULING DECISIONS

Method	Proposed	1	2	3	4
Average schedule length	79.7	90.9	104.3	68.1	90
% of ideal	0.85	0.75	0.65	1.0	0.76

construct. We define the cost of a dynamic construct and choose its compile-time profile in order to minimize the expected cost. The cost of a dynamic construct is the sum of execution length of the construct and the idle time on all processors at run-time due to the difference between the compile-time profile and the actual run-time profile. We discussed in detail how to compute the profile of three kinds of common dynamic constructs: conditionals, data-dependent iterations, and recursion.

To compute the expected cost, we require that the statistical distribution of the dynamic behavior, for example the distribution of the number of iteration cycles for a data-dependent iteration, must be known or approximated at compile-time. For the particular examples we used for experiments, the performance does not degrade rapidly as the stochastic model deviates from the actual program behavior, suggesting that a compiler can use fairly simple techniques to estimate the model.

We implemented the technique in Ptolemy as a part of a rapid prototyping environment. We illustrated the effectiveness of the proposed technique with a synthetic example in this paper and with many other examples in [17]. The results are only a preliminary indication of the potential in practical applications, but they are very promising. While the proposed technique makes locally optimal decisions for each dynamic construct, it is shown that the proposed technique is effective when the amount of data dependency from a dynamic construct is small. But, we admittedly cannot quantify at what level the technique breaks down.

ACKNOWLEDGMENTS

The authors would like to gratefully thank the anonymous reviewers for their helpful suggestions. This research is part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Lucky-Goldstar, Mentor Graphics, Mitsubishi, Motorola, NEC, Philips, and, Rockwell.

REFERENCES

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

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

[3] S.Ha and E.A.Lee, "Compile-Time Scheduling and Assignment of Dataflow Program Graphs with Data-Dependent Iteration", *IEEE Trans. Computers*, November, 1991.

[4] E.G.Coffman,Jr., Computer and Job Scheduling Theory, Wiley, New York, 1976.

[5] M.J.Gonzalez, "Deterministic Processor Scheduling", *Computing Surveys*, 9(3), September, 1977.

[6] G.C.Sih, "Multiprocessor Scheduling to Account for Interprocessor Communications", Ph.D. Thesis, University of California, Berkeley, April, 1991.

[7] S.J.Kim and J.C.Browne, "A General Approach to Mapping of Parallel Computations Upon Multiprocessor Architecture", *Int. Conf. on Distributed Computing Systems*, pp. 1-8, 1988.

[8] K.Konstantinides, R.T.Kaneshiro, and J.R.Tani, "Task Allocation and Scheduling Models for Multiprocessor Digital Signal Processing", *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol.38, No.12, pp. 2151-2161, December, 1990. *Computing Surveys*, 9(3), September, 1977.

[9] G.N.S.Prasanna, A.Agarwal, and B.R.Musicus, "Hierarchical Compilation of Macro Dataflow Graphs for Multiprocessors with Local Memory", *IEEE Trans. on Parallel and Distributed Systems*, Vol.5, No.7, pp.720-736, July, 1994.

[10] E.A.Lee, "Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages", VLSI Signal Processing III, IEEE Press, 1988.

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

[12] 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.

[13] C.Loeffler, A.Lightenberg, H.Bheda, and G. Moschytz, "Hierarchical Scheduling Systems for Parallel Architectures", *Proceedings of Euco*, Grenoble, September, 1988.

[14] P.A.Suhler, J.Biswas, K.M.Korner, and J.C.Browne, "TDFL: A Task-Level Dataflow Language", *Journal of Parallel and Distributed Computing*, 9, pp. 103-115, 1990.

[15] J. Buck, S. Ha, E. A. Lee, and D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation* Vol. 4. April, pp. 155-182, 1994.

[16] P.Hoang and J.Rabaey, "Program Partitioning for a Reconfigurable Multiprocessor System", *IEEE Workshop on VLSI Signal Processing IV*, November, 1990.

[17] S. Ha, "Compile-Time Scheduling of Dataflow Program Graphs with Dynamic Constructs," Ph.D. dissertation, University of California, Berkeley, May, 1992.

Soonhoi Ha Soonhoi Ha is an assistant professor in the Computer Engineering Department at Seoul National University, Seoul, Korea. His research interests include architecture and scheduling techniques for parallel processing, design methodology for digital systems, and hardware-software codesign. He has been actively involved in the Ptolemy project since its birth. He received the B.S. and the M.S. degrees in electronics from Seoul National University, Seoul, Korea, in 1985 and 1987, respectively. He received the Ph.D. degree in the Electrical Engineering and Computer Science Department at U.C. Berkeley in 1992. He is a member of ACM and the IEEE Computer Society.

Edward A. Lee Edward A. Lee is a Professor in the Electrical Engineering and Computer Science Department at U.C. Berkeley. His research interests include real-time software, discrete-event systems, parallel computation, architecture and software techniques for signal processing, and design methodology for heterogeneous systems. He is director of the Ptolemy project at UC Berkeley. He is co-author of four books, numerous papers, and two patents. His bachelors degree (B.S.) is from Yale University (1979), his masters (S.M.) from MIT (1981), and his Ph.D. from U. C. Berkeley (1986). From 1979 to 1982 he was a member of technical staff at Bell Telephone Laboratories in Holmdel, New Jersey, in the Advanced Data Communications Laboratory. He is a founder of Berkeley Design Technology, Inc. and has consulted for a number of other companies. He is a fellow of the IEEE and was an NSF Presidential Young Investigator.